# Using Java 8 Lambdas And Stampedlock To Manage Thread Safety

Dr Heinz M. Kabutz

heinz@javaspecialists.eu

Last updated 2017-05-09

Javaspecialists.eu
java training

# Why Should You Care About StampedLock?

- **Optimistic reads of multiple fields**
  - **e.g. Collection containing size and array or elements**
  - **Invariants across multiple fields cannot be guarded with atomics**
    - **Except if you create value holder objects**
- **Much faster than ReentrantReadWriteLock**
- **Can upgrade read to write lock**
- **Makes you look smarter than your colleagues**

# Pessimistic Exclusive Lock (write)

```
public class StampedLock {
  long writeLock() // never returns 0, might block

  // returns new write stamp if successful; otherwise 0
  long tryConvertToWriteLock(long stamp)

  void unlockWrite(long stamp) // needs write stamp


// and a bunch of other methods left out for brevity
```

# Pessimistic Non-Exclusive Lock (read)

```
public class StampedLock { // continued ...
  long readLock() // never returns 0, might block

  // returns new read stamp if successful; otherwise 0
  long tryConvertToReadLock(long stamp)

  void unlockRead(long stamp) // needs read stamp

  void unlock(long stamp) // unlocks read or write
```

# Optimistic Non-Exclusive Read (No Lock)

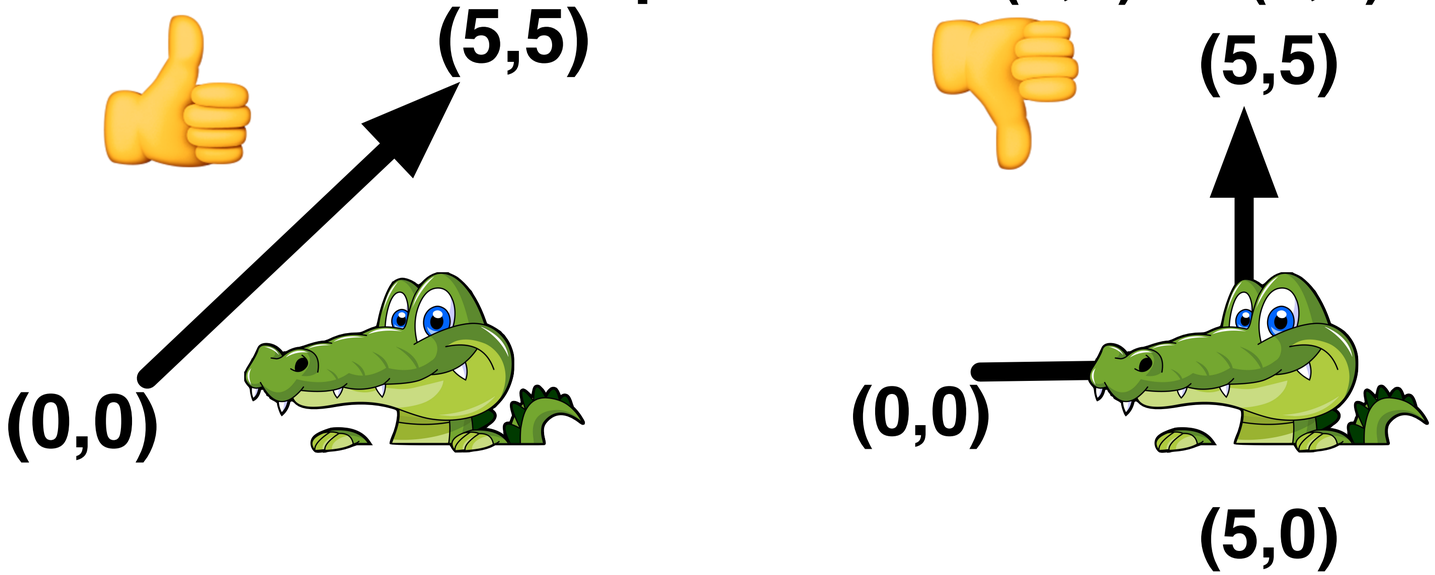
```
public class StampedLock { // continued ...
  // could return 0 if a write stamp has been issued
  long tryOptimisticRead()

  // return true if stamp was non-zero and no write
  // lock has been requested by another thread since
  // the call to tryOptimisticRead()
  boolean validate(long stamp)
```

# Sifis the Crocodile (RIP)

# Introducing the Position Class

- **When moving from (0,0) to (5,5), we want to go in a diagonal line**
  - **We don't want to ever see our position at (0,5) or (5,0)**



# Moving Our Position

- **Similar to ReentrantLock code**

```
public class Position {
  private double x, y;
  private final StampedLock sl = new StampedLock();

  // method is modifying x and y, needs exclusive lock
  public void move(double deltaX, double deltaY) {
    long stamp = sl.writeLock();
    try {
      x += deltaX;
      y += deltaY;
    } finally {
      sl.unlockWrite(stamp);
    }
  }
```

# StampedLock: Optimistic Read

- **We want to avoid becoming crocodile food**
- **We first try an optimistic read**
  - **If it works, great!**
  - **If not, we do a pessimistic, non-exclusive read**

# Code Idiom for Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

# Code Idiom for Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

We get a stamp to use for the optimistic read

# Code Idiom for Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

We read field values into local fields

# Code Idiom for Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

Next we validate that no write locks have been issued in the meanwhile

# Code Idiom for Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

If they have, then we don't know if our state is clean

Thus we acquire a pessimistic read lock and read the state into local fields

# Code Idiom for Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(currentState1, currentState2);
}
```

# Optimistic Read in our Position class

```
public double distanceFromOrigin() {
  long stamp = sl.tryOptimisticRead();
  double currentX = x, currentY = y;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentX = x;
      currentY = y;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return Math.hypot(currentX, currentY);
}
```

The shorter the code path from tryOptimisticRead() to validate(), the better the chances of success

# Value Holder Objects and Atomics

- **We could also hold the x and y pair in a double[]**
- **Then we can use any one of these techniques:**
  - **AtomicReference<double[]>**
  - **AtomicReferenceFieldUpdater**
  - **Unsafe**
  - **VarHandle (Java 9)**

# Using AtomicReference

- **do-while until we finally manage to move**

```
public class PositionAtomicNonBlocking {
  private final AtomicReference<double[]> xy =
      new AtomicReference<>(new double[2]);

  public void move(double deltaX, double deltaY) {
   double[] current, next = new double[2];
    do {
      current = xy.get();
      next[0] = current[0] + deltaX;
      next[1] = current[1] + deltaY;
    } while(!xy.compareAndSet(current, next));
  }
```

# Using AtomicReferenceFieldUpdater

- **Similar to AtomicReference, but one less indirection**

```
public class PositionFU {
  private volatile double[] xy = {0, 0};
  private final static AtomicReferenceFieldUpdater
          <PositionFU, double[]> XY =
    AtomicReferenceFieldUpdater.newUpdater(PositionFU.class,
           double[].class, "xy");
  public void move(double deltaX, double deltaY) {
    double[] current, next = new double[2];
    do {
      current = xy;
      next[0] = current[0] + deltaX; next[1] = current[1] + deltaY;
    } while (!XY.compareAndSet(this, current, next));
  }
```

# CompareAndSwap with sun.misc.Unsafe

- **First we find the memory location offset of the field “xy”**

```
public class PositionUnsafeNonBlocking {
  private final static Unsafe UNSAFE = Unsafe.getUnsafe();
  private static final long XY_OFFSET;
  static {
    try {
      XY_OFFSET = UNSAFE.objectFieldOffset(
          PositionUnsafeNonBlocking.class.
            getDeclaredField("xy"));
    } catch (ReflectiveOperationException e) {
      throw new Error(e);
    }
  }
  private volatile double[] xy = new double[2];
```

# CompareAndSwap with sun.misc.Unsafe

- **Our move() method is similar to AtomicReference**

```
public void move(double deltaX, double deltaY) {
  double[] current, next = new double[2];
  do {
    current = xy;
    next[0] = current[0] + deltaX;
    next[1] = current[1] + deltaY;
  } while (!UNSAFE.compareAndSwapObject(
      this, XY_OFFSET, current, next));
}
```

# So When To Use Unsafe?

- **Simple answer: never**
- **Reputation of “running close to bare metal”**
  - **But just like “Quick Sort”, it can be slower than alternatives**
  - **And it is, um, *u n s a f e !!!***
- **AtomicFieldUpdaters have increased in performance**
  - **http://shipilev.net/blog/2015/faster-atomic-fu/**
- **VarHandles in Java 9**

# Say “Hi” - Or Join My Newsletter

http://tinyurl.com/devoxxuk17

Javaspecialists.eu
java training

# VarHandles Instead of Unsafe

- **VarHandles remove biggest temptation to use Unsafe**
  - **http://gee.cs.oswego.edu/dl/html/j9mm.html**
- **Same speed as Unsafe**
- **Additional cool features, such as:**
  - **get()/set() - plain field access**
  - **getOpaque() / setOpaque()**
  - **getAcquire() / setRelease()**
  - **getVolatile() / setVolatile()**

# VarHandles Instead of Unsafe

Note: Exact API might still change

- **First step is to set up the VarHandle**

```
public class PositionVarHandlesNonBlocking {
  private static final VarHandle XY;

  static {
    try {
      XY = MethodHandles.lookup().findVarHandle(
        PositionVarHandlesNonBlocking.class,
        "xy", double[].class);
    } catch (ReflectiveOperationException e) {
      throw new Error(e);
    }
  }
```

# CompareAndSet with VarHandle

- **Our move() method almost identical to “Unsafe”**

```
public void move(double deltaX, double deltaY) {
  double[] current, next = new double[2];
  do {
    current = xy;
    next[0] = current[0] + deltaX;
    next[1] = current[1] + deltaY;
  } while (!XY.compareAndSet(this, current, next));
}
```

# compareAndExchangeVolatile() with VarHandle

- **Instead of reading the volatile field, get it from CAS**

```
public void move(double deltaX, double deltaY) {
  double[] current, swapResult = xy, next = new double[2];
  do {
    current = swapResult;
    next[0] = current[0] + deltaX;
    next[1] = current[1] + deltaY;
  }
  while ((swapResult =
    (double[]) XY.compareAndExchangeVolatile(
      this, current, next)) != current);
}
```

# Distance Calculation with AtomicReference

- **Extremely easy and very fast**

```
public double distanceFromOrigin() {
  double[] current = xy.get();
  return Math.hypot(current[0], current[1]);
}
```

# Distance with FieldUpdater/Unsafe/VarHandle

- **Even easier**

```
public double distanceFromOrigin() {
  double[] current = xy;
  return Math.hypot(current[0], current[1]);
}
```

# Conditional Change Idiom with StampedLock

```
public boolean moveIfAt(double oldX, double oldY,
                        double newX, double newY) {
  long stamp = sl.readLock();
  try {
    while (x == oldX && y == oldY) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        x = newX; y = newY;
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } finally { sl.unlock(stamp); }
}
```

Unlike ReentrantReadWriteLock, this will not deadlock

# Previous Idiom is Only of Academic Interest

- **This is easier to understand, and faster!**

```
public boolean moveIfAt(double oldX, double oldY,
                        double newX, double newY) {
  long stamp = sl.writeLock();
  try {
    if (x == oldX && y == oldY) {
        x = newX;
        y = newY;
        return true;
    }
  } finally {
    sl.unlock(stamp);
  }
  return false;
}
```

# Conditional Move with VarHandle

- **Multi-threaded is *much* faster than StampedLock**

```
public void moveIfAt(double oldX, double oldY,
                     double newX, double newY) {
  double[] current = xy;
  if (current[0] == oldX && current[1] == oldY) {
    double[] next = {newX, newY};
    do {
     if (XY.compareAndSet(this, current, next))
        return;
      current = xy;
    } while (current[0] == oldX && current[1] == oldY);
  }
}
```

But is it correct? Good question! Difficult to test.

# StampedLock Idioms are Difficult to Master

- **Instead, we can define static helper methods**
  - **Gang-of-Four Facade Pattern**
- **Lambdas make helper methods pluggable**

# Moving with StampedLockIdioms

- **The old move() method**

```
public void move(double deltaX, double deltaY) {
  long stamp = sl.writeLock();
  try {
    x += deltaX;
    y += deltaY;
  } finally {
    sl.unlockWrite(stamp);
  }
}
```

- **Now looks like this**

```
public void move(double deltaX, double deltaY) {
  StampedLockIdioms.writeLock(sl, () -> {
    x += deltaX;
    y += deltaY;
  });
}
```

# Our StampedLockIdioms

- **We simply call writeJob.run() inside the locked section**

```
public class StampedLockIdioms {
  public static void writeLock(StampedLock sl,
                               Runnable writeJob) {
    long stamp = sl.writeLock();
    try {
      writeJob.run();
    } finally {
      sl.unlockWrite(stamp);
    }
  }
  // ...
```

# Optimistic Read using StampedLockIdioms

- **Our old distanceFromOrigin**

```
public double distanceFromOrigin() {
  long stamp = sl.tryOptimisticRead();
  double currentX = x, currentY = y;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentX = x;
      currentY = y;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return Math.hypot(currentX, currentY);
}
```

# Optimistic Read using StampedLockIdioms

- **Becomes this new mechanism**

```
public double distanceFromOrigin() {
  double[] current = new double[2];
  return StampedLockIdioms.optimisticRead(sl,
      () -> {
        current[0] = x;
        current[1] = y;
      },
      () -> Math.hypot(current[0], current[1]));
}
```

# Our StampedLockIdioms.optimisticRead() Method

- **The reading.run() call would probably be inlined**

```
public static <T> T optimisticRead(
        StampedLock sl, Runnable reading, Supplier<T> computation) {
  long stamp = sl.tryOptimisticRead();
  reading.run();
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      reading.run();
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return computation.get();
}
```

# Conditional Change using StampedLockIdioms

- **Our old moveIfAt()**

```
public boolean moveIfAt(double oldX, double oldY,
                        double newX, double newY) {
  long stamp = sl.readLock();
  try {
    while (x == oldX && y == oldY) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        x = newX; y = newY;
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } finally { sl.unlock(stamp); }
}
```

# Optimistic Read using StampedLockIdioms

- **Becomes this new mechanism**

```
public boolean moveIfAt(double oldX, double oldY,
                        double newX, double newY) {
  return StampedLockIdioms.conditionalWrite(
      sl,
      () -> x == oldX && y == oldY,
      () -> {
        x = newX;
        y = newY;
      }
  );
```

# Our StampedLockIdioms.conditionalWrite()

```
public static boolean conditionalWrite(
    StampedLock sl, BooleanSupplier condition,
    Runnable action) {
  long stamp = sl.readLock();
  try {
    while (condition.getAsBoolean()) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        action.run();
        stamp = writeStamp;
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } finally { sl.unlock(stamp); }
}
```

# Using AtomicReference with Lambdas

- **The old move() method**

```
public void move(double deltaX, double deltaY) {
  double[] current, next = new double[2];
  do {
    current = xy.get();
    next[0] = current[0] + deltaX;
    next[1] = current[1] + deltaY;
  } while (!xy.compareAndSet(current, next));
}
```

- **Now looks like this**

```
public void move(double deltaX, double deltaY) {
  xy.accumulateAndGet(new double[2], (current, next) ->  {
    next[0] = current[0] + deltaX;
    next[1] = current[1] + deltaY;
    return next;
  });
}
```

# Conclusion

- **Lambdas help to correctly use concurrency idioms**
  - **Example in JDK is AtomicReference.accumulateAndGet()**
  - **Might increase object creation rate**
    - **Although escape analysis can reduce this**
- **Performance of new Java 9 VarHandles like Unsafe**
  - **Most of Java 9 classes now use VarHandle instead of Unsafe**

# Say “Hi” - Or Join My Newsletter

http://tinyurl.com/devoxxuk17

Javaspecialists.eu
java training