Concurrent Collections and Atomic Variables

Maaike van Putten
Software Developer & Trainer

www.brightboost.nl

Overview

Concurrent collections

- Different collections
- Synchronized collections

Atomic variables

Problem with collections

Collections are not thread safe. Using the normal collections in a multi-thread environment will lead to problems with data integrity.

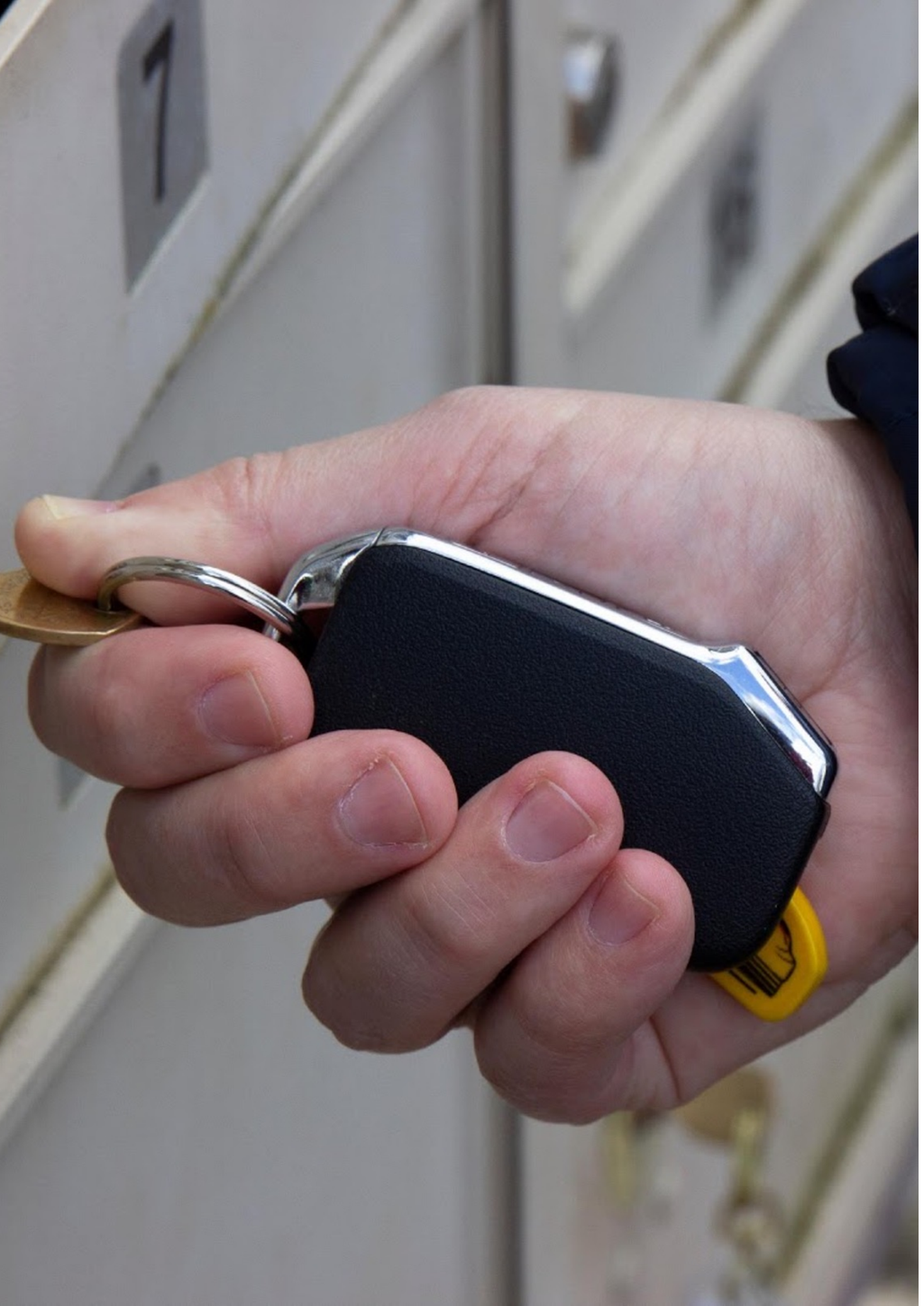
Example of The Problem

```
import java.util.HashMap;
import java.util.Map;

public class CollectionProblems {
    public static void main(String[] args) {
        Map<String, String> stringStringMap = new HashMap<>();
        stringStringMap.put("Maaike", "Java");
        stringStringMap.put("Remsey", "C#");

        for(String k : stringStringMap.keySet()) {
            System.out.println(k + " loves coding " + stringStringMap.get(k));
            stringStringMap.remove(k);
        }
    }
}
```

Concurrent collections to the rescue!

Solution: Concurrent Collections

```
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class CollectionProblemsSolution {
    public static void main(String[] args) {
        Map<String, String> stringStringMap = new ConcurrentHashMap<>();
        stringStringMap.put("Maaike", "Java");
        stringStringMap.put("Remsey", "C#");

        for(String k : stringStringMap.keySet()) {
            System.out.println(k + " loves coding " + stringStringMap.get(k));
            stringStringMap.remove(k);
        }
    }
}
```



Concurrent collections allow locking per segment

Multiple threads can get read access

Assure data integrity

Change collection while looping

We could achieve the same writing wrappers with synchronized or use synchronized collections, but lower performance

Interfaces in Package java.util.concurrent

ConcurrentMap

Child interface of the Java.util.Map

Atomic operations for adding, removing and replacing key-value pairs

Implementing classes: ConcurrentHashMap and ConcurrentSkipListMap

Has the child interface ConcurrentNavigableMap

BlockingQueue

Child interface of Queue

First-in-first-out collection

Thread-safe implementation with internal locks in the methods

Blocks when you want to get an item from an empty queue

Blocks when you add to a full queue

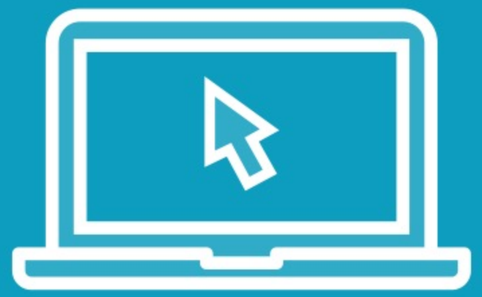
Blocking queues

SkipList

CopyOnWrite

Different Collections

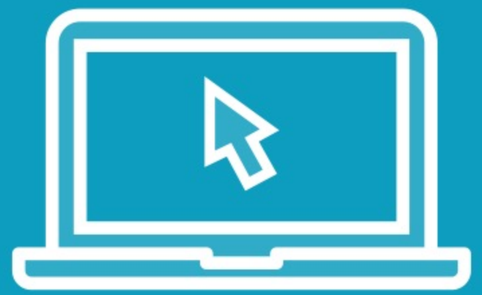
Demo



Let’s have a look at the different blocking queues collections in action

Methods on blocking queues

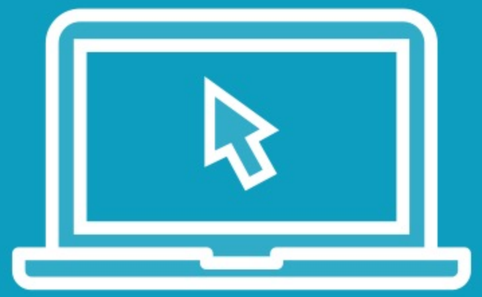
Demo



Let’s have a look at the SkipList collections in action

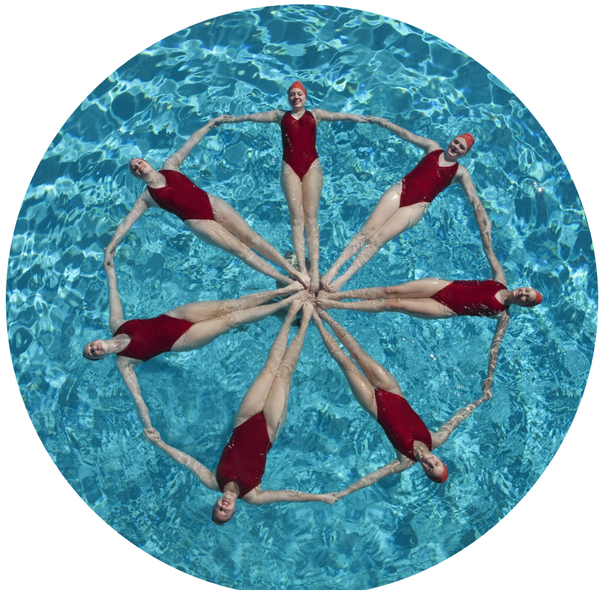
Methods on SkipList collections

Demo



Let’s have a look at the different CopyOnWrite collections in action

Methods on CopyOnWrite collections



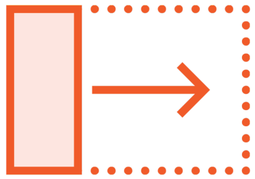
Getting synchronized collections with special methods on Collections

- .synchronizedList(list)
- .synchronizedMap(map)
- .synchronizedSet(set)
- .synchronizedCollection(collection)
- And more...

Example of Synchronized Collection

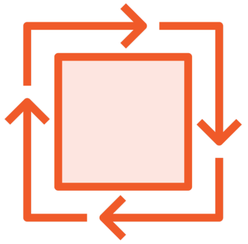
```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SynchronizedCollection {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>(List.of(1, 2, 3, 4, 5));
        var syncList = Collections.synchronizedList(list);

        //go ahead and use syncList safely with multiple threads
    }
}
```

Synchronized Collections



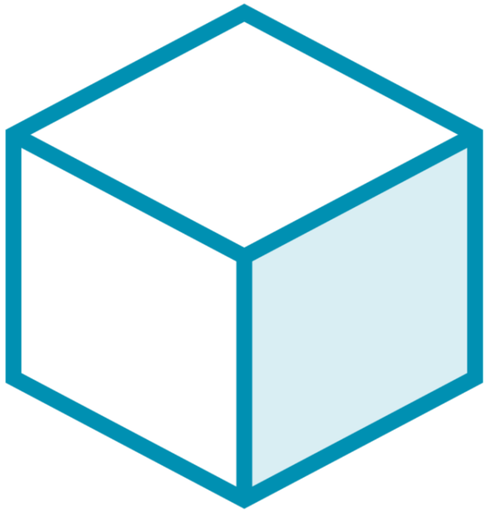
Great way of turning an existing collection into a synchronized and memory safe one

Bad performance, better to choose concurrent collections



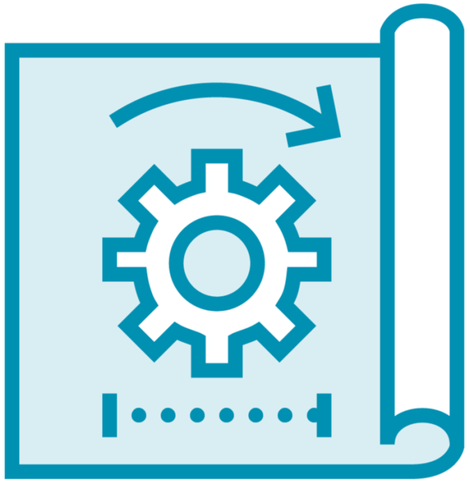
Cannot be modified in a loop

Atomic Variables



java.util.concurrent.atomic package

Memory consistent



Special methods

Most Common Atomic Variables

AtomicInteger

Special integer that gets changed using atomic operations

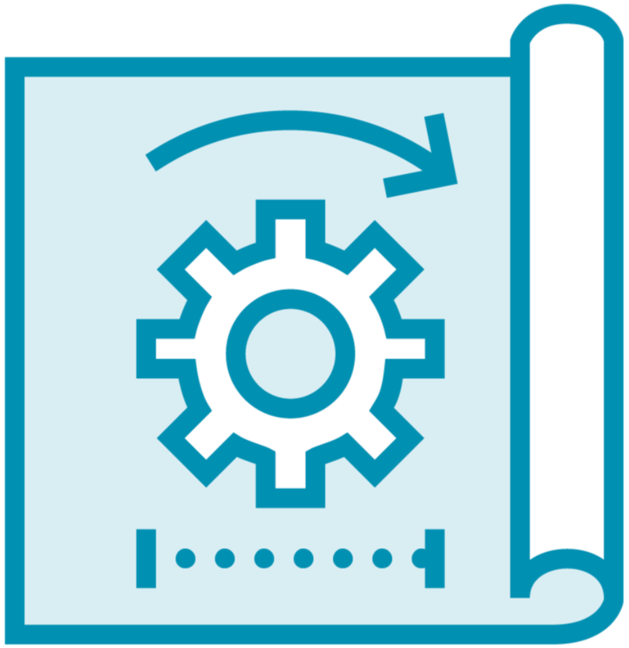
AtomicLong

Special long that gets changed using atomic operations

AtomicBoolean

Special boolean that gets changed using atomic operations

Methods on Atomic Variables



get()

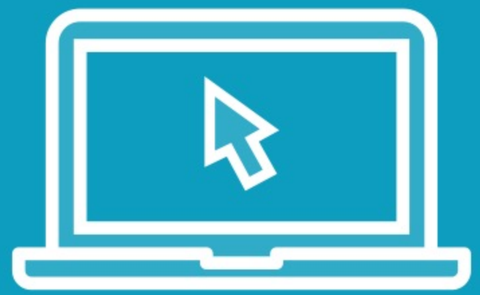
set()

compareAndSet()

weakCompareAndSet()

lazySet()

Demo

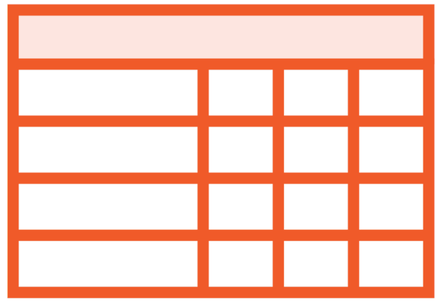


Create atomic variables

Use their methods

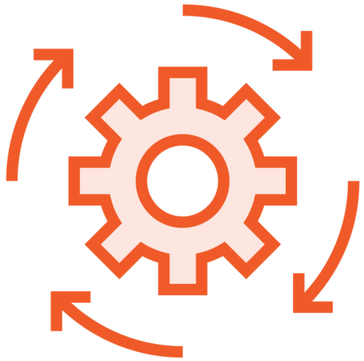
Methods on AtomicInteger and AtomicLong

Since these are numeric types, they have some methods that are not available to all atomic variables.



getAndIncrement & incrementAndGet

Getting and increasing the value atomically



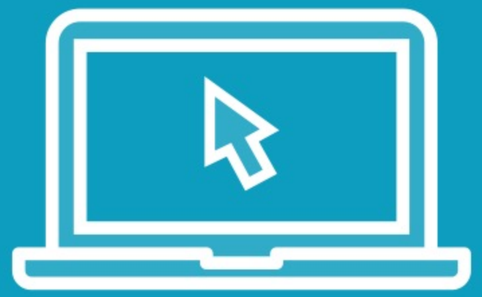
getAndDecrement & decrementAndGet

Getting and dcreasing the value atomically

getAndAdd & addAndGet

Getting and changing the value atomically

Demo



Counting concurrently with AtomicLong and AtomicInteger

Methods on AtomicLong and AtomicInteger

Summary

Concurrent collections

- Blocking queues
- SkipList collections
- CopyOnWrite collections

Atomic variables

- AtomicInteger
- AtomicLong
- AtomicBoolean

Up Next:
Threading Problems

Threading Problems

Maaike van Putten
Software Developer & Trainer

www.brightboost.nl

Overview

Liveness of an application

Common problems with concurrency

Different threading problems

- Deadlock
- Livelock
- Starvation
- Race condition

Liveness

Liveness is the state of an application describing how healthy it is. This depends on readiness for requests and responsiveness to requests.

Memory management

Data integrity

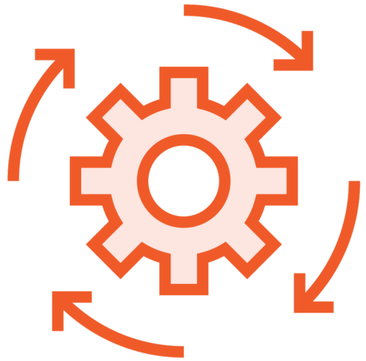
Availability and stability

Threading Problems

Threading problems are a danger to the liveness, memory consistency and data integrity.

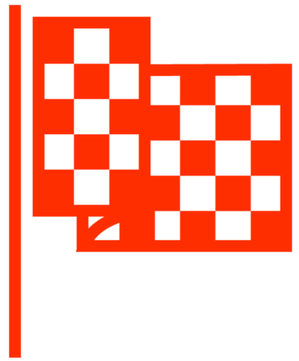
Deadlock and livelock

Multiple threads waiting indefinitely on each other or triggering each other in a loop



Starvation

Low priority thread cannot get access to a resource due to high priority threads



Race condition

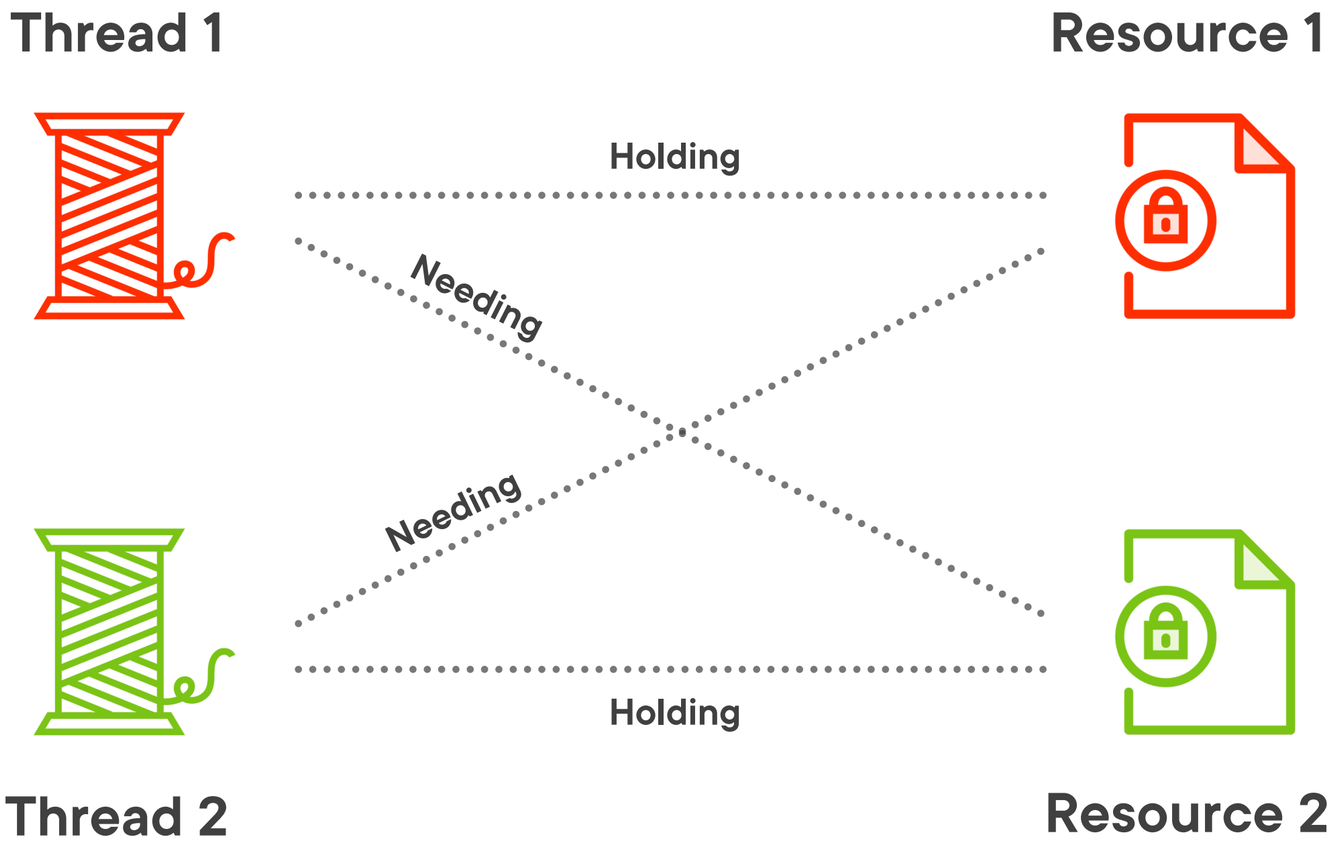
Multiple threads use the same resource and result depends on order of threads accessing

Deadlock

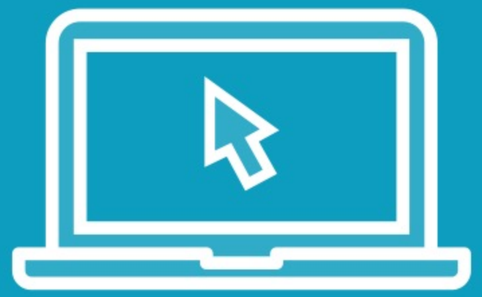
Waiting state of threads, no progress can be made

Threads are holding the resource the other thread(s) need and the other thread(s) holding what the current thread needs

Deadlock


Thread 1
Resource 1
Holding
Needing
Needing
Holding
Thread 2
Resource 2

Demo



Run deadlock code

Examine deadlock code

- Why it happens
- How to solve it

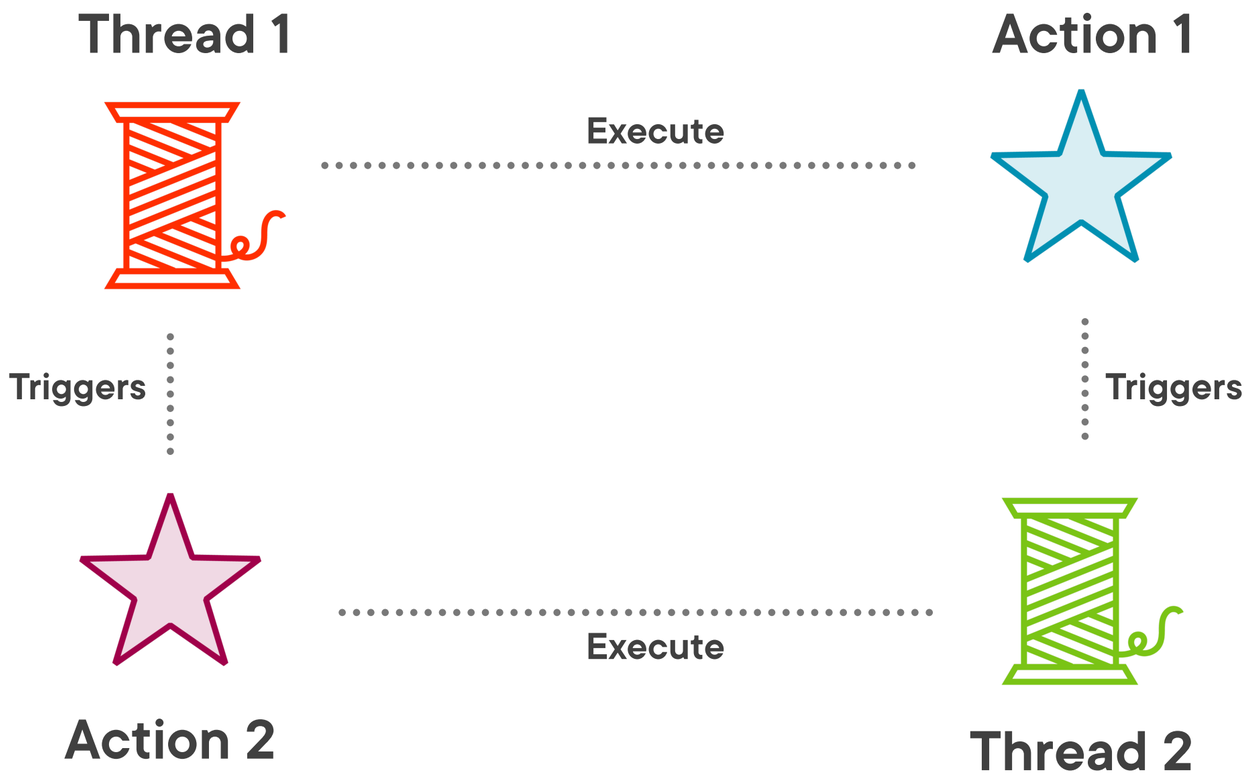
Livelock

Threads are triggering each other to do the same action repeatedly

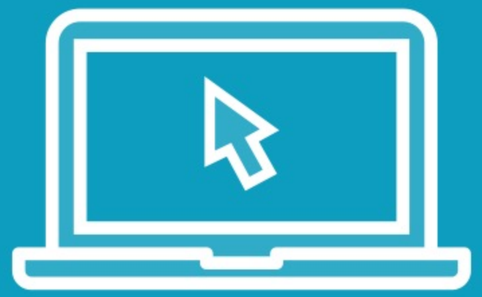
Stuck in a loop, but can end by itself in certain situations

Terrible for performance and hard to spot

Livelock


Thread 1
Action 1
Execute
Triggers
Triggers
Execute
Action 2
Thread 2

Demo



Run livelock code

Examine livelock code

- Why it happens
- How to solve it

Starvation

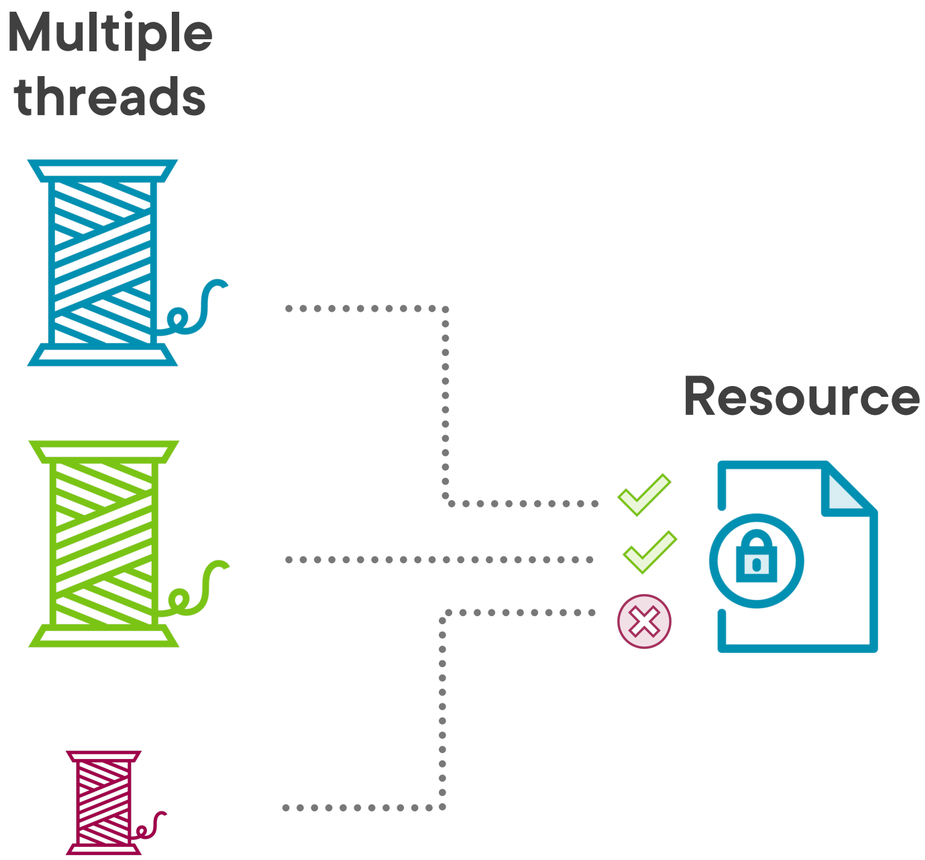
Normally, all threads get the access to resources they need

Starvation: thread with low priority cannot progress due to high priority threads

No access to needed resource due to the resource being high in demand by high priority threads

Endangers liveness of the application

Starvation

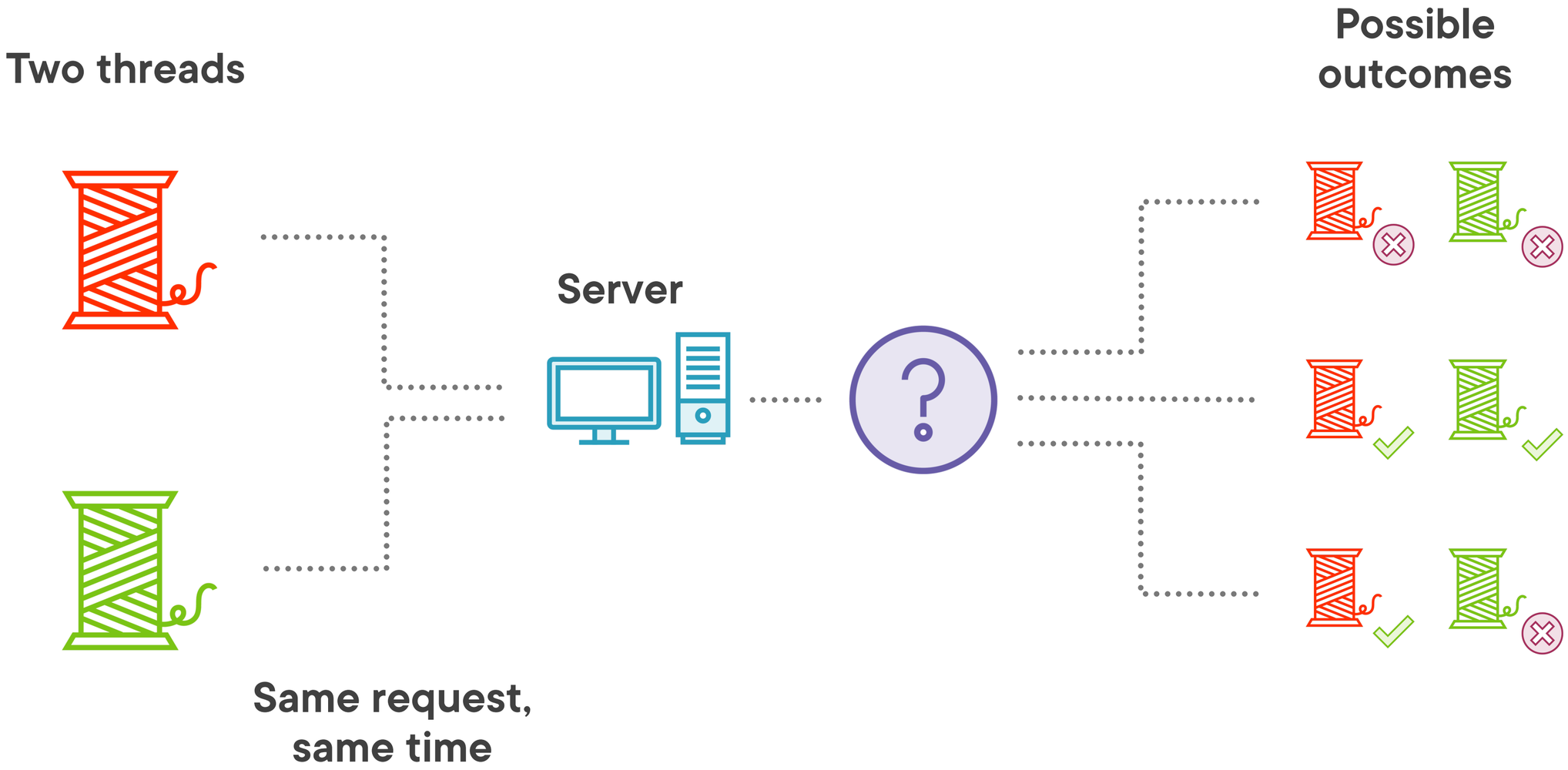


Race Condition

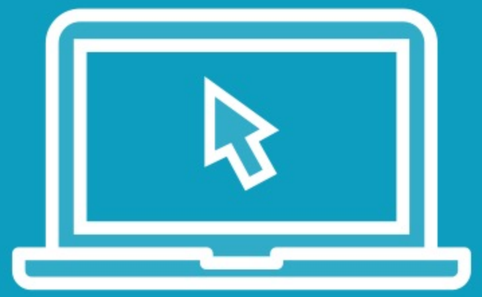
Multiple threads need access to the same resource

Outcome of the operations depends on (coincidental) order of execution

Race Condition



Demo



Run race condition code

Examine race condition code

- Why it happens
- How to solve it

Summary

Threading problems

- Memory inconsistency
- Liveness

Different types of threading problems

- Deadlock
- Livelock
- Starvation
- Race condition