Concurrency Culprit
and
Plain 'Ole Java Concurrency

SWEN-342

The Ultimate Culprit - Shared, Mutable State

- Most of your development has been in imperative languages.
- The fundamental operation is assignment to change state.
  - Assignable variables are mutable.
  - May be exposed as public (bad karma).
  - May be exposed via interface methods (medium warm karma).
  - Things get tricky very fast when > 1 thread can invoke a mutating function.
- Three approaches:
  - Make things immutable.
  - Hide shared state behind sequential access.
  - Provide mechanisms to support controlled access to shared, mutable state.

Immutability

- All state in the Class is final.
- Only assignment is in the constructor.
- Mutators now return a new object.
- Examples:
  - Points in space (x, y, z)
  - Immutable collections
- Performance not as bad as it sounds:
  - Compiler optimizations have improved significantly.
  - Tail recursion lessens the problems of stack explosion.
  - Does require a new way of thinking (Scala, LISP, Clojure)

Immutability

```java
// NOTE: Not thread safe!

public class Point {
    private int x ;
    private int y ;

    public Point(int x, int y) {
        this.x = x ;
        this.y = y ;
    }

    public void move(int dx, int dy) {
        x += dx ;
        y += dy ;
    }

    . . .
}
```

```java
// NOTE: Thread safe

public class Point {
    private final int x ;
    private final int y ;

    public Point(int x, int y) {
        this.x = x ;
        this.y = y ;
    }

    public Point move(int dx, int dy) {
        return new Point(x + dx,
                         y + dy) ;
    }

    . . .
}
```

This is thread safe, but can it be used the same way?

Hide Shared State

- Do not allow direct calls on methods.
- Send messages instead – serialize access.
- State encapsulated in a thread (agent).
  - Process can extract messages w/o interference.
  - Process can (possibly) serve things out of order.
- Note: Much simpler to scale to multiple processors w/o shared memory.
- We'll see this in the second part of the course with Agents.
- Note: Can be combined with immutability approaches
  - Scala
  - Erlang

Shared, Mutable State

- Need someway to
  - Enforce sequential guarantees in face of concurrency.
  - Prevent race conditions.
  - Address safety, liveness, fairness concerns.
- We'll start with the barebones, standard Java language ***mechanisms*** offered in the original version (~1995).
- We'll then branch out into other libraries that build on this base: java.util.concurrent (Java 5, ~2004)

To Get Things Going - What's Wrong Here?

```
@NotThreadSafe
public class UnsafeSequence {
  private int next = 0 ;

  public int getNext( ) {
    return next++ ;
  }
}
```

Is this an "atomic" operation?

This is an example of what type of a race condition?

Fixing The Example

```
@ThreadSafe
public class SafeSequence {
  @GuardedBy("this") private int next = 0 ;

  public synchronized int getNext( ) {
    return next++ ;
  }
}
```

- Cache's flushed on entry to / exit from getNext( )
- One thread at a time can execute getNext( )

What If Client Wants Two Sequential Numbers?

```
@ThreadSafe
public class SafeSequence {
  @GuardedBy("this") private int next = 0 ;

  public synchronized int getNext( ) {
    return next++ ;
  }
}
. . .
SafeSequence s = new SafeSequence( ) ;
. . .
/* Client(s) */
int i, j ;
i = s.getNext() ;  j = s.getNext() ;
assert(  j == i + 1 )    //??
```

How can this break?

What If Client Wants Two Sequential Numbers?

```
@ThreadSafe
public class SafeSequence {
  @GuardedBy("this") private int next = 0 ;

  public synchronized int getNext( ) {
    return next++ ;
  }
}
. . .
SafeSequence s = new SafeSequence( ) ;
. . .
/* Clients */
int i, j ;
synchronized ( s ) {
     i = s.getNext() ;  j = s.getNext() ;
}
assert(  j == i + 1 )    //??
```

This works, but why does it have a bad code smell?

What If Client Wants Two Sequential Numbers?

```
@ThreadSafe
public class SafeSequence {
  @GuardedBy("this") private int next = 0 ;

  public synchronized int getNext( ) {
    return next++ ;
  }
  public synchronized void getVector( int vector[ ] ) {
    for (int i = 0 ; i < vector.length ; ++i ) {
        vector[i] = getNext( ) ;
    }
}
. . .
SafeSequence s = new SafeSequence( ) ;
. . .
/* Clients */
int v[2] ;
s.getVector(v) ;
```

Why do we need to switch to return a vector?

What happens when a thread holding a lock tries to obtain that lock again?

What If Client Wants Two Sequential Numbers?

```
@ThreadSafe
public class SafeSequence {
  @GuardedBy("this") private int next = 0 ;

  public synchronized int getNext( ) {
    return next++ ;
  }
  public synchronized void getVector( int vector[ ] ) {
    for (int i = 0 ; i < vector.length ; ++i ) {
        vector[i] = getNext( ) ;
}
. . .
SafeSequence s = new SafeSequence( ) ;
. . .
/* Clients */
int v[2] ;
s.getVector(v) ;
```

Assumes the lock is reentrant

Plain Ole' Java Concurrency (POJC)

- Passive objects (resource managers)
- Object locks
- Active objects
  - Threads
  - Runnable
  - th.start -> th.run() or rn.run()
  - Thread.currentThread()
  - th.getName(), th.join()
- Synchronized methods and blocks
- Wait / notify / notifyAll
- The nastiness of exceptions
- YUCCH!

Thread Safe Objects

- A thread-safe class behaves correctly
  - When accessed by multiple threads
  - Regardless of scheduling or interleaving
  - With no additional synchronization on the part of the caller
- Thread-safe classes encapsulate necessary synchronization so clients need not provide their own.
- Based on good OO design principles:
  - Encapsulate state in private instance variables
  - Use immutability where practicable
  - Specify state invariants that must be maintained
- Added:
  - Locks to maintain invariants in the face of concurrent access

Thread Safe Object Consequences

- Stateless objects are automatically thread safe.
- Immutable objects are automatically thread safe.
- Effectively immutable objects are automatically thread safe
  - Built from mutable parts.
  - Never change those parts after construction.
  - Never let a mutable part "escape" from encapsulation.
    - Getters
    - Parameters
- In all other cases, we have to ensure thread-safety by proper synchronization of access to mutable state.

Synchronization

- Every object has a built-in lock associated with it.
- The lock is acquired via the synchronized keyword.
- The lock is released at the end of the **synchronized code block**.

```
public class Point {
   private int x ;
   private int y ;

   public Point(int x, int y) {
      this.x = x ;
      this.y = y ;
   }

   public void move(int dx, int dy) {
      synchronized(this) {
         x += dx ;
         y += dy ;
      }
   }
   . . .
}
```

Synchronization

- Every object has a built-in lock associated with it.
- The lock is acquired via the synchronized keyword.
- The lock is released at the end of the **synchronized method**.

```
public class Point {
   private int x ;
   private int y ;

   public Point(int x, int y) {
      this.x = x ;
      this.y = y ;
   }

   public synchronized void move(int dx, int dy) {
      x += dx ;
      y += dy ;
   }
   . . .
}
```

Synchronization

- Every object has a built-in lock associated with it.
- The lock is acquired via the synchronized keyword.
- The lock is released at the end of the synchronized code block.

```
public class Point {
    private int x ;
    private int y ;

    public Point(int x, int y) {
        this.x = x ;
        this.y = y ;
    }

    public synchronized void move(Point delta) {
        x += delta.getX();
        y += delta.getY();
    }
    . . .
}
```

We can move to a Point but this can break. How?

What do we need to do to fix the problem?

Synchronization

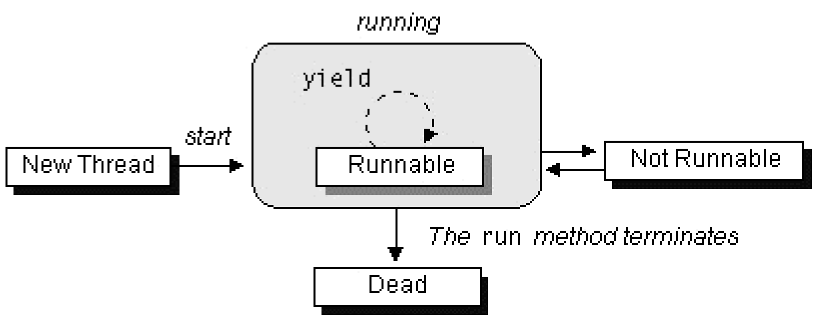
- Every object has a built-in lock associated with it.
- The lock is acquired via the synchronized keyword.
- The lock is released at the end of the synchronized code block.

```
public class Point {
    private int x ;
    private int y ;

    public Point(int x, int y) {
        this.x = x ;
        this.y = y ;
    }

    public synchronized void move(Point delta) {
        synchronized(delta) {
            x += delta.getX();
            y += delta.getY();
        }
    }
    . . .
}
```

Fixed that problem but introduced a new one. What is it?

Thread States

- **Ready**
- **Running**
- **Not Runnable - Waiting, Sleeping, Suspended, Blocked**
- **Dead**



When you invoke start(), a new thread is marked ***ready*** and is placed in the Thread queue.

A thread is placed in the ***waiting*** state, or becomes Not Runnable, when one of these events occurs:

- Its sleep method is invoked.
- The thread calls the wait method to wait for a specific condition to be satisfied.
- The thread is blocking on I/O.

When the run() method terminates, the Thread **dies**. A dead Thread cannot be restarted.

Thread State Transitions

A thread becomes Runnable when one of these events occurs:

• After the initial call to the Thread's `start` method.

• If a thread had been put to sleep, and then the specified number of milliseconds have elapsed.

• If a thread is waiting for a condition, then another object has notified the waiting thread of a change in condition by calling the `notify` or `notifyAll` methods

• If a thread was blocked on I/O, then the I/O has completed.

A thread becomes Not Runnable when one of these events occurs:

• Its sleep method is invoked.

• The thread calls the `wait` method to wait for a specific condition to be satisfied.

• The thread is blocking on I/O.

A thread dies when:

• Its run method completes.

•Threads typically arrange for their own death by executing the `run` method with some loop condition.

• A dead thread cannot be restarted.

wait(), notify(), notifyAll()

wait() - waits for a condition to occur. This is a method of the Object class and must be called from within a synchronized method or block.

When wait is called:

- the current thread is suspended or placed in the wait queue (non-runnable state)
- the synchronization lock for the target object is released, but all other locks held by the thread are retained.
- Note that wait() can also be called with a timeout

notify() - notifies a thread waiting for a condition that the condition has occurred. This is a method of the Object class and must be called from within a synchronized method or block.

When notify() is called:

- an arbitrary thread waiting for the condition attempts to regain the synchronization lock it relinquished as a result of its wait() call.
- After obtaining the lock it resumes execution at the point of its wait()

notifyAll() - works the same as notify except that the steps occur for **ALL** threads waiting in the wait queue for the target object.

(Concurrent Programming in Java - Doug Lea)

State Dependent Behavior

- Assume we have a simple bounded counter.
- Value must range from 0 to some maximum.
- Mutators: up and down

```
public class SBC {
   private int c = 0 ;
   private final int max ;

   public SBC(int max) {
      this.max = max ;
   }

   public int get() {
      return c ;
   }

   public void up() {
      if ( c == max ) {
         ???
      }
      c++ ;
   }

   public void down() {
      if ( c == 0 ) {
         ???
      }
      c-- ;
   }
}
```

What behavior should we have for the ???s?

What is the invariant for this class?

State Dependent Behavior

- Handling end cases: Sequential code
  - Nothing will ever “fix” the problem.
  - Need to signal error
  - Throw an exception
  - Return an error value
- Handling end cases: Concurrent code
  - End case may be temporary
  - If at max, another thread may do a down and we can proceed
  - Therefore, we have an additional option - wait

State Dependent Behavior

```
public class SBC {
   private int c = 0 ;
   private final int max ;

   public SBC(int max) {
      this.max = max ;
   }

   public synchronized int get() {
      return c ;
   }
```

Why did this change from an if statement to a while loop?

```
public synchronized void up() {
   try {
      while( c == max )
         wait() ;
   } catch(Exception e) {} ;

   c++ ;
   notifyAll() ;
}

public synchronized void down() {
   try {
      while( c == 0 )
         wait() ;
   } catch(Exception e) {}

   c-- ;
   notifyAll() ;
}
```

If you care about safety why does this code stink?

What could you do to remove the smell?

State Dependent Behavior

```
public class SBC {
   private int c = 0 ;
   private final int max ;

   public SBC(int max) {
      this.max = max ;
   }

   public synchronized int get() {
      return c ;
   }

           private void waitAtMax {
              try {
                 while( c == max )
                    wait() ;
              } catch (Exception e) {} ;
           }

           private void waitAtMin() {
              . . .
           }
```

```
public synchronized void up() {
   waitAtMax();

   c++ ;
   notifyAll() ;
}

public synchronized void down() {
   waitAtMin() ;

   c-- ;
   notifyAll() ;
}
```

Can you simplify this further?
Would you want to?