# Java Concurrency in Practice

Book overview

*Paweł Oczadly*

# Introduction

Based mostly on:

Brian Goetz et al. (2006), Java  Concurrency in Practice, Addison-Wesley

Citations (or paraphrases) from this book are marked with the page number

Examples used in presentation are accessible here:

https://github.com/Pawlllosss/Java-Concurrency-in-Practice

# Agenda

- What is thread safety?
- Invariants and post conditions.
- How can we achieve thread safety?
- Blocking vs non blocking synchronization.

# What is thread safety?


https://uynguyen.github.io/2018/06/05/Working-In-Thread-Safe-on-iOS/

# What is thread safety?

It is about correctness - fact that class conforms to it’s well-defined specification - invariants, postconditions.

From the class perspective: Clients of the class doesn’t need to provide their own synchronization.

Correct behaviour of a class when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the client code.

# Invariants

It’s crucial to preserve invariants.
Single invariant can consist of more than one variable.

```
public class IntRange {
   private final int lower;
   private final int upper;

   public IntRange(int lower, int upper) {
      this.lower = lower;
      this.upper = upper;
   }

   public int getLower() {
      return lower;
   }

   public int getUpper() {
      return upper;
   }
}
```

Making atomic access to **lower** and **upper** (i.e. synchronized or **AtomicInteger**) is not enough. We need also to ensure that **lower < upper.**

All variables in this invariant must be guarded by the same lock.

# Invariants other examples

Each object and variable has its **state space** - possible states.

The smaller **state space**, simpler to maintain and implement it.

Long - from **Long.MIN_VALUE** to **Long.MAX_VALUE**

Counter - values from 0 to **Long.MAX_VALUE**

Operation **post condition** - only possible next state

is greater by 1

```
public class CounterIntrinsic {

  private long value = 0;

  public synchronized long getValue() {
    return value;
  }

  public synchronized long increment() {
    if (value == Long.MAX_VALUE) {
      throw new
IllegalStateException("Maximum value of
long");
    }
    this.value = value + 1;
    return this.value;
  }
}
```

# How can we achieve thread safety?

- intrinsic locks (`synchronized` keyword)
- explicit locks (like *ReentrantLock)*
- immutability
- stateless code
- thread confinement
- delegate thread safety - let some other class bother about it, like *AtomicInteger* or *ConcurrentHashMap*

# Intrinsic locks

Every Java object can be used as a lock - **intrinsic lock**.

It can be acquired by entering a **synchronized block** or **synchronized method**.

They act as a **mutex** - at most one thread may own the lock

It's **reentrant** - if thread tries to acquire lock that it already hold, the request succeeds (locks are acquired **per-thread** not per-invocation). It makes it possible to use inheritance together with intrinsic locks.

# Inheritance with the intrinsic lock

```
public class IntrinsicExample {
   public static void main(String[] args) {
      ServiceWithLogging serviceWithLogging =
new ServiceWithLogging();
      Runnable runServiceWithLogging = () -> {
         try {
            serviceWithLogging.doAction();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      };
   new Thread(runServiceWithLogging).start();
   new Thread(runServiceWithLogging).start();
   }
}
```

```
class Service {
   public synchronized void doAction() throws InterruptedException {
      System.out.println("In Service");
      Thread.sleep(500);
      System.out.println("Out of Service");
   }
}

class ServiceWithLogging extends Service {
   @Override
   public synchronized void doAction() throws InterruptedException {
      System.out.println("Starting an action " +
Thread.currentThread().getName());
      super.doAction();
      System.out.println("Action completed");

System.out.println("==================================
=====");
   }
}
```

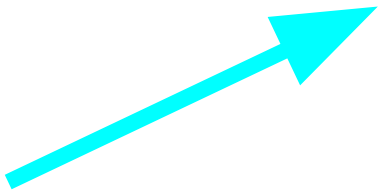
It's possible to call **super.doAction()**, because lock is reentrant.

# How to create an immutable object

- State cannot be modified after construction
- All its field are final
- It is properly constructed
  (*this* reference does not escape during construction)

```
public class ThisEscape {
  private final int value;
  public ThisEscape(EventSource
eventSource) {
    eventSource.registerListener(new
EventListener() {
      @Override
      public void onEvent(Event event) {
        checkSanity();
      }
    });
    value = 42;
  }

  private void checkSanity() {
    if (value != 42) {
      System.out.println("It's insane!");
    }
  }
}
```



*this* reference of the class *ThisEscape* escapes through the constructor of the anonymous class *EventListener*. The reference is added implicitly to its constructor.

# State escape (and hidden iterator)

It’s not only a threat to the immutable classes.

```
public class StateEscape {

  private List<String> states = new LinkedList<>(Arrays.asList("user1", "user2", "user3"));

  public synchronized List<String> getStates() {
      return states;
  }

  public static void main(String[] args) {
      StateEscape stateEscape = new StateEscape();
      List<String> states = stateEscape.getStates();
      System.out.println(states);
      states.set(2, "maliciousUser1");
      // hidden iterator for states, with multiple threads may cause ConcurrentModificationException
      System.out.println(states);
  }
}
```

# Why immutability is useful?

Immutable objects are always thread safe

If properly created they will be only in one state, which is controlled by constructor.

It’s safer - no risk of being modified by some untrusted code, no need of making defensive copies.

# How to safely construct and publish object?

To publish an object safely, both the reference to the object and the object's state must be made visible to other threads at the same time.

- Initializing an object reference from a static initializer [JLS 12.4.2]

```
public static IntWrapper wrapper = new IntWrapper(13);
```

- Storing a reference to it into a *volatile* field or *AtomicReference*

```
private volatile IntWrapper wrapperVolatile;
```

```
private AtomicReference<IntWrapper> wrapperAtomic = new AtomicReference<>(new IntWrapper(31));
```

- Storing a reference to it into a *final* field of a properly constructed object
- Storing a reference to it into a field that is properly guarded by a lock (synchronization)

```
private IntWrapper guardedByLock;
public synchronized void initialise() {
  guardedByLock = new IntWrapper(47);
}
public synchronized int getValue() {
  return guardedByLock.getValue();
}
```

Immutable objects doesn't need any precautions to be safely published.

Effectively immutable objects need to be safely published.

# What can go wrong when publishing an object?

```
public class IntWrapper {

  private int value;

  public IntWrapper(int value) {
    this.value = value;
  }

  public void validateSanity() {
    if (value != 0) {
      throw new IllegalStateException("value is still 0");
    }
  }
}
```

1. Other threads can see a null reference or old reference
2. Other threads can see old state of the object. (**IntWrapper** will throw **IllegalStateException**) - before **IntWrapper** constructor, the **Object** constructor writes default value to all fields

# Thread confinement

If data is not shared between threads then no synchronisation is needed.

**No sharing, no problem.**

Java have no mechanism to enforce that. It's provided by the program design. Using *local variables* or *ThreadLocal* class helps to achieve that, but it's the programmer responsibility to ensure that objects do not escape from the thread.

- Ad-hoc thread confinement
- Stack confinement
- ThreadLocal

# Ad-hoc confinement

The most fragile approach - completely based on the implementation.

Example - volatile variable written from the single thread can be safely read by multiple threads.

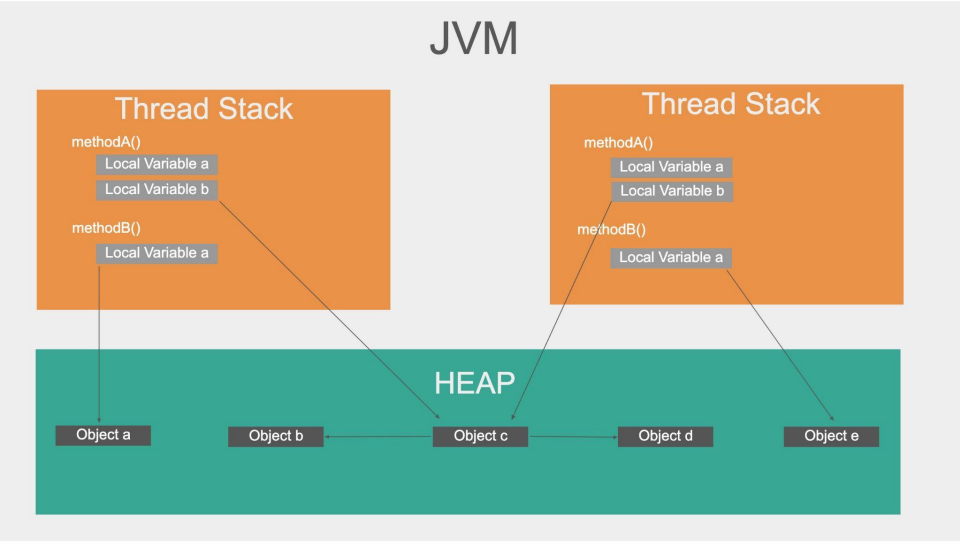
```
// only one thread can write to it
// I rely only on the fact that other devs will consider this comment - that's why it's ad-hoc
private volatile int value;
```

# Stack confinement

Object can be reached only through the local variables.

Local variables exist on the thread stack, which is inaccessible by other threads.

Primitives are extra safe, because there's no way to escape reference!




https://medium.com/javarevisited/java-concurrency-java-memory-model-96e3ac36ec6b

# ThreadLocal

Conceptually, can be considered as *Map<Thread,T>*.

*ThreadLocal* lets to maintain a separate copy of value for each thread.

*RequestContextHolder* in Spring uses it to expose web request as thread-bound object

```
public class UserContext {
  private static ThreadLocal<Long> currentSchemaId = new ThreadLocal<>();

  public static Long get() {
     return currentSchemaId.get();
  }

  public static void set(Long id) {
     currentSchemaId.set(id);
  }
}
```

Global user context for multi-tenant application. Where each request is associated with one user.

# Blocking vs Nonblocking synchronization

| **Blocking** | **Nonblocking** |
|---|---|
| Easier to design | Harder to design |
| Bigger scheduling overhead | Lower scheduling overhead - no block when multiple threads contend for the same data |
| May cause deadlock or other liveness failures | Immune to individual thread failures |
| Pessimistic technique | Optimistic technique |
| Uses locks | Uses *compare-and-swap* (CAS) instruction |

# CAS

Compare-and-swap instruction

**CAS** v a b

v - memory location to operate

a - expected old state

b - new value

Java supports those instruction with use of the AtomicXXX variables.

On platforms supporting CAS, the JVM uses appropriate machine instruction. If not the JVM uses spin lock.

# Preserve multivariable invariant using AtomicReference

```
public class CasNumberRange {
  private AtomicReference<IntRange> intRange = new AtomicReference<>(new IntRange(0, 0));

  public int getLower() {
    return intRange.get().getLower();
  }

  public void setLower(int lower) {
    while (true) {
      IntRange oldRange = intRange.get();
      int upper = oldRange.getUpper();
      if (lower > upper) {
        throw new IllegalArgumentException("Lower value is greater than upper");
      }
      IntRange newRange = new IntRange(lower, upper);
      if (intRange.compareAndSet(oldRange, newRange)) {
        //successfully changed
        return;
      }
    }
  }
}
```

With use of the *AtomicReference* we can preserve invariant, which consists of 2 variables (*lower*, *upper*)

# Sources other than this book

- https://stackoverflow.com/questions/3705425/java-reference-escape
- https://www.javaspecialists.eu/archive/Issue192-Implicit-Escape-of-this.html