



The Hidden Art of Thread-Safe Programming: Exploring `java.util.concurrent`

Dr Heinz M. Kabutz

Last Updated 2026-06-08

Time for Q & A

● Questions:

1. Why is the book for concurrency bigger than that for the Java Language?
2. Load and read barriers between different architectures. Sloppy concurrency coding. How do we code correctly?
3. How do you think in concurrency? How do you test it?
4. When spawning lots of threads, how do you track them during debugging? Can you know which thread spawned which threads?
5. With virtual threads, do we still care about concurrency?
6. Coroutines in Java?

A Tale of `java.util.Vector`

- **One of the first classes in Java**

- Part of Java 1.0

- **Designed thread-safe from concurrent updates**

- Most methods synchronized, locking on "this"

- But missed synchronization on read-only methods like `size()`

Java 1.0 Vector

- **size() could return stale values**

```
public class Vector1_0 {  
    protected int elementCount;  
  
    public final int size() {  
        return elementCount;  
    }  
  
    public final synchronized void addElement(Object obj) {  
        // ...  
    }  
}
```

Moving to Java 1.1

● Introduced a potential race condition

```
public class Vector1_1 implements Serializable {
    protected int elementCount;

    public final int size() {
        return elementCount;
    }

    public final synchronized void addElement(Object obj) {
        // ...
    }
}
```

Moving to Java 1.4

◎ Fixed size() visibility and serialization race condition

```
public class Vector1_4 implements Serializable {
    protected int elementCount;

    public synchronized int size() {
        return elementCount;
    }

    public synchronized void addElement(Object obj) { /* ... */ }

    private synchronized void writeObject(ObjectOutputStream s)
        throws IOException {
        s.defaultWriteObject();
    }
}
```

However, Java 1.4 Can Deadlock!

- **Often, fixing one type of bug, introduces others**

```
Vector v1 = new Vector();  
Vector v2 = new Vector();  
v1.addElement(v2);  
v2.addElement(v1);  
// serialize v1 and v2 from two different threads
```

- Mentioned in The Java Specialists' Newsletter #184
 - <https://www.javaspecialists.eu/archive/Issue184.html>

Moving to Java 1.7

© Fixed deadlock by calling writeFields() outside of lock

```
public class Vector1_7 implements Serializable {
    private void writeObject(ObjectOutputStream s) throws IOException {
        final ObjectOutputStream.PutField fields = s.putFields();
        final Object[] data;
        synchronized (this) {
            fields.put("capacityIncrement", capacityIncrement);
            fields.put("elementCount", elementCount);
            data = elementData.clone();
        }
        fields.put("elementData", data);
        s.writeFields();
    }
}
```

New Potential Deadlock Added in Java 8

- © Should not call "alien methods" like `accept()` whilst locked

```
public class Vector8<E> implements Serializable {
    public synchronized void forEach(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        final int expectedModCount = modCount;
        final E[] elementData = (E[]) this.elementData;
        final int elementCount = this.elementCount;
        for (int i=0; modCount==expectedModCount && i<elementCount; i++) {
            action.accept(elementData[i]);
        }
        if (modCount != expectedModCount) {
            throw new ConcurrentModificationException();
        }
    }
}
```

Takeaways from Vector Bugs

- **Thread safety is subtle**
- **Tests don't always expose concurrency bugs**
 - We need to know what to look for



`java.util.concurrent` Teardown

Writing Correct Thread-Safe Code is a Challenge

- ◎ **The Java Memory Model is our rule book**
 - happens-before, ordering, access safety, etc.
 - However, we cannot test whether a class adheres to the JMM 100%
- ◎ **We run our code, and hope it works correctly**
 - Some bugs are very hard to detect

LockSupport Rare Lost unpark()

● Bug 8074773

- In JDK 7, class loading could consume the unpark()
 - Extremely difficult to diagnose and discover, took a week of CPU time
 - Recommended workaround was to force LockSupport to load early

```
static {  
    // Prevent rare disastrous classloading in first call to LockSupport.park.  
    // See: https://bugs.openjdk.java.net/browse/JDK-8074773  
    Class<?> ensureLoaded = LockSupport.class;  
}
```

- Since JDK 9, ConcurrentHashMap ensures LockSupport is loaded

So Why Study the `java.util.concurrent` Classes?

- **Brian Goetz, JCiP:**

- If you need to implement a state-dependent class the best strategy is usually to build upon an existing library class such as `Semaphore`, `BlockingQueue`, or `CountDownLatch`.

- **By studying `java.util.concurrent` in detail, we learn**

- What is available
- How to write robust, thread-safe classes

Good vs Bad Code

● **We all make mistakes**

- In German, we say: „Vertrauen ist gut, Kontrolle ist besser!“
- Test Driven Development
 - Super difficult with multi-threaded code
 - Java Concurrency Stress can help: github.com/openjdk/jcstress

● **Best to rely on well-known synchronizers**

- And then, use those that are most commonly used
 - Favour ConcurrentHashMap over ConcurrentSkipListMap
 - Favour LinkedBlockingQueue over LinkedBlockingDeque

Contributing Bug Reports

◎ **Anybody can report a Java bug: <https://bugreport.java.com>**

- I've reported quite a few javaspecialists.eu/about/jdk-contributions/
- Most of these were in little used classes
 - 1 in ThreadLocalRandom (fixed in Java 21+9)
 - 1 in ConcurrentSkipListMap (fixed in Java 24)
 - 1 in ArrayBlockingQueue (fixed in Java 24)
 - 2 in LinkedTransferQueue (fixed in Java 1.8.0+70 and 26)
 - 5 in LinkedBlockingDeque (all fixed in Java 26)
- The less used a class is, the higher the chance of bugs

Eat Your Own Dogfood Collections

◎ How many new instances of each in the JDK

- 213: ConcurrentHashMap
- 11-24: CopyOnWriteArrayList, ConcurrentLinkedQueue, ConcurrentLinkedDeque, FutureTask, LinkedBlockingQueue
- 2-6: CountdownLatch, ArrayBlockingQueue, SynchronousQueue, ConcurrentSkipListSet
- 1: ConcurrentSkipListMap, LinkedBlockingDeque, LinkedTransferQueue, Semaphore
- 0: CopyOnWriteArraySet, CyclicBarrier, Exchanger, Phaser, PriorityBlockingQueue

Let's Say That Again

- **Use extremely common thread-safe classes**
 - ConcurrentHashMap
 - LinkedBlockingQueue
 - ConcurrentLinkedQueue
- **I only found bugs in rarely used classes**

Before we continue ...

● **Get our Data Structures in Java Course here**

- Will also post link in chat
- Coupon expires tomorrow
 - You have life-time access to course
 - But try complete before end July 2026

● **In-house courses @ Commerzbank**

- Either in-person or remote
- Also, self-study courses
 - But please contact me first re EU VAT





Lessons from Striped64

LongAdder vs AtomicLong

- © **Let's do a quick comparison of incrementing 100m times**
 - AtomicLong vs LongAdder (Striped64) (Demo1)

```
IntStream.range(0, 100_000_000)
    .parallel()
    .forEach(_ -> atomicLong.getAndIncrement());
```

```
IntStream.range(0, 100_000_000)
    .parallel()
    .forEach(_ -> longAdder.increment());
```



Demo

- **Magic? Let's look at how LongAdder / Striped64 works**



Takeaways

- © **Best way to deal with contention is to not have any**





Changing Hardware Landscape

Changing Hardware Landscape

● **Started coding Java in 1997**

- 64 MB of RAM, single core, 233 MHz, 32-bit

 - And that was one of the better machines in the company in 1997

- My laptop has 96 GB of RAM, 12 cores, 38 GPU cores, 3.7GHz, 64-bit

● **Memory was scarce**

- Could not imagine creating a collection with billions of entries

- Only platform threads - limited to thousands

Bugs at the Limits

◎ Oodles of memory and virtual threads

- Bug in `LinkedBlockingDeque` allowed us to fill it with too many items
 - `size()` returned a negative value
 - www.javaspecialists.eu/archive/Issue328.html
 - Fixed in Java 26
- Bug in `ReentrantReadWriteLock` ran out of read locks after 65536
 - Resulted in `Error` being thrown
 - Could not have conceived a system with that many threads
 - Fixed in Java 25

◎ Demo2: ManyReadLocks



StartingGun Synchronizer

StartingGun Synchronizer

- ◎ **Let's say we have a service that takes time to be started**
 - Any other part of the system that depends on it should wait
 - But we do not want to deal with InterruptedException
 - Once all the data is set up, we call ready(), awaking waiting threads

```
public interface StartingGun {  
    void awaitUninterruptibly();  
    void ready();  
}
```

Using `synchronized` and `wait()/notifyAll()`

```
public class StartingGunMonitor implements StartingGun {
    private boolean ready = false;
    public synchronized void awaitUninterruptibly() {
        boolean interrupted = Thread.interrupted();
        while (!ready) {
            try {
                wait(); // not fully compatible with older Loom versions
            } catch (InterruptedException e) {
                interrupted = true;
            }
        }
        if (interrupted) Thread.currentThread().interrupt();
    }
    public synchronized void ready() { ready = true; notifyAll(); }
}
```

Basing StartingGun on CountdownLatch

```
public class StartingGunCountDownLatch implements StartingGun {
    private final CountdownLatch latch = new CountdownLatch(1);
    public void awaitUninterruptibly() {
        var interrupted = Thread.interrupted();
        while (true) {
            try {
                latch.await();
                break;
            } catch (InterruptedException e) {
                interrupted = true;
            }
        }
        if (interrupted) Thread.currentThread().interrupt();
    }
    public void ready() { latch.countDown(); }
}
```

Issues With These Approaches

- ◎ **Synchronized wait() not fully compatible with virtual threads**
 - Fixed in Java 24
- ◎ **Both times, interrupt would cause InterruptedException**
 - We hide it, but we still pay the cost of creating the exception
- ◎ **Another way is to copy what CountdownLatch does**
 - Demo3



Lock Splitting: LinkedBlockingQueue

LinkedBlockingQueue Design

- ◎ **Single lock would cause put()/take() contention**
- ◎ **Has separate putLock and takeLock ReentrantLock**
 - We can put() and take() from a single queue at the same time
 - Has higher throughput for the SPSC case
 - And surprises for the SPMC case
 - Subtleties regarding visibility due to two locks
 - Use AtomicInteger count as a volatile synchronizer
- ◎ **Demo4 LockSplittingDemo**



Weakly Consistent Iterators – ArrayBlockingQueue

ArrayBlockingQueue Circular Array Queue

◎ Weakly consistent iteration

- ArrayDeque would cause a ConcurrentModificationException

```
var queue = new ArrayBlockingQueue<Integer>(10);
Collections.addAll(queue, 1, 2, 3, 4, 5);
var iterator = queue.iterator();
for (int i = 0; i < 3; i++) System.out.println(iterator.next()); // 1, 2, 3
Collections.addAll(queue, 6, 7, 8, 9, 10);
iterator.forEachRemaining(System.out::println); // 4, 5, 6, 7, 8, 9, 10
```

- However, what if we circle completely around the array?
 - ArrayBlockingQueue has to notify its current iterators
 - But how?



Double-Checked-Locking – CopyOnWriteArrayList

CopyOnWriteArrayList DCL

● In several places, checks before locking

```
public boolean remove(Object o) {  
    Object[] snapshot = getArray();  
    int index = indexOfRange(o, snapshot, 0, snapshot.length);  
    return index >= 0 && remove(o, snapshot, index);  
}  
// also addIfAbsent(E e),
```

● Demo6 DCLOnSteroidsCOWDemo



Conclusion

The Java Specialists' Newsletter

- **Join our Java Specialists community**
 - www.javaspecialists.eu/archive/subscribe/
- **Readers in 150+ countries**
- **25 years of newsletters on advanced Java**
 - All previous newsletters available on www.javaspecialists.eu
 - Longest running Java newsletter in the world
 - Courses, consulting, additional training, etc.



Don't Forget ...

● **Get our Data Structures in Java Course here**

– Coupon expires tomorrow

• You have life-time access to course

