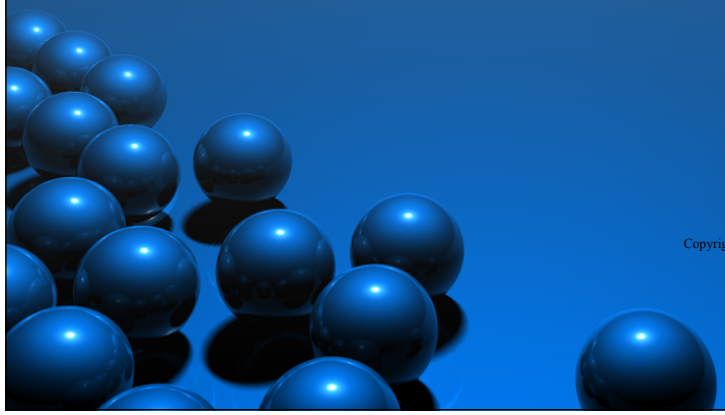
TDD & Concurrency

Brett L. Schuchert, Object Mentor
shoe@objectmentor.com
http://schuchert.wikispaces.com/TddAndConcurrency



Quick Quiz

```
public class ObjectWithValue {
    private int value;
    public void incrementValue() { ++value; }
    public int getValue() { return value; }
}
```

Table Discussion (5 minutes)

- This code is not MT safe
  - Prove this with a test
- Here are some candidate non-functional requirements for the creation of tests:
  - Repeatable
  - Isolated
  - Fast
  - Self-validating
  - Timely
- Which (if any) seem to apply to your approach?

While you were away…

- Let’s review the driver’s results…

Which table would like to present their solution?

Here's what's happening...

- ++value turns into 7 byte-codes:
  - Assume the value field has a value of 42

Byte-code	Description	Operand Stack After
aload 0	Load local variable 0 onto the stack. In a non-static method, local variable 0 is always the "this" pointer, or the object that received the most recent message.	this
dup	Place a copy of the top operand on the operand stack back onto the operand stack.	this, this
getfield value	Retrieve the field named "value" from the object pointed to on the top of the operand stack, which is "this". Put the resulting value back on to the top of the operand stack.	this, 42
iconst_1	Put the constant value 1 onto the operand stack.	this, 42, 1
iadd	Perform integer addition on the top two items on the operand stack and place the result back on the operand stack.	this, 43
putfield value	Put the top item on the stack (43) in the "value" field of the object pointed to by the next to top item on the operand stack (this).	<<empty>>
return	Return back to the place where this method was called.	

How many possible paths of execution?

- Assume the following:
  - Two threads
  - One instance of ObjectWithValue
  - Both calling incrementValue()
- Answer the following:
  - How many possible ways can those two threads execute?
  - What are the possible results?

Answers

- How many possible orderings?
  - 3,432
- How can we calculate it?
  - Assuming no looping/branching
  - All instructions executed through to completion
  - All threads complete
  - N → Number of instructions
  - T → Number of threads
  - $(N * T)! / N!^{T}$
- How many possible outcomes?
  - 2
  - value incremented by 2
  - value incremented by 1
    - How?

Candidate Tests

```
@Test
public void singleThreaded() throws InterruptedException {
    Thread t = new Thread(incrementValue);
    t.start();
    t.join();
    assertEquals(10000, object.getValue());
}

@Test
public void multipleThreads() throws InterruptedException {
    Thread[] threads = new Thread[5];

    for (int i = 0; i < threads.length; ++i)
        threads[i] = new Thread(incrementValue);

    for (Thread t : threads)  t.start();
    for (Thread t : threads)  t.join();

    assertEquals(50000, object.getValue());
}
```


Demo...

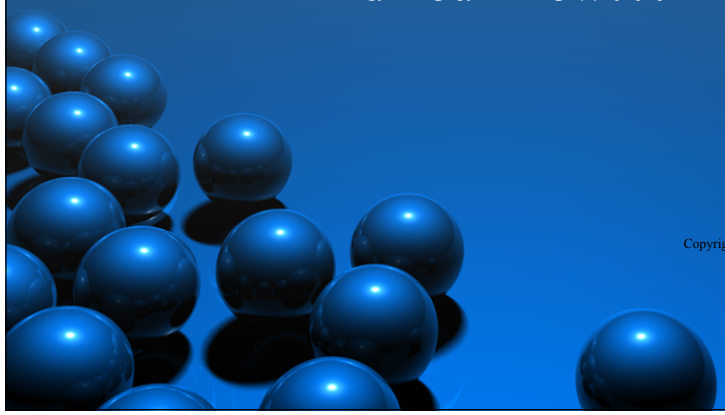
Impressions

Impressions, ConTest

Table-Top Discussion

- Describe a success or failure you've experienced using multiple threads.
- Our list:

Did You Know???



Know the Big Three

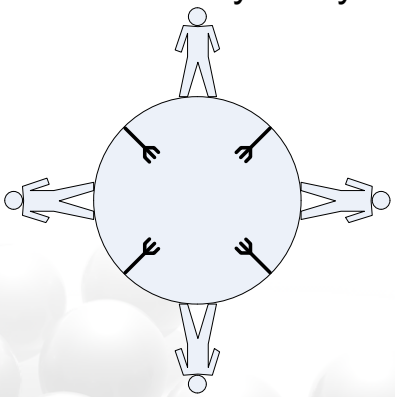
- Producer/Consumer
- Readers/Writers
- Dining Philosophers

Yes, read Wikipedia, go ahead, I'll wait…

No, I won't!

Dining Philosophers

- Imagine 4 hungry philosophers:
  - 2 forks to eat
  - Grab one at a time
  - Won't release until done eating
- What happens when they all try to eat?



Avoiding Deadlock

- Break any of the 4 conditions
  - Makes deadlock impossible
  - Has performance and responsiveness ramifications
- Mutual Exclusion
  - Don't lock mutual resources
  - Don't share
  - Make more available
  - Only lock one at a time
- Lock and Wait
  - Don't wait – back-off if everything not available
- No preemption
  - Demand threads to release
  - DO NOT use Thread.stop() – can leave resources locked
- Circular Wait
  - Order your locks

And now for something completely different?

Testing Multi-threaded servers

- Imagine you have a client-server
  - Clients make requests of the server
  - Server receives request
  - Server processes request
    - Spending over half the time waiting on I/O
  - Server sends response
- Under light load, I don't want one client waiting for another.
- Confirmation:
  - Process a single client request
  - Process 10 client requests in less than 10x
  - (Yes, only the happy path.)

Let’s work on it...

Questions?

You're waiting for another team to load the database

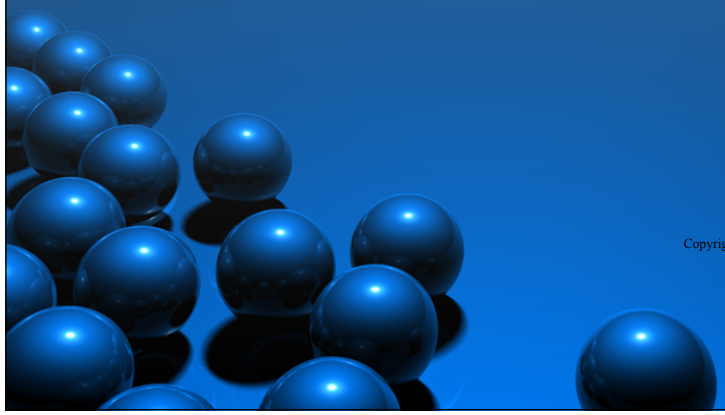
- Problem: Database synchronization takes about 10 hours.

- A Few facts
  - Currently convert roughly 2 records per second
  - As written, running tests requires another team to load records in to an external database.
  - Think wait-time is roughly 300 – 400 ms per record (reading and writing)

- Final Results: 15 threads, 28 minutes…

TDD & Concurrency

Part 4: Keep it away from me!



Scope it way down...

- A guideline to live by…
  - Manage concurrent-based stuff managed in one place
    - Close to where that stuff lives

Dependent State

- Consider the following (incomplete) iterator:

```
public class IntegerIterator
    implements Iterator<Integer>, Iterable<Integer> {
    private Integer nextValue = 0;

    public boolean hasNext() {
        return nextValue < 100000;
    }
    public Integer next() {
        return nextValue++;
    }
    public Integer getNextValue() {
        return nextValue;
    }
}
```

Dependent State+Multi-Threaded

- Use this code in a test:

```
IntegerIterator iterator = new IntegerIterator();

for (Integer value : iterator) {
}

assertEquals(10000, iterator.getNextValue()
```

Dependent-State+Multi-Threaded

- What about this use (including next page)?

```
public class UseIntegerIterator implements Runnable {
    IntegerIterator iterator;

    public UseIntegerIterator(IntegerIterator iterator) {
        this.iterator = iterator;
    }

    @Override
    public void run() {
        while (iterator.hasNext()) {
            iterator.next();
        }
    }
}
```

Dependent State+Multi-Threaded

```
IntegerIterator iterator = new IntegerIterator();
Thread t1 = new Thread(new UseIntegerIterator(iterator));
Thread t2 = new Thread(new UseIntegerIterator(iterator));
t1.start();
t2.start();
t1.join();
t2.join();

assertEqual(10000, iterator.getNextValue()); // ??
```

Dependent State + Multi-Threaded

- How can we fix this?
  - Client-based locking
  - Server-based locking

Client-Based Locking

- The client (user) of the common data locks:

```
public class UseIntegerIteratorClientBasedLocking
    implements Runnable {
    public void run() {
        while (true) {
            synchronized (iterator) {
                if (iterator.hasNext())
                    iterator.next();
                else
                    break;
            }
        }
    }
}
```

Server-Based Locking

- The server guards the dependent calls:

```
package dependent.serverbasedlocking;

public class IntegerIteratorServerLocked {
    private Integer nextValue = 0;

    public synchronized Integer getNextOrNull() {
        if (nextValue < 100000)
            return nextValue++;
        else
            return null;
    }
    public Integer getNextValue() {
        return nextValue;
    }
}
```

Server-Based Locking

- Here's an updated client that now works:

```
public void run() {
    while (true) {
        Integer next = iterator.getNextOrNull();
        if (next == null)
            break;
    }
}
```

- Evaluate, between client-based and server-based locking
  - Which do you prefer?
  - Which solution most reduces the scope of the shared data?

Client or Server-Based?

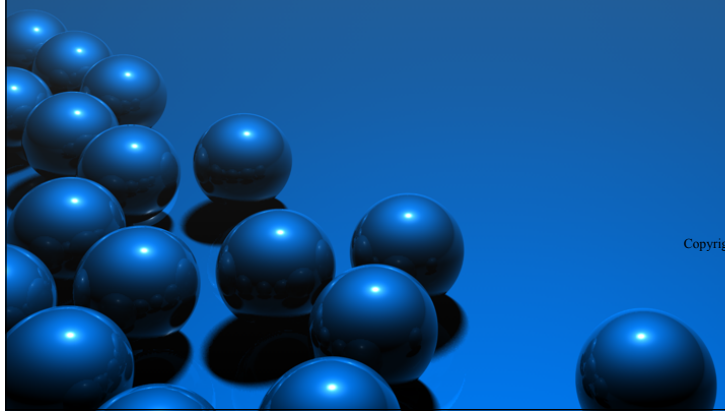
- When you have control of the code: Prefer server-based locking
  - It reduces the possibility of error
  - It reduces repeated code
  - It will change your API (your client has change anyway)
  - It enforces a single policy
  - It reduces the scope of a multi-thread mutable variable
  - It makes tracking down errors far easier

Client or Server-Based Locking

- What if you don't have control?
  - Adapter to change the API & add Locking
  - The Java collections use the collection object itself. Even so:
    - Adapt
    - OR better yet, use the thread-safe collections with extended interfaces
  - Pick client-based locking only if it is impossible to use an adapter – which should be nearly never

TDD & Concurrency

Part 5: A few recommendations



Hints

- Run with more threads than processors
- Run on the target platform when tuning
- Run with the –server VM argument
- Rely on published algorithms when possible
  - Producer/consumer
  - Readers/writers
  - Dining Philosophers
- Don't build your own thread-safe containers
  - Use java.concurrent
  - JDK 1.4, use http://g.oswego.edu/dl/classes/collections/
  - http://blogs.azulsystems.com/cliff/2007/03/a_nonblocking_h.html
- Consider using locks if in Java 5 or stick with intrinsic locks if moving to 6.
- There's a lot of gold in java.lang.concurrent!

Atomic Variables

- Modern Processors have a single instruction
  - Compare And Swap (CAS)
  - Logical example of optimistic locking
    - Increment value

```
int v;
do {
    v = get current value
} while(v != compareAndSwap(v, v+1))
```

  - Keep attempting to increment value until it actually happens
  - If processor does not support CAS operation, Java compiler simulates it.
    - If this is the case, high-contention systems will burn CPU time

One Final Request

TDD & Concurrency

Brett L. Schuchert, Object Mentor
shoe@objectmentor.com
http://schuchert.wikispaces.com/TddAndConcurrency