Full Version

Advanced Introduction to Java Multi-Threading

LAU Chok Sheak, July 2012
chok.lau@citi.com

Programmer/Analyst
Trading Services Team,
Citi Equities Technology

Synopsis

What is an "advanced introduction to Java multi-threading"?

Advanced	Talks about concepts not easily obvious to Java developers.
Introduction	Broad overview of what can be done with Java threads.
Multi-threading	Talks about interaction among different threads.

This presentation is suitable for Java developers at all levels (beginner to expert).

Synopsis (cont'd)

Why is an advanced introduction needed?

1. Goes underneath the hood of JVM to see what kind of multi-threaded code is correct and what isn’t.

2. Explores utilities and libraries available to the Standard JDK developer.

Structure of this Presentation

This presentation has two parts:

Part I. Java memory model

Part II. Concurrent utilities

Part I. Java Memory Model

Question:

Do I need to know what is the Java Memory Model?

Answer:

If you only write single-threaded programs => No.

If you write multi-threaded programs => Yes.

What is the Java Memory Model (JMM)?

- Some good, standard definitions can be easily found online.
- The JMM is defined in JSR-133.

To rephrase in my own way, the JMM is:

A. **Machine-level** – A set of rules to define how CPU caches should be affected by Java code.

B. **Source-level** – A set of rules to define whether a multi-threaded Java program will work correctly or not, given three areas of consideration:
   1. Visibility
   2. Ordering
   3. Atomicity

JMM – Sample Code 1: Non-volatile conflicts

Consider the following Java code snippet:

Initialization (both not volatile)	**Thread 1 (not synchronized)**	**Thread 2 (not synchronized)**

int n = 0;
boolean v = false;
	n = 1;
	v = true;
		if (v == true)
		System.out.println("n=" + n);

Question: Is it possible for Thread 2 to print “n=0”?

JMM – Sample Code 1: Non-volatile conflicts (cont'd)

Initialization (both not volatile)	**Thread 1 (not synchronized)**	**Thread 2 (not synchronized)**

int n = 0;
boolean v = false;
	n = 1;
	v = true;
		if (v == true)
		System.out.println("n=" + n);

Answer: Yes! (may print “n=0”)

Why? v=true might come before n=1 in compiler reorderings. Also n=1 might not become visible in Thread 2, so Thread 2 will load n=0.

Fix: If you make v as volatile, then Thread 2 can only print “n=1”. n does not need to be volatile. n cannot be reordered to come after v=true when v is volatile.

JMM – Sample Code 2: Synchronized block

Initialization	**Thread 1**	**Thread 2**

int x = 0;
int y = 0;
final Object obj = new Object();
	`synchronized (obj) {`
	`    x = x + 1;`
	`    y = x;`
	`}`
		`synchronized (obj) {`
		`    if (x != y)`
		`        System.out.println("x!=y");`
		`}`

Execution: Threads 1 and 2 run their own code repeatedly in their own infinite loops. Both x and y are not volatile. Synchronization is by obj, not by the instance object "this".

Question: Is it possible for Thread 2 to print "x!=y"?

JMM – Sample Code 2: Synchronized block (cont'd)

Initialization	**Thread 1**	**Thread 2**

int x = 0;
int y = 0;
final Object obj = new Object();
	synchronized (obj) {
	    x = x + 1;
	    y = x;
	}
		synchronized (obj) {
		    if (x != y)
		        System.out.println("x!=y");
		}

Answer: No! (will never print “x!=y”)

Why? The values of both x and y get synchronized between Threads 1 and 2 by implicit memory barriers across the synchronized block boundaries. All values will be synchronized even if they are not volatile. The synchronization object does not have to be the owning class instance object "this". It can be any object.

JMM – Sample Code 3: Double-checked locking

```
public class MyClass {
  private MyClass() {}
  private static MyClass instance;
  public static MyClass
   getInstance() {
    if (instance == null)
    synchronized (MyClass.class) {
            if (instance == null)
                       instance = new
   MyClass();
    }
    return instance;
  }
}
```

Question: Is it possible to create two or more instances of MyClass using getInstance() when this code is run in multi-threaded mode?

Hint: Does "instance" need to be volatile in order for this to work correctly?

JMM – Sample Code 3: Double-checked locking (2)

Answer: No, but the code is buggy.

Why? "instance" must be volatile, otherwise the invocation of the constructor could be reordered out of the synchronized block and a different thread might get "instance" in an uninitialized state even though it is not possible to create two instances here.

JMM – Sample Code 3: Double-checked locking (3)

Some people claim that this code:

```
instance = new MyClass();
```

Can be compiled into this code (which is correct):

```
A. instance = create new uninitialized object
B. invoke constructor on instance
```

Which means that between points A and B, instance is in an uninitialized state, and another thread may obtain a reference to it. Therefore this code is only correct in a single-threaded context, but not in a multithreaded context. This is possible when instance is not volatile and thus can be reordered!

JMM and Double-Checked Locking

There are two cases to consider to see whether you should use double-checked locking or not:

Case 1: Instance initialization is cheap.

If it is very quick to create a new instance and the instance does not take up much memory, then you should simply use:

public static final MyClass instance = new MyClass();

There is no need to save on creating this instance if it does not require much resources.

JMM and Double-Checked Locking (2)

Case 2: Instance initialization is expensive.

Some instances might be expensive to create, then lazy init should be used. Examples of expensive init:

- Runs a heavyweight algorithm to compute certain values.
- Consumes much memory (maybe over a few thousand bytes)
- Creates a new thread (could be a new java.lang.Timer, which is a thread)

JMM and Double-Checked Locking (3)

Then you can use the below code to do lazy initialization:

```
public class SingletonExample {
  private static final class
   InnerSingleton {
    private static final SingletonExample
   s =
       new SingletonExample();
  }
  public static SingletonExample
   getInstance() {
    return InnerSingleton.s;
  }
}
```

- Guaranteed to be visible to all threads that use that class.
- Inner class is not loaded until some thread references one of its fields or methods.
- Maybe looks a bit cleaner than using double-checked locking.
- Uses classloader synchronization, which could be a bit more efficient.

JMM and Double-Checked Locking (4)

- Joshua Bloch recommends using volatile with local variable to cache the field value.
- http://java.sun.com/developer/technicalArticles/Interviews/bloch_effective_08_qa.html)
- Feel free to use this idiom if you like.

Part I. Java Memory Model

A. Machine-Level Memory Model

B. Source-Level Memory Model

Machine-Level Memory Model

- The JMM is a noteworthy achievement.
- Given that every CPU architecture (x86, ia64, ARM, PPC etc.) is different, you can really appreciate that Java has defined the JMM to standardize the execution of all Java code when executed on any platform.
- Similar thing is done for the C# CLR. (maybe for my next presentation!)

Machine-Level Memory Model (2)

Given a particular variable x (whether static, instance, volatile or anything) in Java, when you access (read or write) the variable, you could be referring to one or all of these physical entities:

- CPU register (super-fast)
- CPU cache (very fast)
- Main memory (slow)

Machine-Level Memory Model (3)

In order to synchronize accesses to these machine-level entities, you have to make sure that the following software observe the same JMM rules:

1. Java compiler and its optimizations (including instruction reorderings)
2. JVM and its emitted memory barrier instructions (including instruction reorderings)

Memory Barriers

<u>What is a Memory Barrier?</u>

It is a machine-level instruction to ensure that any previous loads/stores/monitors in any thread will be made visible to the current load/store/monitor in a particular thread.

They are meaningful only when you talk about multi-threaded programs.

Only JVMs (not Java bytecode compilers) need to worry about Memory Barriers. However, Java developers should also be conscious of it because it might come in handy.

Types of Java Memory Operations

All Java memory operations are one of six types:

1. Normal Load (non-volatile)
2. Normal Store (non-volatile)
3. Volatile Load
4. Volatile Store
5. Monitor Enter
6. Monitor Exit

Types of Java Memory Barriers (All theoretically possible)

LoadLoad	LoadStore	LoadEnter	LoadExit
StoreLoad	StoreStore	StoreEnter	StoreExit
EnterLoad	EnterStore	EnterEnter	EnterExit
ExitLoad	ExitStore	ExitEnter	ExitExit

So basically, all combinations of regex "((Load)|(Store)|(Enter)|(Exit)){2}" - 24 possibilities.

Memory Barriers are used to enforce strict ordering or reading from main memory or flushing caches to main memory.

Any more details would be beyond the scope of this presentation, but you may feel free to read up further on your own.

Part I. Java Memory Model

A. Machine-Level Memory Model

B. Source-Level Memory Model ⇦

How to Use Memory Barriers in Java?

Java Developers have indirect control of the usage of memory barriers by using these Java keywords:

1. volatile
2. synchronized
3. final

Many details in JSR-133. We will cover some, but not all.

JMM – 1. Visibility

Rule 1: Two accesses (reads or writes) to the same variable are conflicting if any one of them is a write.

Rule 2: For two conflicting volatile accesses, or non-volatile accesses across a synchronized block boundary, they are guaranteed to happen one after the other, and the later access will never read/overwrite a stale value.

Rule 3: A volatile read can never read a value that has never been written to it, i.e. cannot create a value out of thin air.

Rule 4: You will always read the same value from a final variable after the constructor setting the final variable exits. (If you try to read it before the constructor exits, you might read a different value.)

JMM – 2. Ordering

When we say "instructions reordering", we are referring to both the Java bytecode compiler and JVM hotspot compiler.

Rule 1: Instructions can never be reordered around volatile variables.

Rule 2: Instructions can never be reordered around synchronization boundaries, except when we are moving instructions into the critical section.

Rule 3: Accesses to final fields cannot be reordered.

JMM – 3. Atomicity

Rule 1: Word-tearing: Bytes, booleans, shorts, integers and floats will never be overwritten by adjacent writes (whether in array or not).

Rule 2: Reads and writes of volatile doubles and longs are always atomic, but no guarantee of atomicity for non-volatile doubles and longs.

Rule 3: Reads and writes of references (whether 32-bit or 64-bit) are always atomic (regardless of whether they are volatile or not).

JMM – Some Technical Terms

Reading JSR-133 is not exactly fun and it is not easy to understand. Here are two technical terms being used in JSR-133 whose meanings might not be obvious:

Happens-before: Means that within a single thread, if instruction A happens-before B, then B is guaranteed to see the value of all variables modified by A after the modification.

Synchronizes-with: Means that if instruction A in Thread T1 happens-before B in Thread T2, then B is guaranteed to see the value of all variables modified by A after the modification.

Hence, happens-before and synchronizes-with essentially mean the same thing, except happens-before is for single threads, while synchronizes-with is for multiple threads.

Applying JMM in Your Code

- Check variables accessed by multiple threads. Do they need to be volatile/synchronized? (not always yes)
- Note that reads can be reordered and also need to be synchronized! (or made volatile) Don't think that only writes need to be synchronized.
- Write multi-threaded, high-volume, stressful test cases and run them on machines with multiple CPUs.
- Prefer using volatiles and CAS over synchronization as volatiles and CAS are very fast.

End of Part I. Java Memory Model

You might have some questions at this point, but not to worry, you will be able to find all the answers online.

Part I. Java memory model

Part II. Concurrent utilities

Part II. Concurrent Utilities

Brief overview of what is available under java.util.concurrent.*.

- Lock objects
- Executors
- Concurrent collections
- Atomic variables
- Other stuff

Concurrent Utilities
1. Lock Objects

Package: java.util.concurrent.locks

ReentrantLock

- Almost the same as the synchronized keyword with wait(), notify() and notifyAll(), but gives more flexibility.
- "Reentrant" means that when you have already acquired the lock, you can acquire the same lock again without deadlocking yourself.
- "Reentrant" also means that if two or more threads execute the same piece of code at the same time, there won’t be any conflicts (this definition does not apply here).
- The synchronized keyword and the ReentrantLock uses two completely different sets of code and has nothing in common.
- The synchronized keyword seems to perform better with uncontended locks, but the ReentrantLock seems to perform better with contended locks. (no guarantees here)
- Main use of ReentrantLock – can try locking with timeout.

Concurrent Utilities
1. Lock Objects (2)

ReentrantReadWriteLock

- Encapsulates two different types of locks within one lock – a read lock and a write lock.
- Usually only useful when you have a large number of readers and small number of writers, and user operations outweigh synchronization overhead.
- Usually only useful when you have multiple CPUs.

Concurrent Utilities
2. Executors

- Abstraction of the execution of a large number of tasks immediately or in the future.
- Basically only three types of Executors in the JDK:
  - ThreadPoolExecutor
  - ScheduledThreadPoolExecutor
  - ForkJoinPool (new in Java 7)

Concurrent Utilities
2. Executors (2)

- You might find that ThreadPoolExecutor is the only executor you will ever need to use.
- Option of either fixed or variable sized thread pools.
- I have never found variable-sized thread pools useful, so I only need fixed-sized thread pools (thread creation is a bit expensive).
- Make sure never to use an unbounded thread pool (unlimited number of threads).

Concurrent Utilities
3. Concurrent Collections

ConcurrentHashMap

- Perhaps the most commonly-used concurrent collection that exists.
- Similar to a synchronized map, but allows more concurrency by splitting the table into lock sections.
- Performance is comparable to Cliff Click’s NonBlockingHashMap, which is a lock-free hashmap that outperforms ConcurrentHashMap only when you have a large number of processors.

Concurrent Utilities
3. Concurrent Collections (2)

ArrayBlockingQueue

- Fixed-length array-backed blocking queue.

LinkedBlockingQueue

- Variable-length linked-list-backed blocking queue.
- Allows option to specify max possible queue size so that it does not get filled up indefinitely.

Concurrent Utilities
3. Concurrent Collections (3)

ArrayBlockingQueue vs LinkedBlockingQueue

- Both use very basic wait/notifyAll implementation with no spin retries.
- Advantage of using linked list: 1) saves memory when queue is empty.
- Advantage of using array: 1) less memory allocation and GC, 2) less pointer dereferencing (faster), 3) uses less memory when queue is full.
- Hence most of the time I will choose to use an ArrayBlockingQueue, not a LinkedBlockingQueue.

Concurrent Utilities
3. More Concurrent Collections

- Some more concurrent collections that I have never used before.
- Provided here only as an overview.
- Source code and javadoc readily available online.

ConcurrentLinkedDeque
ConcurrentLinkedQueue
ConcurrentSkipListMap
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet

DelayQueue
ForkJoinPool (new in Java 7)
LinkedBlockingDeque
LinkedTransferQueue
PriorityBlockingQueue
SynchronousQueue

Concurrent Utilities
The LMAX Disruptor Blocking Queue

- The LMAX Disruptor is perhaps the fastest blocking queue implementation in the planet.
- Boasts a speed of 100K events per second (10 nanoseconds per event)
- Main difference from ArrayBlockingQueue is that the Disruptor uses spin retries. Spin retries prevent the threads from doing lock, park, and release immediately when there is no work to do, hence saving CPU cycles.
- Spin retries are slower when your input is slow (does more work).
- Disruptor is faster than the ArrayBlockingQueue only at super high throughputs (100K per second). At slower throughputs, the Disruptor is the same speed or slower than the ArrayBlockingQueue.
- I implemented an AtomicBlockingQueue that does something similar, and is slightly faster than the Disruptor, but difference is hardly noticeable.
- Conclusion: Your application can hardly catch up with these speeds, so using the ArrayBlockingQueue and LinkedBlockingQueue should be your best options!

Concurrent Utilities
4. Atomic Variables

Package:

java.util.concurrent.atomic

Utilities to provide atomic updates to variables without the overhead of synchronization.

Most commonly used:

AtomicBoolean

AtomicInteger

AtomicLong

Those that I have never used:

AtomicIntegerArray

AtomicLongArray

AtomicMarkableReference

AtomicReference

AtomicReferenceArray

AtomicStampedReference

Concurrent Utilities
What is CAS?

- Atomic variables work using a principle/algorithm called Compare-And-Set (CAS).
- CAS is supported at the machine instruction level. Hence it is very fast.

The basic idea is like this:

1. Get the expected value (usually current value)
2. Get the update value (usually expected value + delta)
3. Check that the variable currently has the expected value
4. Set the variable to the update value
5. Check that the variable now has the update value
6. If true, return success. Else retry from 1.

Concurrent Utilities
CAS Algorithm

The getAndIncrement() method in AtomicInteger gives the algorithm for this:

```
public final int getAndIncrement() {
  for (;;) {
    int current = get();
    int next = current + 1;
    if (compareAndSet(current, next)) // calls native code
      return current;
  }
}
```

The native instruction to perform CAS guarantees that the operation is atomic (all or nothing).

Concurrent Utilities
5. Other Stuff

There are a few other small utilities that I think are quite useful:

CountDownLatch

- Allows one or more threads to wait at a certain point until countdown is completed.
- Any thread that does the countdown will not wait at the point of countdown. This is the difference between the CountDownLatch and the CyclicBarrier.

CyclicBarrier

- Allows two or more threads to wait at a certain point until all threads have arrived at the wait point. Then all threads will resume execution at the same time.
- Useful when you want to sync up all threads to arrive at a certain point in the code before proceeding further.

Concurrent Utilities
5. Other Stuff (2)

Phaser (new in Java 7)

- Combines the functionality of both the CountDownLatch and the CyclicBarrier.
- With the new Phaser, you will not need to use any of the CountDownLatch and the CyclicBarrier anymore.

ThreadLocalRandom (new in Java 7)

- Better performance for Random when using multiple threads.
- Implementation is only a ThreadLocal wrapper around java.util.Random. Hence this is very useful in conceptual reuseability.

Concurrent Utilities
5. Other Stuff (3)

Semaphore

- Counter and waiter for availability of a particular resource.

TimeUnit

- Time unit conversions utility

Concurrent Utilities
Some Recommendations

- JDK collections are reasonably good quality so you should use them and not re-invent the wheel.
- Some open-source projects (Trove, Google Collections) are worth looking at but do expect only very slight performance gains.
- When in doubt, synchronize.
- But don’t over-synchronize unnecessarily!

Concurrent Utilities
5. Other Stuff - ThreadLocal

java.lang.ThreadLocal

- Not part of the concurrent package, but closely-related to concurrency.
- Implementation is super-fast using a rehash lookup hashtable (not linked list of entries).
- Saves programmers from needing to extend java.lang.Thread to add additional attributes to each thread.
- Entries for dead threads will be cleaned up by GC automatically.

Concurrent Utilities
5. Other Stuff - ThreadLocal Sample

- Don't add "synchronized" on method "initialValue")
- @Override is optional

```
private static final ThreadLocal<StringBuilder>
  perThreadSB =
 new ThreadLocal<StringBuilder>() {
  @Override protected StringBuilder initialValue() {
  return new StringBuilder();
  }
 };
```

Concurrent Utilities
5. Other Stuff – Sample Code

Here is a simple example of how I often use barriers (using Phaser here). The task is to run N threads concurrently, starting them at the same time, and then stopping automatically after a few seconds.

SimpleBarrierExample.java:

```
import java.util.concurrent.Phaser;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class SimpleBarrierExample {
    public static void main(String[] args) {
     // Config.
     int numThreads = 10;
     int numSeconds = 3;
```

```
// Init threads.
final Phaser p = new Phaser(numThreads + 1);
for (int i = 0; i < numThreads; i++) {
        new Thread() {
                public void run() {
                        p.arriveAndAwaitAdvance();
                        for (;;) doSomething();
                }
        }.start();
}

// Auto-exit after numSeconds seconds.
System.out.println("Wait " + numSeconds + " seconds");
new ScheduledThreadPoolExecutor(1).schedule(new Runnable() {
        public void run() {
                System.out.println("Done");
                System.exit(0);
        }
}, numSeconds, TimeUnit.SECONDS);
```

```
     // Start now.
     p.arriveAndDeregister();
    }

    public static void doSomething() {
     try {
            System.out.println(Thread.currentThread().getId());
            Thread.sleep(500L);
     } catch (InterruptedException e) {
            e.printStackTrace();
     }
    }

}
```

End of Part II. Concurrent Utilities

We are done with a brief overview of what is available out of the box from the java.util.concurrent package.

Part I. Java memory model

Part II. Concurrent utilities

The End

Hope you had enjoyed this presentation, and now back to coding work!!

Abbreviations Used

CPU	Central Processing Unit
JDK	Java Development Kit
JMM	Java Memory Model
JSR	Java Specification Requests
JVM	Java Virtual Machine

References

All material presented here is publicly available but this presentation is a digested version of some selected online materials. There is no reason why you cannot go and dig up the same information given enough time and motivation.

Hence this presentation is designed to save you time.

References - Official JSR-133

JSR-133: Java Memory Model and Thread Specification

(Complete proposal document for JSR-133)

http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf

Java Language Specification – Section 17.4 Memory Model

(The official documentation defining how the JMM works)

http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4

References - Low Level

The Unofficial JSR-133 Guide for Compiler Writers

(Human-readable guide to memory barriers required by JSR-133)

http://gee.cs.oswego.edu/dl/jmm/cookbook.html

CPU Caching

(General background info on how CPU caches work)

http://en.wikipedia.org/wiki/CPU_cache

References - Unofficial JSR-133

FAQ to JSR 133 - The Java Memory Model
(Human-readable introduction to JSR-133)
http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html

Problems with the Old Java Memory Model
(Online article to summarize the old problems)
http://www.ibm.com/developerworks/library/j-jtp02244/index.html

Fixes to the Old Java Memory Model
(Online article to summarize the new fixes)
http://www.ibm.com/developerworks/library/j-jtp03304/

References - Others

The "Double-Checked Locking is Broken" Declaration

(Detailed description of the problems with double-checked locking)

http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

Using Unsafe in Java

(How to use C-pointers-style direct memory access in Java)

http://highlyscalable.wordpress.com/2012/02/02/direct-memory-access-in-java/