IBM Software Group

Concurrency in WTP
aka Multi-Threaded Issues in WTP

2009-02-26

Rational® software

ON DEMAND BUSINESS™

Agenda

- Why is this topic important?
- Correctness (Race conditions)
- Deadlock

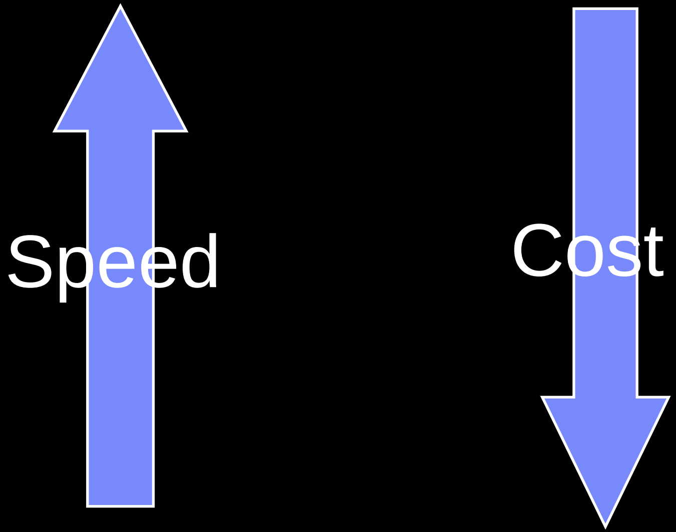
Why do we care about this topic?

- As the size of WTP based products increase and as our users adopt more multi-core machines, WTP based products are hitting concurrency problems more often
  - This leads to poor user satisfaction
  - Higher support costs
  - Higher development costs
- Concurrency problems are:
  - Devastating for our users. Recovery is often, kill the product.
  - Hard to debug
- Two main types of problems
  - Random failures (Race conditions)
  - Deadlocks (these are the easier ones!)

Java Memory Model

- All modern computers have different levels of memory

- Registers
- L1 Cache
- L2 Cache
- L3 Cache
- Main memory



Which of these are Thread safe?

private int j;

private A a;

a) j = 5;

b) j ++;

c) If (a != null)a = new A();

d) None of the above

Which of these are Thread safe?

a) j = 5;

b) j ++;

c) If (a != null)a = new A();

[✓ d) None of the above]

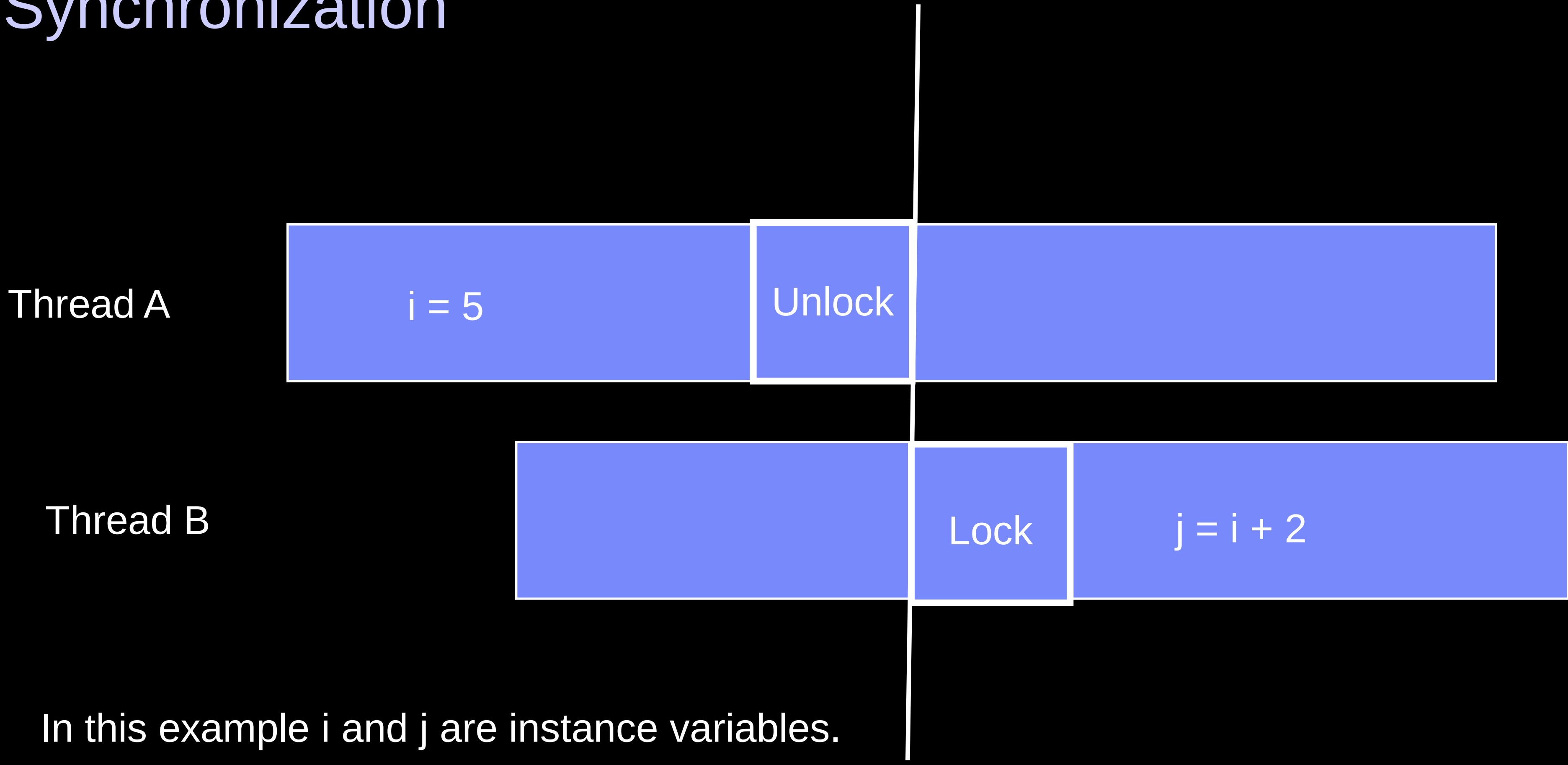
Synchronization

- Synchronization is about more than locking
  - It is about multi-threaded visibility

- - *In the absence of synchronization, the compiler, processor or runtime can do downright weird things, with respect to visibility and order of events.*

- - “*If multiple threads access some mutable state variables without appropriate synchronization, your program is broken.*” – Java Concurrency in Practice

Synchronization



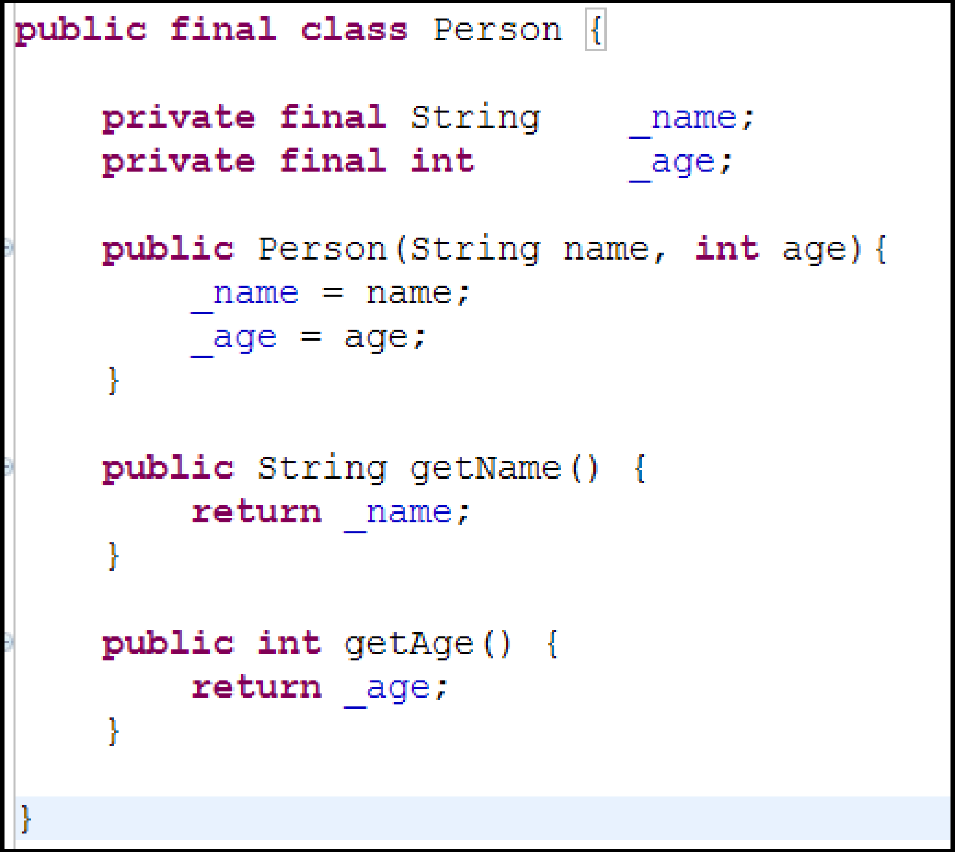
In this example i and j are instance variables.

```
int i;
int j;
```

What to do? Use Immutable Objects

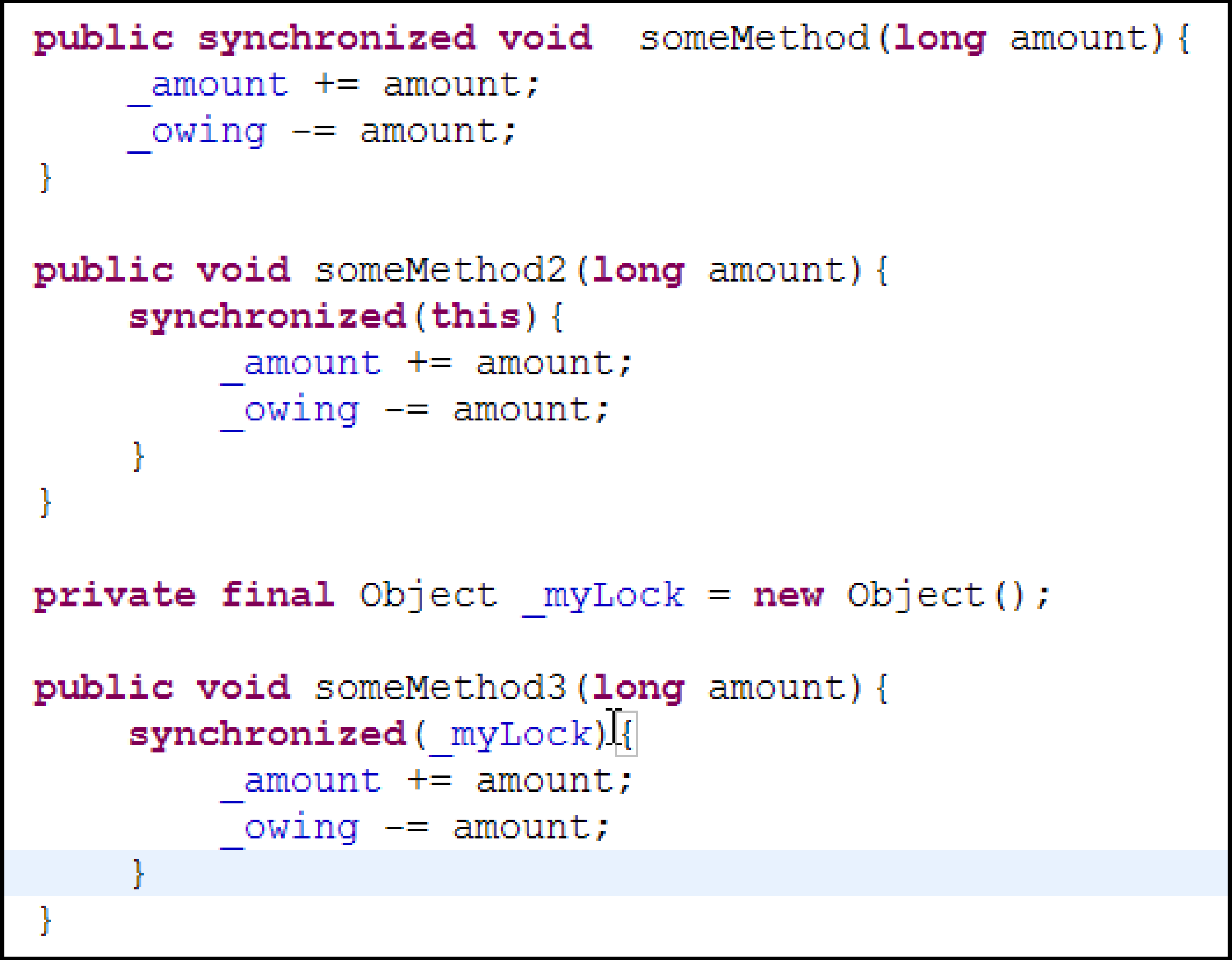
- 1) Immutable objects are your friends
  - Never need to worry about concurrency issues
  - Final is also your friend

This is the number one best technique for solving concurrency problems!



```
public final class Person {

    private final String    _name;
    private final int       _age;

    public Person(String name, int age){
        _name = name;
        _age = age;
    }

    public String getName()  {
        return _name;
    }

    public int getAge()  {
        return _age;
    }

}
```

Use synchronization



```
public synchronized void  someMethod(long amount){
    _amount += amount;
    _owing -= amount;
}

public void someMethod2(long amount){
    synchronized(this){
        _amount += amount;
        _owing -= amount;
    }
}

private final Object _myLock = new Object();

public void someMethod3(long amount){
    synchronized(_myLock){
        _amount += amount;
        _owing -= amount;
    }
}
```

Use Synchronization

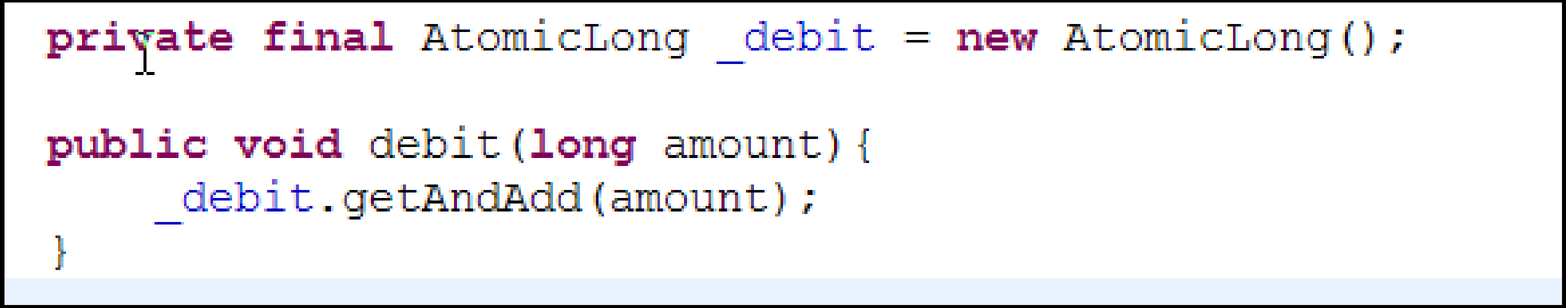
- It is not that expensive
  - While I measured a performance difference between using synchronized code and unsynchronized code, the synchronized code was still very fast. My test was to call the accessor 400 Million times. When synchronized it took 8s (50 M requests/s) and unsynchronized it took .23 seconds (1.7 B requests/second).
  - This was on a quad machine, where 4 threads where calling the accessor. I was trying to generate lots of contention.
- But still, use when appropriate

Use volatiles

- The Java Memory Model
  - Volatile - write before subsequent reads
- Private volatile long _something;

Use Atomics

- Atomics are like better volatiles (because they support some compound operations like increment)
- Good to use when only a single variable is changing



```
private final AtomicLong _debit = new AtomicLong();

public void debit(long amount){
    _debit.getAndAdd(amount);
}
```

Use **java.util.concurrent**

- Java 5 added a number of concurrent classes
- They are very well implemented
- They are supported by the JVM and the JIT
- Use them
  - But only when needed. A concurrent collection (1,700 bytes) is much bigger than a non concurrent collection (100 bytes). Avoid having a lot of small concurrent collections.

What is a Deadlock?

- Aka as a deadly embrace
  - Thread 1 wants a lock on A and B
  - Thread 2 wants a lock on B and A

- Thread 1 has a lock on A and is waiting for B
- Thread 2 has a lock on B and is waiting for A
- These threads will wait forever

- External Symptoms
  - The product appears hung, the UI threads may or may not be responding
  - Little or no CPU activity
  - Only solution is to kill the product

Deadlocks

- Lots of things can trigger locks
  - Synchronized blocks
  - wait()
  - ILock
  - Scheduling Rules
  - Jobs
  - java.util.concurrent.locks
  - Workspace
  - Display
- The JVM only knows about some of these

Deadlocks

- No magic here
- Locks need to be ordered
  - If you always lock A then B then C you can’t hit a deadlock
- Use open calls to alien methods
  - An open call is one that doesn’t hold any locks
- If someone calls you and they are holding a lock (like a resource change listener)
  - Be aware that you now have a lock
  - If that is a problem, consider calling your unsafe code on another thread
- Document your locking

Debugging a Deadlock

- You need to get the stack traces
  - Ctrl-Break on Windows (or SendSignal)
  - kill -QUIT <pid> on Linux
- Look at all the threads that have some “length” to them.
  - Look for threads that are blocked (B) or waiting (CW)
  - Usually the two to three longest threads are the cause of the deadlock

Recap

- Use immutable objects
- Use the appropriate level of synchronization
  - Don’t leave ticking bombs
  - Don’t think just because you can’t reproduce the problem that there is no problem
- Use the new java 5 concurrency support
- Only make Open calls
- Document your locking
- Further Reading
  - “Java Concurrency in Practice” – Brian Goetz