Principles of Software Construction:
Concurrency, Pt. 3 – `java.util.concurrent`

Josh Bloch Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 5b due Tuesday 11:59 p.m.
  - Turn in your work by Wednesday 9 a.m. to be considered as a Best Framework

Key concepts from Tuesday…

- Never use wait outside of a while loop!
  - Think twice before using it at all
- Neither an under- nor an over-synchronizer be
  - Under-synchronization causes safety (& liveness) failures
  - Over-synchronization causes liveness (& safety) failures

Do as little as possible in synchronized regions

- **Get in, get done, and get out**
  - Obtain lock(s)
  - Examine shared data
  - Transform as necessary
  - Drop lock
- **If you must do something slow, move it outside synchronized region**
  - But synchronize before publishing result

Avoiding Deadlock

- Definition: when threads wait for each other and none make any progress
- More formally, a cycle in the waits-for graph
- Classic example
  - T1 locks A, then B
  - T2 locks B, then A
- To avoid deadlocks:
  - Have each thread obtain locks in same order

`java.util.concurrent` is BIG (1)

I. Atomic vars - `java.util.concurrent.atomic`
   - Support various atomic read-modify-write ops
II. Locks - `java.util.concurrent.locks`
   - Read-write locks, conditions, etc.
III. Synchronizers
   - Semaphores, cyclic barriers, countdown latches, etc.
IV. Concurrent collections
   - Shared maps, sets, lists

`java.util.concurrent` is BIG (2)

V. Data Exchange Collections
   – Blocking queues, deques, etc.

VI. Executor framework
   – Tasks, futures, thread pools, completion service, etc.

VII. Pre-packaged functionality - `java.util.arrays`
   – Parallel sort, parallel prefix

I. Overview of java.util.atomic

- **Atomic{Boolean,Integer,Long}**
  – Boxed primitives that can be updated atomically
- **AtomicReference<V>**
  – Object reference that can be updated atomically
  – Cool pattern for state machine AtomicReference<StateEnum>
- **Atomic{Integer,Long,Reference}Array**
  – Array whose elements may be updated atomically
- **Atomic{Integer,Long,Reference}FieldUpdater**
  – Reflection-based utility enabling atomic updates to volatile fields
- **LongAdder, DoubleAdder**
  – Highly concurrent sums
- **LongAccumulator, DoubleAccumulator**
  – Generalization of adder to arbitrary functions (max, min, etc.)

AtomicInteger example (review)

```
public class SerialNumber {
    private static AtomicLong nextSerialNumber = new AtomicLong();

    public static long generateSerialNumber() {
        return nextSerialNumber.getAndIncrement();
    }
}
```

II. Overview of `j.u.c.locks` (1)

- `ReentrantReadWriteLock`
  - Shared/Exclusive mode locks with tons of options
    - Fairness policy
    - Lock downgrading
    - Interruption of lock acquisition
    - Condition support
    - Instrumentation
- `ReentrantLock`
  - Like Java's intrinsic locks
  - But with more bells and whistles

Overview of `j.u.c.locks` (2)

- `Condition`
  - `wait/notify/notifyAll` with multiple wait sets per object
- `AbstractQueuedSynchronizer`
  - Skeletal implementation of locks relying on FIFO wait queue
- `AbstractOwnableSynchronizer`, `AbstractQueuedLongSynchronizer`
  - More skeletal implementations

ReentrantReadWriteLock example

Does this look vaguely familiar?

```
private final ReentrantReadWriteLock rwl =
        ReentrantReadWriteLock();

lock.readLock().lock();
try {
    // Do stuff that requires read (shared) lock
} finally {
    lock.readLock().unlock();
}

lock.writeLock().lock();
try {
    // Do stuff that requires write (exclusive) lock
} finally {
    lock.writeLock().unlock();
}
```

III. Overview of synchronizers

- `CountDownLatch`
  - One or more threads to wait for others to count down
- `CyclicBarrier`
  - a set of threads wait for each other to be ready
- `Semaphore`
  - Like a lock with a maximum number of holders ("permits")
- `Phaser` – Cyclic barrier on steroids
- `AbstractQueuedSynchronizer` – roll your own!

CountDownLatch example

Concurrent timer

```
public static long time(Executor executor, int nThreads,
          final Runnable action) throws InterruptedException {
    CountDownLatch ready = new CountDownLatch(nThreads);
    CountDownLatch start = new CountDownLatch(1);
    CountDownLatch done = new CountDownLatch(nThreads);
    for (int i = 0; i < nThreads; i++) {
        executor.execute(() -> {
            ready.countDown(); // Tell timer we're ready
            try {
                start.await(); // Wait till peers are ready
                action.run();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                done.countDown(); // Tell timer we're done
            }});}
    ready.await(); // Wait for all workers to be ready
    long startNanos = System.nanoTime();
    start.countDown(); // And they're off!
    done.await(); // Wait for all workers to finish
    return System.nanoTime() - startNanos;
}
```

IV. Concurrent Collections

- Provide high performance and scalability

Unsynchronized	Concurrent
`HashMap`	`ConcurrentHashMap`
`HashSet`	`ConcurrentHashSet`
`TreeMap`	`ConcurrentSkipListMap`
`TreeSet`	`ConcurrentSkipListSet`

You can’t exclude concurrent activity from a concurrent collection

- This works for synchronized collections…

```
Map<String, String> syncMap =
   Collections.synchronizedMap(new HashMap<>());
synchronized(syncMap) {
   if (!syncMap.containsKey("foo"))
       syncMap.put("foo", "bar");
}
```

- But ***not*** for concurrent collections
  - They do their own internal synchronization
  - **Never synchronize on a concurrent collection!**

Concurrent collections have prepackaged read-modify-write methods

- V **putIfAbsent**(K key, V value)
- boolean **remove**,(Object key, Object value)
- V **replace**(K key, V value)
- boolean **replace**(K key, V oldValue, V newValue)
- V **compute**(K key, BiFunction<...> remappingFn);
- V **computeIfAbsent**,(K key, Function<...> mappingFn)
- V **computeIfPresent**,(K key, BiFunction<...> remapFn)
- V **merge**(K key, V value, BiFunction<...> remapFn)

Concurrent collection example: canonicalizing map

```
private static final ConcurrentMap<String, String> map =
       new ConcurrentHashMap<String, String>();

// This implemenation is OK, but could be better
public static String intern(String s) {
    String previousValue = map.putIfAbsent(s, s);
    return previousValue ==
      null ? s : previousValue;
}
```

A better canonicalizing map

- ConcurrentHashMap optimized for read
  - So call `get` first, `putIfAbsent` only if necessary

```
// Good, fast implementation!
public static String intern(String s) {
   String result = map.get(s);
   if (result == null) {
       result = map.putIfAbsent(s, s);
       if (result == null)
           result = s;
   }
   return result;
}
```

Concurrent observer pattern requires open calls

This code is prone to liveness and safety failures!

```
private final List<SetObserver<E>> observers =
       new ArrayList<SetObserver<E>>();
public void addObserver(SetObserver<E> observer) {
   synchronized(observers) { observers.add(observer); }
}
public boolean removeObserver(SetObserver<E> observer) {
   synchronized(observers) { return observers.remove(observer); }
}
private void notifyElementAdded(E element) {
   synchronized(observers) {
   for (SetObserver<E> observer : observers)
       observer.notifyAdded(this, element);  // Callback!
   }
}
```

A decent solution: *snapshot iteration*

```
private void notifyElementAdded(E element) {
    List<SetObserver<E>> snapshot = null;

    synchronized(observers) {
        snapshot = new ArrayList<SetObserver<E>>(observers);
    }

    for (SetObserver<E> observer : snapshot) {
        observer.notifyAdded(this, element); // Open call
    }
}
```

A better solution: CopyOnWriteArrayList

```
private final List<SetObserver<E>> observers =
        new CopyOnWriteArrayList<SetObserver<E>>();

public void addObserver(SetObserver<E> observer) {
    observers.add(observer);
}
public boolean removeObserver(SetObserver<E> observer) {
    return observers.remove(observer);
}
private void notifyElementAdded(E element) {
    for (SetObserver<E> observer : observers)
        observer.notifyAdded(this, element);
}
```

V. Data exchange collections summary

- **`BlockingQueue`** - Supports blocking ops
  - `ArrayBlockingQueue`, `LinkedBlockingQueue`
  - `PriorityBlockingQueue`, `DelayQueue`
  - `SynchronousQueue`
- **`BlockingDeque`** - Supports blocking ops
  - `LinkedBlockingDeque`
- **`TransferQueue`** - `BlockingQueue` in which producers may wait for consumers to receive elements
  - `LinkedTransferQueue`

Summary of `BlockingQueue` methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	n/a	n/a

Summary of `BlockingDeque` methods

- First element (head) methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`addFirst(e)`	`offerFirst(e)`	`putFirst(e)`	`offerFirst(e, time, unit)`
Remove	`removeFirst()`	`pollFirst()`	`takeFirst()`	`pollFirst(time, unit)`
Examine	getFirst()	peekFirst()	n/a	n/a

- Last element (tail) methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`addLast(e)`	`offerLast(e)`	`putLast(e)`	`offerLast(e, time, unit)`
Remove	`removeLast()`	`pollLast()`	`takeLast()`	`pollLast(time, unit)`
Examine	getLast()	peekLast()	n/a	n/a

VI. Executor framework Overview

- Flexible interface-based task execution facility
- Key abstractions
  - `Runnable`, `Callable<T>` - kinds of tasks
- `Executor` - thing that executes tasks
- `Future<T>` - a promise to give you a T
- Executor service - `Executor` that
  - Lets you manage termination
  - Can produce `Future` objects

A very simple executor service example

- Background execution on a long-lived worker thread
  - To start the worker thread:

```
ExecutorService executor =
  Executors.newSingleThreadExecutor();
```

  - To submit a task for execution:

```
executor.execute(runnable);
```

  - To terminate gracefully:

```
executor.shutdown();
```

- Better replacement for our `runInBackground` *and* `WorkQueue` examples from previous lectures.

Other things you can do with an executor service

- Wait for a task to complete

```
Foo foo = executorSvc.submit(callable).get();
```

- Wait for any or all of a collection of tasks to complete

```
invoke{Any,All}(Collection<Callable<T>> tasks)
```

- Retrieve results as tasks complete

```
ExecutorCompletionService
```

- Schedule tasks for execution in the future

```
ScheduledThreadPoolExecutor
```

- Etc., ad infinitum

`ForkJoinPool`: executor service for `ForkJoinTask` instances

```
class SumSqTask extends RecursiveAction {
   final long[] a; final int l, h; long sum;
   SumSqTask(long[] array, int lo, int hi) {
       a = array; l = lo; h = hi;
   }
   protected void compute() {
       if (h - l < THRESHOLD) {
           for (int i = l; i < h; ++i)
               sum += a[i] * a[i];
       } else {
           int m = (l + h) >>> 1;
           SumSqTask rt = new SumSqTask(a, m, h);
           rt.fork(); // pushes task
           SumSqTask lt = new SumSqTask(a, l, m);
           lt.compute();
           rt.join(); // pops/runs or helps or waits
           sum = lt.sum + rt.sum;
       }
   }
}
```

Summary

- `java.util.concurrent` is big and complex
- But it's very well designed
  - Easy to do simple things
  - Possible to do complex things
- Executor framework does for execution what Collections framework did for aggregation
- This talk just scratched the surface
  - But you know the lay of the land and the javadoc is good
- **Always better to use j.u.c than to roll your own!**