JavaSpecialists

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

Dr Heinz M. Kabutz

Last Updated 2025-05-07

Time for Q & A

- **Questions:**
  - 1. Any classes in java.util.concurrent we should avoid like the plague
  - 2. Which of the classes has changed the most?
  - 3. How much did you have relearn with virtual threads?

JavaSpecialists.eu

A Tale of java.util.Vector

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

  – **Part of Java 1.0**

- **Was designed to be thread-safe from concurrent updates**

  – **Most methods synchronized, locking on "this"**

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

JavaSpecialists.eu

Java 1.0 Vector

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

```java
public class Vector1_0 {
    protected int elementCount;

    public final int size() {
        return elementCount;
    }

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

JavaSpecialists.eu

Moving to Java 1.1

- **Introduced a potential race condition**

```java
public class Vector1_1 implements java.io.Serializable {
    protected int elementCount;

    public final int size() {
        return elementCount;
    }

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

JavaSpecialists.eu

Moving to Java 1.4

- **Fixed size() visibility and serialization race condition**

```java
public class Vector1_4 implements java.io.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**

JavaSpecialists.eu

Moving to Java 1.7

- **Fixed deadlock by calling writeFields() outside of lock**

```java
public class Vector1_7 implements Serializable {
    private void writeObject(java.io.ObjectOutputStream s)
            throws java.io.IOException {
        final java.io.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();
    }
}
```

JavaSpecialists.eu

New Potential Deadlock Added in Java 8

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

```java
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**

JavaSpecialists.eu

JavaSpecialists

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 if a class adheres to the JMM 100%**

- **We run our code, and hope it works correctly**

  – **Some bugs are very hard to detect**

JavaSpecialists.eu

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**

JavaSpecialists.eu

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**

JavaSpecialists.eu

Good vs Bad Code

- **We all make mistakes**

  – **In German, we say: „Vertrauen ist gut, Kontrolle ist besser!"**

  – **Test Driven Development**
    - **But very difficult to do with multi-threaded code**

- **Better to rely on well-known synchronizers**

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

JavaSpecialists.eu

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 LinkedTransferQueue**
    - **1 in ThreadLocalRandom**
    - **1 in ConcurrentSkipListMap**
    - **1 in ArrayBlockingQueue**
    - **5 in LinkedBlockingDeque**
  - **The less used a class is, the higher the chance of bugs**

JavaSpecialists.eu

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**

- **All the bugs I found were in classes that are not used a lot**

JavaSpecialists.eu

Before we continue ...

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

– **tinyurl.com/devoxx-uk-25**

– **Coupon expires Friday at 18:00**

  • **You have life-time access to course**

● **For those watching the recording**

– **Sign up to The Java Specialists' Newsletter**

  • **www.javaspecialists.eu**

  • **Reply to the welcome mail that you would like this course**

JavaSpecialists

Lessons from Striped64

LongAdder vs AtomicLong

- **Let's do a quick comparison of incrementing 100m times**

  - **AtomicLong vs LongAdder (Striped64)**

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


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

tinyurl.com/devoxx-uk-25

Demo

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

tinyurl.com/devoxx-uk-25

Takeaways

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

tinyurl.com/devoxx-uk-25

JavaSpecialists.eu

JavaSpecialists

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**

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

JavaSpecialists.eu

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

```java
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(); }
}
```

JavaSpecialists.eu

Basing StartingGun on CountDownLatch

```java
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(); }
}
```

JavaSpecialists.eu

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**

- **A better way is to copy what CountDownLatch does**

  – **Quick demo**

JavaSpecialists.eu

JavaSpecialists

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**

- **Demo LockSplittingDemo**

JavaSpecialists.eu

JavaSpecialists

Weakly Consistent Iterators – ArrayBlockingQueue

ArrayBlockingQueue Circular Array Queue

- **Weakly consistent iteration**

  - **ArrayDeque would cause a ConcurrentModificationException**

```java
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?**

- **Demo WeaklyConsistentViaWeakReferences** JavaSpecialists.eu

Double-Checked-Locking – CopyOnWriteArrayList

CopyOnWriteArrayList DCL

- ## In several places, checks before locking

```java
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),
```

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

- ## Demo DCLOnSteroidsCOWDemo

JavaSpecialists.eu

JavaSpecialists

Conclusion

The Java Specialists' Newsletter

- **Make sure to subscribe**

  – **www.javaspecialists.eu/archive/subscribe/**

- **Readers in 150+ countries**

- **Over 24 years of newsletters on advanced Java**

  – **All previous newsletters available on www.javaspecialists.eu**

  – **Courses, additional training, etc.**

JavaSpecialists.eu

Don't forget ...

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

  – **tinyurl.com/devoxx-uk-25**

  – **Coupon expires Friday at 18:00**
    - **You have life-time access to course**

- **For those watching the recording**

  – **Sign up to The Java Specialists' Newsletter**
    - **www.javaspecialists.eu**
    - **Reply to the welcome mail that you would like this course**