Concurrency and Multithreading In Java - An Overview

Lakshmi Mareddy

Bellevue University

College of Information Technology, Bellevue University, Nebraska 68005. This paper is a partial fulfillment of the course CIS 602A-Intermediate Java Programming taught by Prof. Lomax.

Table of Contents

Abstract

Java has inbuilt support for multithread programming. Prior to Tiger (J2SE5.0) version, programmers tackled threading tasks using traditional approaches such as synchronized blocks, wait and notify calls. They had to write their own rules to achieve synchronization, affecting resources, performance and code scalability. Through its concurrent package, the Tiger version of Java has introduced concurrency utilities that help multithreaded applications and servers. This paper will analyze some of the utility classes, the Application Interfaces (APIs) available their benefits as well as their downside while implementing them.

Keywords: multithreading, concurrency, utilities, J2SE, API

Concurrency and Multithreading In Java - An Overview

Most tasks these days involve multitasking such as online gaming, online document editing such as Google docs, and developing code online in a cloud environment such as Google App Engine, Microsoft Azure, and Amazon AWS. Such offerings depend on concurrent programming, which is the ability to run parallel tasks at the same. With the dawn of dual core and multi-core processors, and the emergence of cloud computing, it is necessary for programmers to purchase the full potential of options offered by both the hardware and software.

Threads support user centric GUI applications and aid in server applications where resource utilization and throughput are critical. In the JVM, the garbage collector usually runs in one or more dedicated threads. (Goetz p.216)

A multi-core environment performs better if the implementing language also supports multithreading. The introduction of the new concurrent package in Java is a big plus for multithread programming. J2SE5 has introduced new interfaces, implementation classes and methods in the concurrent package. This paper will examine the benefits and drawbacks if any, of these implementations.

Challenges faced by concurrency

Some of the major challenges faced by concurrency are race conditions, slipped condition, busy waiting, spurious wakeups and deadlocks. According to Jenkov (n.d.) "The situation where two threads compete for the same resource, where the sequence in which the resource is accessed is significant, is called race conditions. A code section that leads to race conditions is called a critical section."

Christopher, T. W., & Thiruvathukal, G. K. (2000) have stated that race conditions, deadlock and starvation are problems created by concurrency. They have defined race

conditions as “Threads can try to update the same data structure at the same time. The result can be partly what one thread wrote and partly what the other thread wrote.” (p.44), this could lead to mixed up data, as a result.

In a multithread environment, if there is a very long wait to access a resource, it is defined as a busy wait. Busy waits can happen if certain threads have more priority than others, and are constantly given access. A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup (Oracle.com).

The threads that are forever waiting for access can lead to a starvation scenario. It can be countered by fairness, where all threads are allowed a turn. Fairness usually leads to a slowing down of processes (Christopher, T. W., & Thiruvathukal, G. K., 2000).

According to Jenkov (n.d.) “from the time a thread has checked a certain condition until it acts upon it, the condition has been changed by another thread so that it is errorneous for the first thread to act”. The change in state or value of the condition is called a “slipped condition”.

Sometimes thread X waits on a lock release by thread Y, while thread Y waits on a lock release by thread X. This leads to a deadlock situation. Sometimes “multiple threads ask for locks but get them in a different order than they asked for” (Jenkov n.d.), leading to a deadlock.

Java Concurrency Interfaces, Classes and Methods

J2SE5 provides a number of utility classes for multithreaded applications. These classes can be found in the java.util.concurrent, java.util.concurrent.atomic, and java.util.concurrent.locks packages. In this section, the paper will discuss some of the key interfaces, classes and methods available in these packages, that have been introduced in J2SE5 and that support multithreading.

Concurrent Collections

Concurrent collections are designed for concurrent access from multiple threads.

Demir (2010) has described concurrent collections as:

> Since JDK 1.5, Java provides a rich set of collection classes that are designed to help increase the scalability of multi-threaded applications by reducing the scope of exclusive locks while working on data sets. They also provide locking to guard compound actions such as iteration, navigation, and conditional operations (put-if-absent).

Concurrent collections have iterators that are thread safe, but cannot account for or reflect changes in the collection size, as it iterates (weakly consistent). Java 5.0 adds ConcurrentHashMap and CopyOnWriteArrayList for situations where travelling through a collection is the important aspect. If earlier implementations were achieved using Hashtable or synchronizedMap , the newer ConcurrentHashMap can do the same work, in a thread safe manner.

Queues: BlockingQueue Interface

Prior to J2SE5, programmers had to build a thread safe queue themselves and take care of synchronization points. With the introduction of the Queue class, built-in methods and synchronization is available to the programmer.

It is a collection class that implements the first in first out (FIFO) data structure. It also has other implementations such as a priority queue and a stack. Traditionally a queue is a combination of a provider, such as a client request and a consumer, such as a server that listens for client requests. A client adds requests to a queue and a server reads requests off it. When the queue is empty, the server waits for the next request.

The BlockingQueue interface is a Queue subclass. Langr (2006) demonstrated the implementation of BlockingQueue as seen in CodeExample 1. In the example, a server polls for client requests, and adds them to a queue. A separate server thread loops indefinitely, asking the BlockingQueue to indicate when a request is available. Once a request is available, it's removed from the queue and handled by the server.

```
import java.util.concurrent.*;

public class Server extends Thread {
    private BlockingQueue<Request> queue =
        new LinkedBlockingQueue<Request>();

    public void accept(Request request) {
        queue.add(request);
    }

     public void run() {
        while (true)
           try {  execute(queue.take());  }
           catch (InterruptedException e) {  }
    }

     private void execute(final Request request) {
        new Thread(new Runnable() {
           public void run() { request.execute(); }
        }).start();
   }
}
```

CodeExample 1: Simple Server using BlockedQueue

ConcurrentHashMap

The ConcurrentHashMap class is a thread-safe implementation of ConcurrentMap that offers far better concurrency than the earlier synchronizedMap (Goetz 2003). This class supports multiple reads, multiple readwrites and multiple writes concurrently.

This class is designed to optimize retrieval operations. The get operation does not need locking. When queried it returns the value of the most recent insert (either completed or in

progress), and the concept is termed “adjustable expected concurrency”. ConcurrentHashmaps are used in situations where there is no need to lock the entire table in order to prevent updates.

ConcurrentHashMaps provide far superior scalability over Hashtable, without compromising its effectiveness in a wide variety of common-use cases, such as shared caches (Goetz 2003).

Callable vs Runnable Interfaces

The Callable and Runnable interfaces are designed for classes whose instances are executed by another thread. A thread can be created in two ways; by implementing the Runnable class and calling the run method or by overriding the run method from a subclass of Thread.

Callable is a relatively new interface In Java 5 and it was introduced as a part of concurrency package. Both Callable and Runnable can be used with executors. The advantage of Callable is that one can send it to an executor and immediately get back a Future result that gets updated when the execution is finished. When we use a Runnable interface the results have to be managed by the developer.

Future Interface

Whenever a callable is submitted to an executor, the framework returns a Future. The methods in this interface allow us to determine the status of the callable and to get results from the callable. The Future holds the result of the computation executed by the callable.

Methods such as isDone, cancel, isCancelled and get are implemented through its class FutureTask. The isDone method lets us know whether a task was completed irrespective of a completion or cancellation. The isCancelled method helps us know if a task was cancelled. Once a computation has completed, the computation cannot be cancelled.

According to Jenkov (n.d.), a task might terminate because of an exception and not because it has successfully completed. Based on the Future returned, there is no way to tell, how it completed.

Executors

The most important new feature in J2SE5 is the Executor framework. The framework relieves the programmer of drudgery such as creating thread pools and managing memory related tasks such as heaps and stacks.

A java.util.concurrent.Executor is the interface that executes Runnable tasks. An Executor is important in a multithreaded application where creating new threads can turn out to be expensive. The Executor framework separates job submission from job execution which includes thread usage, scheduling and so on. Instead of manually creating new threads, one can simply add threads to an Executor, and it takes care of the maintenance of the same. The interface is implemented by the classes provided such as 'ScheduledThreadPoolExecutor'.

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 2";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 3";
    }
});
List<Future<String>> futures = executorService.invokeAll(callables);
for(Future<String> future : futures){
    System.out.println("future.get = " + future.get());
}
executorService.shutdown();
```

CodeExample 2: Example of an ExecutorService

Executor tasks include creating and maintaining thread pools or performing asynchronous IO operations. The tasks can run in a new thread, in an existing thread or within a thread that calls the execute() method. The type of task being executed is based on the Executor class being implemented.

The Executor runs and completes tasks by reusing threads from the thread pool as needed, without always creating new threads. The threads can be stopped with the command executor.shutdown();. A composite Executor, serializes tasks using a BlockingQueue and hands them to another Executor.

Synchronizers

A synchronizer is any object that coordinates the control flow of threads based on its state. Blocking queues can act as synchronizers; other types of synchronizers include semaphores, barriers, and latches.

In Java you can mark a method or a block of code as synchronized. For threads held by the same lock, when synchronized the Java Memory Model (JMM) guarantees that it can see the effects of all previous modifications that were guarded by the same lock (Oracle.com).

```
synchronized (obj) {
         while (<condition does not hold>)
             obj.wait(timeout);
         ... // Perform action appropriate to condition
     }
```

CodeExample 3: Example of a synchronized block

Semaphore

The semaphore class has been introduced from Java 5. It is a protected variable that allows a fixed number of threads access to a resource and acts as a gatekeeper. It also maintains the availability status of a resource.

For instance, there may be several threads trying to enter a fixed amount of server sessions. A semaphore keeps the count of how many sessions the server allows. The minute a session is closed, a notice is sent to the semaphore by the quitting thread through a release method. The semaphore now grants access to the next thread through a permit. A thread accesses the resource via the acquire method. At this time, there is no overt lock held by the thread, as the acquire method prevents the thread being returned to the semaphore, until the task is done. A semaphore can restrict the number of threads accessing a resource through a parameter setting.

One can set a fairness parameter in a semaphore, allowing all threads an equal chance. When fairness is set true, the semaphore guarantees that threads gain access to the resource in a first in first out (FIFO) manner. When fairness is not set to true, a thread can gain priority over other threads. Similarly if a thread tries to gain access using an untimed tryAcquire method, it can also get priority and can jump the queue.

A binary semaphore has only two states available and acts as a mutual exclusion lock. Since there is no concept of ownership, any thread can unlock the lock. This is a useful property, especially when overcoming a possible deadlock.

CyclicBarrier

A CyclicBarrier waits for all threads to complete their tasks, and reach a certain point, before they are released. Suppose a group of threads are performing an interconnected task such

as in a bouncing ball game, where there are more than three balls, and they occasionally collide with each other. Each thread will calculate the speed and angle of a single ball. The threads also need to calculate positions with respect to each other, to compute relative positions. The system cannot go to the next step, until the individual calculation and referential calculations are done for every ball, and then they start all over again. In this model, one thread may be ahead of another, but will have to wait until all threads have finished their calculations. This way all the motion calculations will be computed correctly. Such a scenario would employ the barrier synchronization technique to perform their calculations.

Latches

A latch is a Boolean condition that is set once. Once the lock is released, it will not be available. A synchronization object that behaves in this way is the CountDownLatch - once "open" it never closes and cannot be reset.

A CountDownLatch is initialized with a given count. All threads wait until the current count reaches zero and then they are released. The count is kicked off by the countDown() method. Once it is set to zero, it cannot start again. Consider the example where a set of threads have to be created in an application. If the program cannot start unless all the threads have initialized, a CountDownLatch is the best way to implement this.

Exchangers

An Exchanger is a synchronization point at which two threads can exchange objects. While entering the exchange method, each thread will offer an object to the other, and take the other thread’s object, and leaves. Typical usage is when an Exchanger swaps data between threads, or fills an empty thread.

Lock Interface

One of the mechanisms that Java offers for communicating between threads is synchronization. Synchronization is implemented using monitors. An object in Java is associated with a monitor, which a thread can lock or unlock. At any time, only one thread can hold a lock on a monitor. All other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor. A thread may lock a particular monitor multiple times and each unlock reverses the effect of one lock operation (Oracle.com)

A synchronized statement first performs a lock action on that object's monitor and does not proceed further until the lock action has successfully completed. Once the object is locked, the body of the statement is executed. The lock is released for both successful and terminated executions.

The three forms of lock acquisition (interruptible, non-interruptible, and timed) may differ in their performance characteristics, ordering guarantees, or other implementation qualities.

ReadWriteLock

A lock controls the access to a shared resource, by multiple threads. Normally the access is granted one thread at a time, to threads that have acquired the lock. A ReadWriteLock can offer concurrent access where one thread writes and several other threads read. Locks offer more operations than the synchronized method.

In a mutual exclusion lock, the threads have to wait in sequence, until they can acquire a lock, to perform their activity. The read-write lock was developed to improve concurrency, where only one thread writes and the rest read from the resource, thereby improving performance.

In practice this increase in concurrency will only be fully realized on a multi-processor, and then only if the access patterns for the shared data are suitable. (Oracle.com)

```
public class ReadWriteLock{
  private int readers       = 0;
  private int writers       = 0;
  private int writeRequests = 0;
  public synchronized void lockRead() throws InterruptedException{
    while(writers > 0 || writeRequests > 0){
      wait();
    }
    readers++;
  }
public synchronized void unlockRead(){
    readers--;
    notifyAll();
  }
public synchronized void lockWrite() throws InterruptedException{
    writeRequests++;
while(readers > 0 || writers > 0){
      wait();
    }
writeRequests--;
writers++;
  }
public synchronized void unlockWrite() throws InterruptedException{
    writers--;
    notifyAll();
  }
}
```

CodeExample 4: A ReadWritelock implementation

ReentrantLock

A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. This can be checked using methods isHeldByCurrentThread(), and getHoldCount(). Reentrant locks do not perform as well as non reentrant locks, and may be hard to implement.

When the fairness parameter is set in the constructor under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order. Programs using fair locks accessed by many threads may display lower

overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. (Oracle.com)

```
import java.util.concurrent.locks.ReentrantLock;

final ReentrantLock _lock = new ReentrantLock();

private void method() throws InterruptedException
{
    //Trying to enter the critical section
    _lock.lock(); // will wait until this thread gets the lock
    try
    {
        // critical section
    }
    finally
    {
        //releasing the lock so that other threads can get notified
        _lock.unlock();
    }
}
```

CodeExample 5: A ReentrantLock Implementation

In a synchronized block, the locks have to be released in the reverse sequence of the acquisition, and all within the same block. This is good to contain the monitor locks within one scope, there can be situations where lock A is acquired in one scope and is released in an altogether different scope B. Sometimes the activity may involve acquiring locks A and B, releasing A and acquiring C, releasing B and acquiring D, which is the "hand over hand" or "chain locking" technique encountered while traversing data structures.

Implementations of the Lock interface enables the use of techniques such as allowing a lock to be acquired and released in different scopes, and allowing multiple locks to be acquired and released in any order. The ability to release locks in different order, helps reduce the chance of a deadlock.

As Java can neither predict nor prevent deadlock conditions, programs should be written using higher-level locking primitives that don't deadlock. The use of volatile variables and classes provided in the java.util.concurrent package is an alternative to using synchronization.

Atomic Variables

The java.util.concurrent.atomic package provides classes for automatically manipulating single variables, which can be primitive types or references. These implementations have higher performance and speed over synchronization. Typical uses are for implementing high performance concurrent algorithms, counters and sequence number generators.

According to Lorimer (2005), Atomic variables provide the synchronization-like safety of locks, without explicitly using locks. They do this by relying on operating-system specific locking mechanisms whenever possible, which are typically much faster than the high-level Java locking mechanisms.

```
class AtomicIdGenerator {
    private static AtomicLong id = new AtomicLong(0);
    public static long nextId() {
        long next = id.incrementAndGet();
        return next;
    }
}
```

CodeExample 6: Atomic Variable as an ID generator

Java Memory Model

Modern hardware can consist of several CPUs or a single CPU with multicore processors. They can possess caches both on and off the chip. This in turn can lead to instructions being executed in parallel or any random order, from when they receive it. Sometimes the memory may not even be sitting on the RAM but on a remote core register.

The JMM describes when one thread's actions are guaranteed to be visible to another. Prior to Java 5, memory models were specific to each processor's architecture. From Java 5 onwards, the memory model is truly cross platform, and the same source code can be run anywhere.

Wong (2009) in his article stated that:

> But between threads, if you don't use synchronized or volatile, there are no visibility guarantees. As far as visible results go, there is no guarantee that thread A will see them in the order that thread B executes them. One must use synchronization to get inter-thread visibility guarantees.

The JMM offers some guarantees when certain keywords are used. According to the structuring of the Java memory model, unlock on a monitor happens-before every subsequent lock on that monitor. When using the volatile keyword, a write to a volatile variable happens-before subsequent reads of that variable. Static initialization is done by the class loader, so the JVM guarantees thread safety. A call to start () on a thread happens-before any actions in the started thread. All actions in a thread happen-before any other thread successfully returns from a join() on that thread. The default initialization of any object happens-before any other actions (other than default-writes) of a program. In addition to the above, the JMM offers a guarantee of initialization safety for immutable objects.

Conclusion

J2SE5 handles multithreading activities much better than its previous versions. The new interfaces, classes and implementers help fix some of the challenges faced in a multithreaded environment. The JMM also has been improved and it supports the classes and methods.

In his paper, Wong (2009) states that the primary danger in multithreading is in a shared, mutable state. Even though the JMM assures us of cross-platform visibility guarantees, minimizing shared mutable data is a better approach. He mentions Scala's Actor construct which relies on passing immutable messages instead of sharing memory.

The Semaphore class provides convenience methods to acquire and release multiple permits at a time. (Oracle.com) There is an increased risk of indefinite postponement when these methods are used without fairness set true. On the other hand, setting fairness to true slows down the process.

Even though Java5 has introduced lock fairness, it does not guarantee fairness of thread scheduling. Thus, one of many threads using a fair lock may obtain it multiple times in succession while other active threads are not progressing and not currently holding the lock. The untimed tryLock() method does not honor the fairness setting. It will succeed if the lock is available even if other threads are waiting. Similarly the untimed tryAcquire() in a Semaphore will also acquire whatever permits are available. These methods are useful, when we try to recover from a deadlock, or jump into a queue. (Oracle.com)

In addition to this J2SE5 provides out-of-the-box remote monitoring and management on the Java platform and of applications that run on it. JDK 5.0 includes the Java Monitoring and Management Console (JConsole) tool (Chung 2006). The tool helps access several core monitoring and management functionalities provided by the Java platform including but not limited to detection of low memory and deadlock detection. This helps in detecting concurrency chokes and aids in writing better code.

According to Sletten (2008), concurrency structures such as Syncs, Barriers, and Rendezvouses were introduced to implement priority queues in a bid to improve performance.

Complicated processes such as web service calls, querying databases or transforming XML affect the way multithreads interact, as the underlying business rules can change (Sletten 2008), but it may not be reflected in the multithreaded interactions.

In a modern cloud scenario, the key to scaling is the ability to separate code into executable blocks. One cannot rely on just language level constructs. If this is not possible, the ability to take advantage of extra CPUs and truly scale up will be affected.

Inspite of Java's rich and powerful platform and language-level features, using them can be difficult. These issues are pushing developers to consider alternate languages like Erlang, Scala, and Clojure (Sletten 2008). These languages are classified as functional and hybrid and have introduced concepts such as Software Transactional Memory. They have reduced the amount of state maintenance and the need for concurrency locks. As both Scala and Clojure run on the JVM, the amount of Java code reuse is high (Sletten 2008). Because of all these factors, developers are moving away from pure Java implementations in favor of these functional languages, where not just the language, but the code is also pro concurrency.

Overall, the application being developed, the environment such as single core, multi core, the services such as stand alone, or web and amount of scalability such as a cloud application, all determine the kind of approach and choice of classes and methods. But should a developer need it, Java 5 has some excellent solutions built into it to implement various multithreaded applications.

References

java.util.concurrent (Java 2 Platform SE 5.0) . (n.d.). *Automatic Redirection*. Retrieved March 22, 2011, from http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/package-summary.html

Borgers, J. (2008, June 18). InfoQ: Do Java 6 threading optimizations actually work?. *InfoQ: Tracking change and innovation in the enterprise software development community*. Retrieved May 1, 2011, from http://www.infoq.com/articles/java-threading-optimizations-p1

Callable interface vs Runnable interface in Java | Geek Explains: Java, J2EE, Oracle, Puzzles, and Problem Solving. (n.d.). *Geek Explains: Java, J2EE, Oracle, Puzzles, and Problem Solving*. Retrieved April 20, 2011, from http://geekexplains.blogspot.com/2008/05/callable-interface-vs-runnable.html

Christopher, T. W., & Thiruvathukal, G. K. (2000). *High-performance Java platform computing: multithreaded and networked programming*. Upper Saddle River, N.J: Prentice Hall.

Chung, M. (2006). Monitoring and Managing Java SE 6 Platform Applications. *Oracle Technology Network for Java Developers*. Retrieved April 22, 2011, from http://java.sun.com/developer/technicalArticles/J2SE/monitoring/

Demir, D. (2010, October 19). Concurrent Collections from Java « Deniz Demir's Blog. *Deniz Demir's Blog*. Retrieved May 6, 2011, from http://denizdemir.com/2010/10/19/concurrent-collections-from-java/

Goetz, B. (2003, July 23). Java theory and practice: Concurrent collections classes. *IBM - United*

States. Retrieved May 4, 2011, from http://www.ibm.com/developerworks/java/library/j-jtp07233/index.html

Goetz, B. (2006). *Java concurrency in practice*. Upper Saddle River, NJ : Addison-Wesley.

Jenkov, J. (n.d.). Java Concurrency Tutorial. *http://jenkov.com*. Retrieved April 18, 2011, from tutorials.jenkov.com/java-concurrency/index.html

Jenkov, J. (n.d.). Java Concurrency - Locks. *www.jenkov.com*. Retrieved April 18, 2011, from tutorials.jenkov.com/java-concurrency/locks.html

Jenkov, J. (n.d.). Java Concurrency - Thread Signalling. *Java Concurrency*. Retrieved April 20, 2011, from http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#sharedobjects

Johnson, R., Hoeller, J., Arendsen, A., Sampaleanu, C., & Harrop, R. (2008, May 29). Chapter 23. Scheduling and Thread Pooling. *SpringSource.org* | . Retrieved April 26, 2011, from http://static.springsource.org/spring/docs/2.0.5/reference/scheduling.html

Langr, J. (2006, November 21). Java 5's BlockingQueue - Developer.com. *Developer.com: Your Home for Java and Open Source Development Knowledge*. Retrieved May 1, 2011, from http://www.developer.com/java/ent/article.php/3645111/Java-5s-BlockingQueue.htm

Lea, D., Scott, M., & Scherer, B. (n.d.). java.util.concurrent: Exchanger.java. *DocJar*. Retrieved May 8, 2011, from http://www.docjar.com/html/api/java/util/concurrent/Exchanger.java.html

Learning Java - Chapter 10 : Java . (n.d.). *PAP2009*. Retrieved May 2, 2011, from http://www.particle.kth.se/~lindsey/JavaCourse/Book/Part1/Java/Chapter10/concurrencyTools.html

Lorimer, R. J. (2005, July 17). Concurrency: Ensure Thread Safety On Single Values With Atomic Variables. *Javalobby | The heart of the Java developer community*. Retrieved April 28, 2011, from http://www.javalobby.org/java/forums/t19676.html

Luo, Z. D., Buchbinder, Y. N., & Das, R. (n.d.). Java concurrency bug patterns for multicore systems. *IBM - United States*. Retrieved March 22, 2011, from http://www.ibm.com/developerworks/java/library/j-concurrencybugpatterns/

Mahmoud, Q. H. (n.d.). Concurrent Programming with J2SE 5.0. *Oracle Technology Network for Java Developers*. Retrieved March 22, 2011, from http://java.sun.com/developer/technicalArticles/J2SE/concurrency/

Manson, J. (2009, June 17). Java Concurrency (&c): Volatile Arrays in Java. *Java Concurrency (&c)*. Retrieved May 9, 2011, from http://jeremymanson.blogspot.com/2009/06/volatile-arrays-in-java.html

McNaughton, A. (2010, November 14). Multi-Threading in a Java Environment - Intel® Software Network - Intel® Software Network. *Intel Software Network communities - Intel® Software Network - Intel® Software Network*. Retrieved April 10, 2011, from http://software.intel.com/en-us/articles/multi-threading-in-a-java-environment/

Morris, S. (2009, May 12). An Introduction to Concurrent Java Programming. *InformIT: An Introduction to Concurrent Java Programming > Why Concurrent Programming?*. Retrieved March 21, 2011, from www.informit.com/articles/article.aspx?p=1339471

Sletten, B. (2008, November 21). NetKernel: Moving Beyond Java™ Concurrency. *DevX - Your Information Source for Enterprise Application Development*. Retrieved March 22, 2011, from http://www.devx.com/Java/Article/39973

The Java Language Specification, Third Edition - TOC. (n.d.). *Oracle Technology Network for Java Developers*. Retrieved April 19, 2011, from http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

Threads and Locks. (n.d.). *Oracle Technology Network for Java Developers*. Retrieved May 6, 2011, from http://java.sun.com/docs/books/jls/third_edition/html/memory.html

Wong, C. (2009, October 5). Multithreading and the Java Memory Model | Javalobby. *Javalobby | The heart of the Java developer community*. Retrieved April 28, 2011, from http://java.dzone.com/articles/multithreading-and-java-memory