JAVA

FUNDAMENTALS

CONCURRENCY API

Bram Inniger
bram.inniger@zeroturnaround.com

March 20, 2017

AGENDA

- Legacy Synchronised Collections
- Concurrent Collections
- Synchronisers

LEGACY SYNCHRONISED COLLECTIONS

JAVA 1.0

- `java.util.Vector`
- `java.util.Hashtable`

- Internally synchronised: synchronisation is **not** optional

JAVA 1.2

- `java.util.ArrayList`
- `java.util.LinkedList`
- `java.util.HashMap`
- `java.util.TreeMap`

- Not thread safe: needs external synchronisation

JAVA.UTIL.COLLECTIONS

- Use synchronised polymorphic wrappers

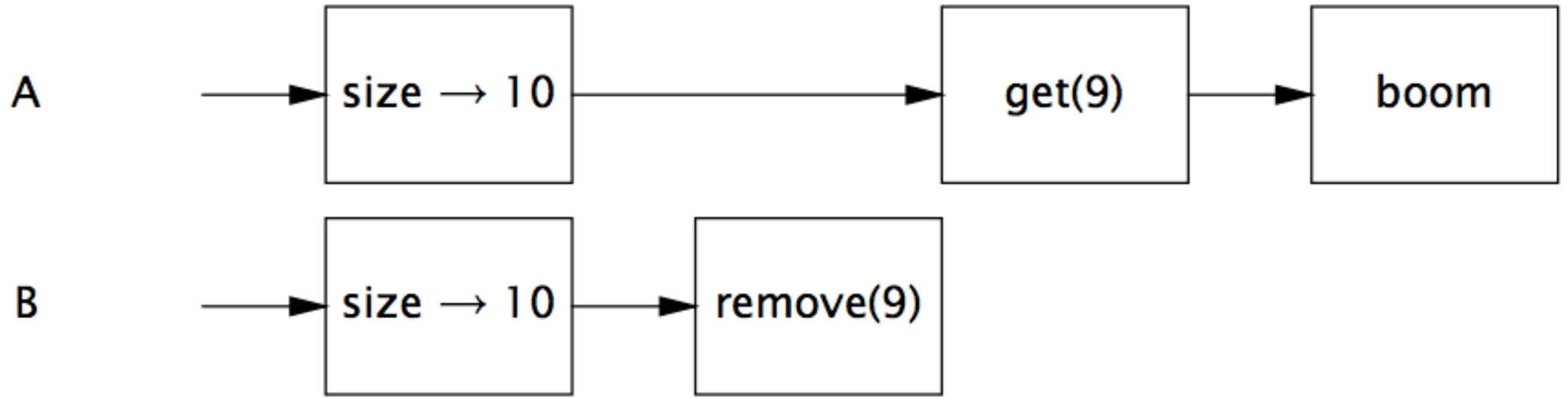
- `Collections.sychronized[List/Map/Set/…]`
- (**Conditionally**) Thread safe

COMPOUND ACTIONS (1)

```
public static Object getLast(Vector list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
}
```

```
public static void deleteLast(Vector list) {
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
}
```

COMPOUND ACTIONS (2)



Interleaving of `getLast()` and `deleteLast()` throws `ArrayIndexOutOfBoundsException`

ATOMIC PRIMITIVES

- `AtomicBoolean`
- `AtomicInteger`
- `AtomicLong`
- `AtomicReference<V>`
- …

Atomic operations -> solves issue of compound actions

ATOMICINTEGER

- incrementAndGet()
- addAndGet(int delta)
- getAndSet(int newValue)
- compareAndSet(int expect, int update)

COMPOUND ACTIONS CLIENT-SIDE LOCKING

```
public static Object getLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public static void deleteLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```

NAIVE ITERATION

```
for (int i = 0; i < vector.size(); i++) {
    doSomething(vector.get(i));
}
```

This may throw `ArrayIndexOutOfBoundsException`

CLIENT-SIDE LOCKING ITERATION

```
synchronized (vector) {
    for (int i=0; i<vector.size(); i++){
        doSomething(vector.get(i));
    }
}
```

CONCURRENTMODIFICATIONEXCEPTION

```
Iterator it = vector.iterator();
while (it.hasNext()) {
    doSomething(it.next());
}
```

Iterators returned by the synchronised collections are **fail-fast**

HIDDEN ITERATORS

```
for (Widget w : widgetList) {
    doSomething(w);
}
```

```
System.out.println("Widgets: " +
                                widgetList);
```

SYNCHRONISE ITERATORS

```
Set s = Collections.synchronizedSet(new HashSet());
synchronized(s) {
   for (Iterator i = s.iterator();
        i.hasNext(); ) {
      doSomething(i.next());
  }
}
```

COPY ON READ (1)

```
for (Iterator it = new
        ArrayList(vector).iterator();
      it.hasNext(); ) {
   doSomething(it.next());
}
```

The constructor may throw
`ArrayIndexOutOfBoundsException` or
`ConcurrentModificationException`

COPY ON READ (2)

```
for (Iterator it =
Arrays.asList(vector.toArray()).iterator();
                                    it.hasNext(); ) {
  doSomething(it.next());
}
```

Finally: Thread safe! **(please don’t use it though)**

PERFORMANCE

The synchronised collections:

- Serialised access —> not efficient
- Only one thread at once (thread contention)
- Reads, writes, iterations all use a single lock

CONCURRENT COLLECTIONS

JSR166 CONCURRENCY UTILITIES

- Chaired by prof. **Doug Lea** (State University of New York at Oswego)

- http://www.jcp.org/en/jsr/detail?id=166
- http://gee.cs.oswego.edu/dl/concurrency-interest/index.html

CONCURRENT COLLECTIONS JAVA 1.5

- `java.util.concurrent` package:
  - `ConcurrentHashMap`
  - `CopyOnWriteArrayList`
  - `CopyOnWriteArraySet`
- Designed for multiple threads
- More efficient
- Replaces synchronised `HashMap`, `ArrayList` and `LinkedHashSet`

CONCURRENTHASHMAP

- Finer-grained locking mechanism: **lock striping**
- Multiple write locks, arbitrary many reads

Threads	ConcurrentHashMap	Hashtable
1	1.00	1.03
2	2.59	32.40
4	5.58	78.23
8	13.21	163.48
16	27.58	341.21
32	57.27	778.41

ADDITIONAL ATOMIC MAP OPERATIONS

```
interface ConcurrentMap<K,V> extends Map<K,V> {
  V putIfAbsent(K key, V value);
  boolean remove(K key, V value);
  boolean replace(K key, V oldVal, V newVal);
  V replace(K key, V newValue);
}
```

No need for client side locking!

PUTIFABSENT(KEY,VALUE)

```
if (map.containsKey(key)) {
   return map.get(key);
}
else {
   return map.put(key, value);
}
```

REPLACE(KEY,OLDVAL,NEWVAL)

```
if (map.containsKey(key)
        && map.get(key).equals(oldVal)) {
   map.put(key, newVal);
   return true;
}
else {
   return false;
}
```

COMPARE AND SET

```
ConcurrentMap<Object, Integer> countHits =
                              new ConcurrentHashMap<>();

private void incrementCount(Object key) {
   Integer oldVal, newVal;
   do {
      oldVal = countHits.get(key);
      newVal = (oldVal==null) ? 1: (oldVal + 1);
   }
   while (!countHits.replace(key, oldVal, newVal));
}
```

THREAD SAFE ITERATION

```
ConcurrentMap<Object, Object> chm =
                        new ConcurrentHashMap<>();
for (Iterator it=chm.iterator(); it.hasNext(); ) {
  doSomething(it.next());
}
```

`ConcurrentMap` iterators are weakly consistent:

- No `ConcurrentModificationException` is thrown
- Modifications and removals are visible
- Insertions **may** be visible, no guarantees

PERFORMANCE

- Parallel access —> more efficient
- Only little overhead versus regular collections
- But: `size()` may either be not atomic, or not correct

LOCK FREE DESIGN

- Spin Lock:
  normally evil, but welcome here, very cheap
  `AtomicInteger.incrementAndGet()`
  `ConcurrentMap.replace(K key, V oldV, V newV)`
- Traditional locking makes Threads go to sleep:
  very expensive context switch
  `java.util.concurrent.locks.Lock`
  `synchronized` / `synchronized(Object)`

COPYONWRITEARRAY[LIST/SET]

- Use when:
  - Care most about traversing the Collection
  - Don’t care about temp inconsistency
- `putIfAbsent()` sequentially scans the array
- `Iterator.remove()` not supported!
- Creates "snapshot" at `Iterator` creation

QUEUE

- Idea: list without random access, add in back, take from front
  - `LinkedList`, basic implementation
  - `PriorityQueue`, allow assigning priorities
  - `ConcurrentLinkedQueue`, thread-safe, lock-free, does not block

BLOCKINGQUEUE

- Wait until an item can be taken / free space appears, thread safe! *Consumer-Producer*
- `ArrayBlockingQueue` (bounded, FIFO)
- `LinkedBlockingQueue` (optionally bounded)
- `SynchronousQueue` (zero capacity)
- `PriorityBlockingQueue` (unbounded, priority)

INTERRUPTIBLE METHODS

```
E take() throws InterruptedException
```

- Blocking method
- Allows to stop blocking
- Can be interrupted with Thread.interrupt()
- Only thread-owner should ever call this

HANDLING CALLING INTERRUPTIBLE METHODS

- Propagate the InterruptedException
- Restore the interrupt:

```
try {
    processTask(queue.take());
}
catch (InterruptedException e) {
    // Restore interrupted status!
    Thread.currentThread().interrupt();
}
```

NON-CANCELABLE TASKS

```
public Task getNextTask(BlockingQueue<Task> queue) {
  boolean interrupted = false;
  try {
      while (true) {
          try {
              return queue.take();
          } catch (InterruptedException e) {
              interrupted = true;
              // fall through and retry
          }
      }
  } finally {
      if (interrupted)
      Thread.currentThread().interrupt();
  }
}
```

TIMEUNIT

- Convenient time conversions:

```
TimeUnit.MILLISECONDS.convert(20L,
                              TimeUnit.SECONDS);
TimeUnit.SECONDS.toMillis(20L);
```

- Collections with timeouts:

```
queue.offer(task,100L, TimeUnit.NANOSECONDS);
```

- Thread.sleep():

```
TimeUnit.SECONDS.sleep(10L);
```

CONCURRENT COLLECTIONS JAVA 1.6

- Interface `ConcurrentNavigableMap`:
  - `ConcurrentSkipListMap` (synchronised `TreeMap`)
- `ConcurrentSkipListSet` (synchronised `TreeSet`)

DEQUE

- Efficient insertion and removal from both the head and the tail
- **Not** thread safe basic implementations:
  - `LinkedList`
  - `ArrayDeque`

BLOCKINGDEQUE

- `LinkedBlockingDeque`
  - Thread safe
  - Blocks
  - Work stealing pattern

CONCURRENT COLLECTIONS JAVA 1.7

- `ConcurrentLinkedDeque`
- Improved `ConcurrentLinkedQueue`
- Interface `TransferQueue`
  - `LinkedTransferQueue`

TRANSFERQUEUE

```
interface TransferQueue<E> extends
                                BlockingQueue<E> {
  boolean tryTransfer(E e)
  void transfer(E e) throws
                               InterruptedException
  boolean hasWaitingConsumer()
  int getWaitingConsumerCount()
  ...
}
```

SYNCHRONISERS

SYNCHRONISER

- Any object that coordinates the **control flow** of threads based on its state
- Decide if an incoming thread can pass or has to wait

LATCH

- Synchroniser that can delay the progress of threads until it reaches its **terminal state**
- Acts as a gate, and remains open, once opened
- Usages:
  Game lobby, waiting for all players to join
  Dependant services startup
  Resources initialisation

COUNTDOWNLATCH

```
class CountDownLatch {
  CountDownLatch(int count);
  void await();
  void await(long timeout, TimeUnit unit);
  void countDown();
  long getCount();
}
```

General idea: every incoming thread calls `countdown()`, then `await()`, until count is reached

FUTURETASK

```
class FutureTask<V> implements Future<V> {
  FutureTask(Callable<V> callable)
  FutureTask(Runnable r, V result)
  void run()
  V get()
 ...
}

interface Callable<V> {
  V call()
}
```

Three states: waiting, running, done (final)

COMPLETABLEFUTURE JAVA 1.8

```
class CompletableFuture<V> implements Future<V> {

CompletableFuture<V> supplyAsync(Supplier<V> supplier)

CompletableFuture<V> thenApply(Function<? super T, ? extends U> fn)

CompletableFuture<Void> thenAccept(Consumer<? super T> action)

boolean complete(T value)

CompletableFuture<T> exceptionally(Function<Throwable,
                                            ? extends T> fn)
```

COMPLETABLEFUTURE CHAINING

```
CompletableFuture
   .supplyAsync(this::findReceiver)
   .thenApply(this::sendMsg)
   .exceptionally(ex->new Result(Status.FAILED))
   .thenAccept(this::notify);
```

SEMAPHORE (1)

Counting semaphores are used to control the number of activities that can access a certain resource or perform a given action at the same time

- Great to implement resource pools (e.g. database connectors)
- Can make any collection both blocking and bounded

SEMAPHORE (2)

```
class Semaphore {
   Semaphore(int permits)
   void acquire()
   void release()
   ...
}
```

- `acquire()` takes permit or blocks
- `release()` returns permit
- Basically, more flexible `Lock`

BARRIER

- Synchroniser that can delay the progress of threads until **all parties have arrived**
- Generally:
  a `Latch` waits for **events**
  a `Barrier` for **threads**

CYCLICBARRIER

```
class CyclicBarrier {
   CyclicBarrier(int parties)
   int await()
   ...
}
```

General idea:

1. Every incoming thread calls `barrier.await()`
2. When the nr of parties is reached, all threads are unblocked
3. All threads on unblock receive their arrival index
4. The barrier resets -> usable multiple times (cyclic)

TAXONOMY

- High level concurrency abstractions
  - `java.util.concurrent`
- Low level locking
  - synchronised blocks & `java.util.concurrent.locks`
- Low level primitives
  - `volatile`, `java.util.concurrent.atomic`

SUMMARY

- Use concurrent collections over synchronised collections

- Use synchronisers instead of implementing your own low-level synchronisation

HOMEWORK

https://github.com/JavaFundamentalsZT/jf-hw-password-cracker

RECOMMENDED READING

Java Concurrency in Practice,
Brian Goetz