Understanding Mutexes

David Flynn

SOFTWARE ENGINEER, FLYNN IT LTD

In the Last Module...

Saw how to share memory safely and avoid data races

- Using volatile
- Publishing immutable objects

Is this the only worry we might have?

Race Conditions

Race conditions lead to:

- Inconsistent data
- Invariants broken

Volatile cannot help

Publishing objects helps if only one thread may write

Mutual Exclusion

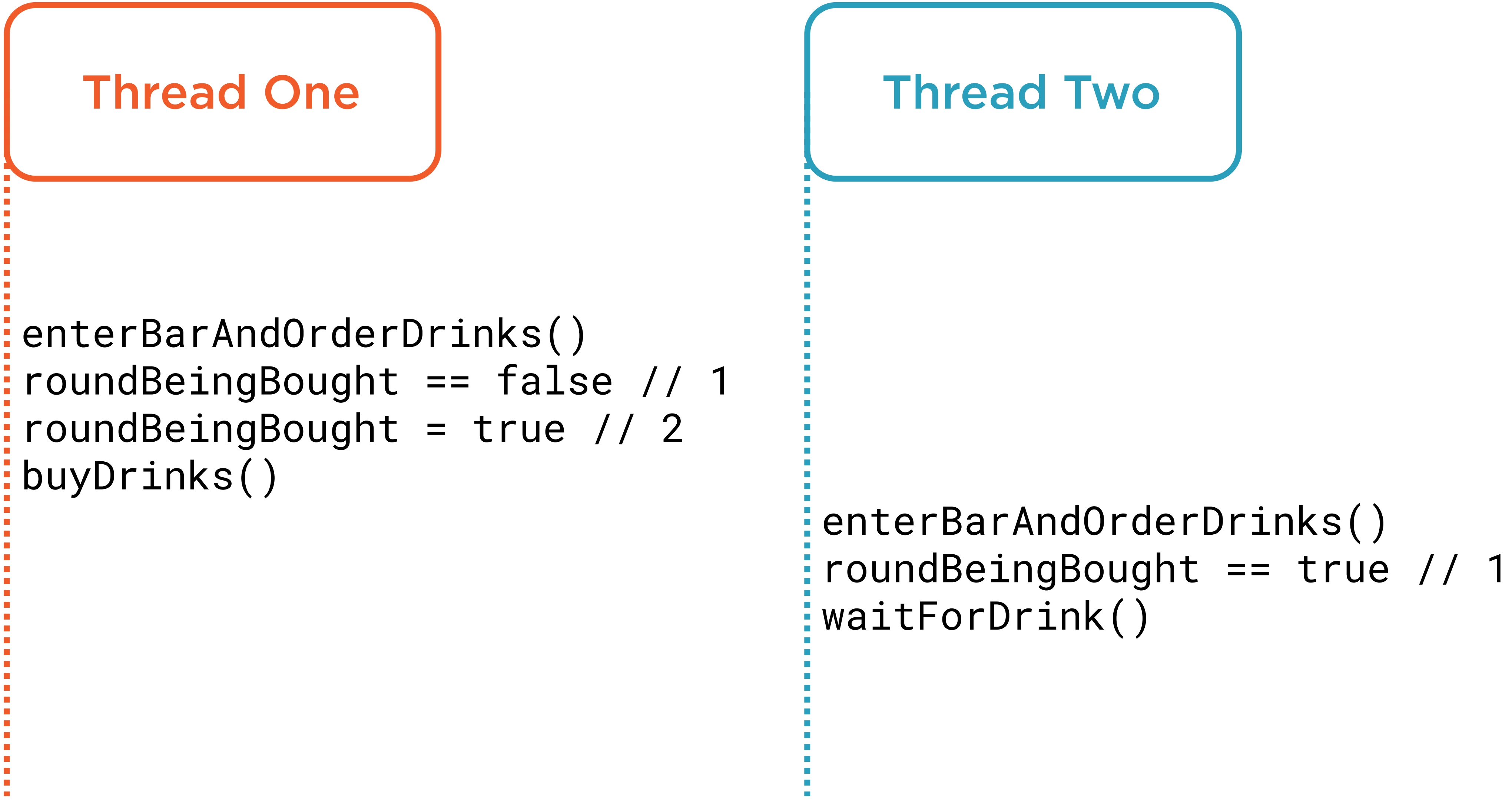
Allows one thread to access a critical section at once

Built in 'synchronized' gives mutual exclusion

Prevents concurrent access to resources

A Joke...

Two threads walk into a bar...



Thread One

```
enterBarAndOrderDrinks()
roundBeingBought == false // 1
roundBeingBought = true // 2
buyDrinks()
```

Thread Two

```
enterBarAndOrderDrinks()
roundBeingBought == true // 1
waitForDrink()
```

Race Conditions

Incorrect behaviour caused by threads interleaving and executing code in an unintended order

Code usually works but occasionally goes wrong

- Code is not thread safe

Critical Section

One or more parts of the code which may not be accessed by more than one thread at a time

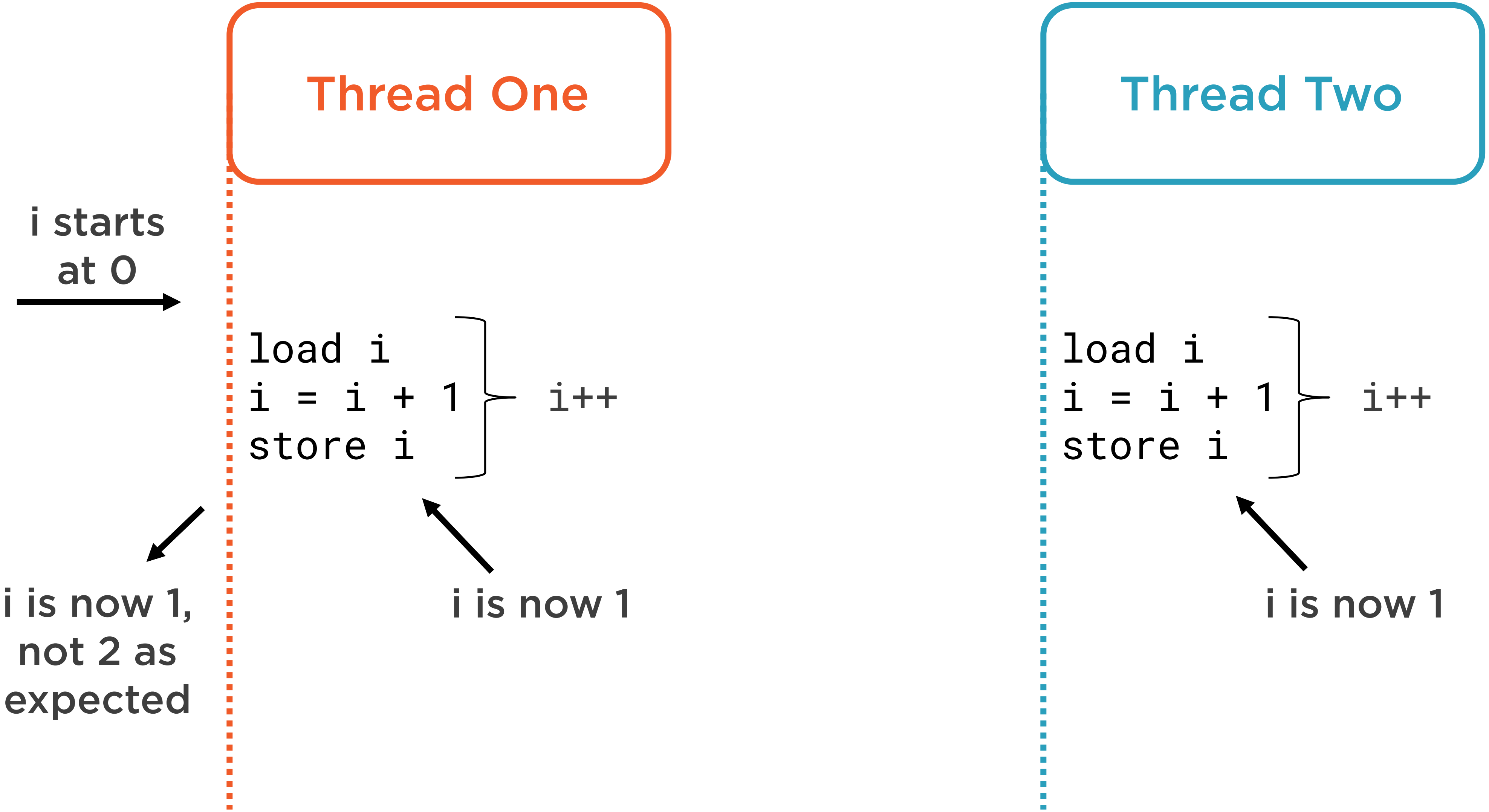
Critical sections may span more than one part of the code

Several critical sections may exist in the same program

Increment and Decrement

Not atomic if can be divided into smaller pieces

Increment is not atomic: Load, add, store



Thread One

Thread Two

i starts at 0

```
load i
i = i + 1
store i
```

i++

i is now 1

```
load i
i = i + 1
store i
```

i++

i is now 1

i is now 1, not 2 as expected

AtomicInteger

Provides safe pre and post increment and decrement

Cannot be divided into smaller parts

- So no interleaving

```
// static volatile int counter;
static final AtomicInteger counter = new AtomicInteger();

public void incrementAndDecrement()
{
  counter.getAndIncrement(); // instead of counter++;
  counter.incrementAndGet(); // instead of ++counter;
  counter.getAndDecrement(); // instead of counter--;
  counter.decrementAndGet(); // instead of --counter;
}
```

Mutex

Short for mutual exclusion object

Prevents mutual access like a key to a bathroom

- Others wanting access have to wait

Demonstrating Race Conditions

Problems occur occasionally

- Impractical to manually run over and over

Could try:

- Putting code in a loop
- Using sleeps
- Using a CountDownLatch

```
if (!roundBeingBought) // 1
{
    roundBeingBought = true; // 2
    buyDrinks();
}
else
{
    waitForDrink();
}
```

Critical Section

Obtaining Mutual Exclusion

Use synchronized keyword

Implemented with monitor locks (aka intrinsic locks)

Implementation in Java

Objects are associated with intrinsic lock (a monitor)

Threads entering the monitor are put in the entry set

If no thread owns the monitor, one thread is chosen from the set

Thread acquires monitor and enters the critical section

Implementation in Java

If another thread owns the monitor, thread moves to Blocked state

Thread leaves gives up ownership releasing the monitor

Thread then exits the monitor

Another thread is chosen from the entry set moving from Blocked to Runnable

Selection from Entry Set

No guarantee which thread chosen

Possible under load that some threads may never get chosen

- No guarantee of fairness

Knowing if a Lock Is Being Held

We don’t actually get a mutex object

- Can check Thread.holdsLock(object) to check if we own the monitor

Reentrancy

Monitors are reentrant

- Means we can acquire one if we own it without blocking
- Blocking would cause the thread to lock up (deadlock)
- Problem with some non-Java implementations

Probably should only check holdsLock to debug

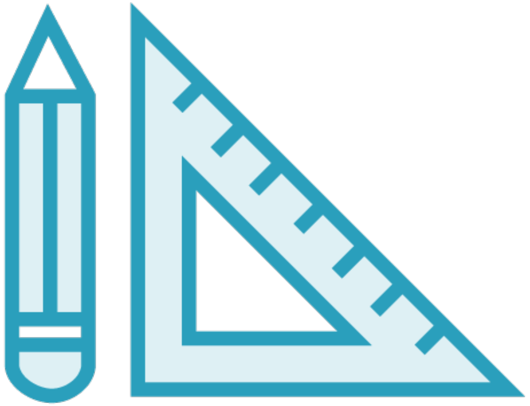
Monitors and Objects

Monitor is associated with the object, not the reference

Should prevent reference changing (use final)

Same object for all threads that share critical section

Creating an Object to Use for Synchronization



```
private static final Object mutex = new Object();
```

Object is smallest to create so we're using that

Hide from other parts of the code

Should synchronize on different objects for different critical sections

- Using Boolean is bad as there are only two instances

```
synchronized(mutex)
{
    if (!roundBeingBought) // 1
    {
        roundBeingBought = true;
    }
}

// did we set the flag or not and thus should buy drinks?
```

```
boolean shouldBuyDrinks = false;

synchronized(mutex)
{
   if (!roundBeingBought) // 1
   {
      roundBeingBought = true; // 2
      shouldBuyDrinks = true;
   }
}
// now just need to test shouldBuyDrinks
```

AtomicBoolean

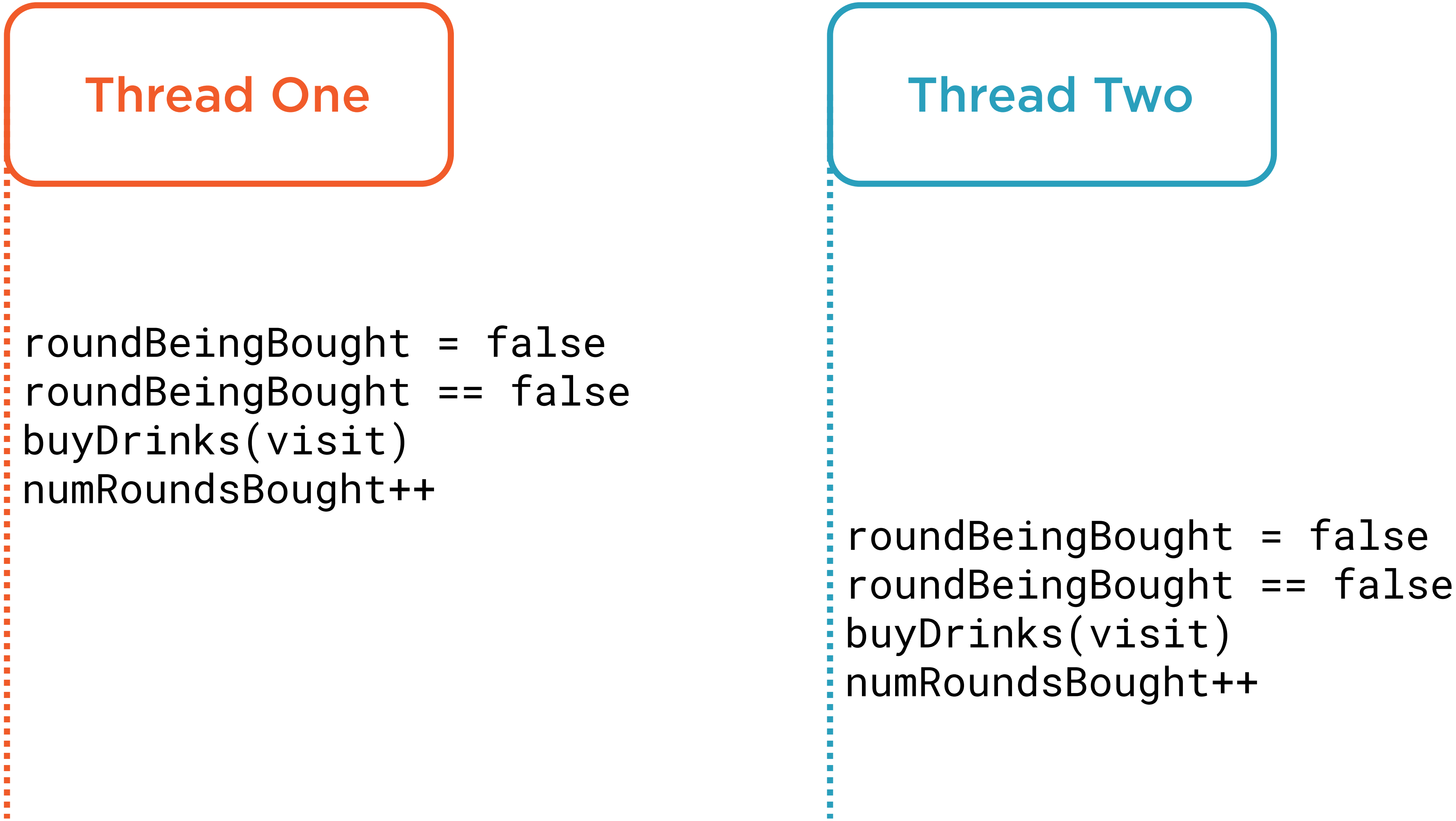
Using mutual exclusion for 'test-and-set' heavyweight

AtomicBoolean is better for implementing 'test-and-set'

Use method: compareAndSet(boolean expect, boolean update)

- If the value is ‘expect’, it is set to ‘update’
- Returns true if the set took place

```
private static final AtomicBoolean roundBeingBought =
                                      new AtomicBoolean();
...
if (roundBeingBought.compareAndSet(false, true)) {
   buyDrinks();
} else {
   waitForDrink();
}
```



Thread One

```
roundBeingBought = false
roundBeingBought == false
buyDrinks(visit)
numRoundsBought++
```

Thread Two

```
roundBeingBought = false
roundBeingBought == false
buyDrinks(visit)
numRoundsBought++
```

CountDownLatch

Only good for a single use

- Need to create when needed
- Or
- In advance

CyclicBarrier

Like a reusable CountDownLatch

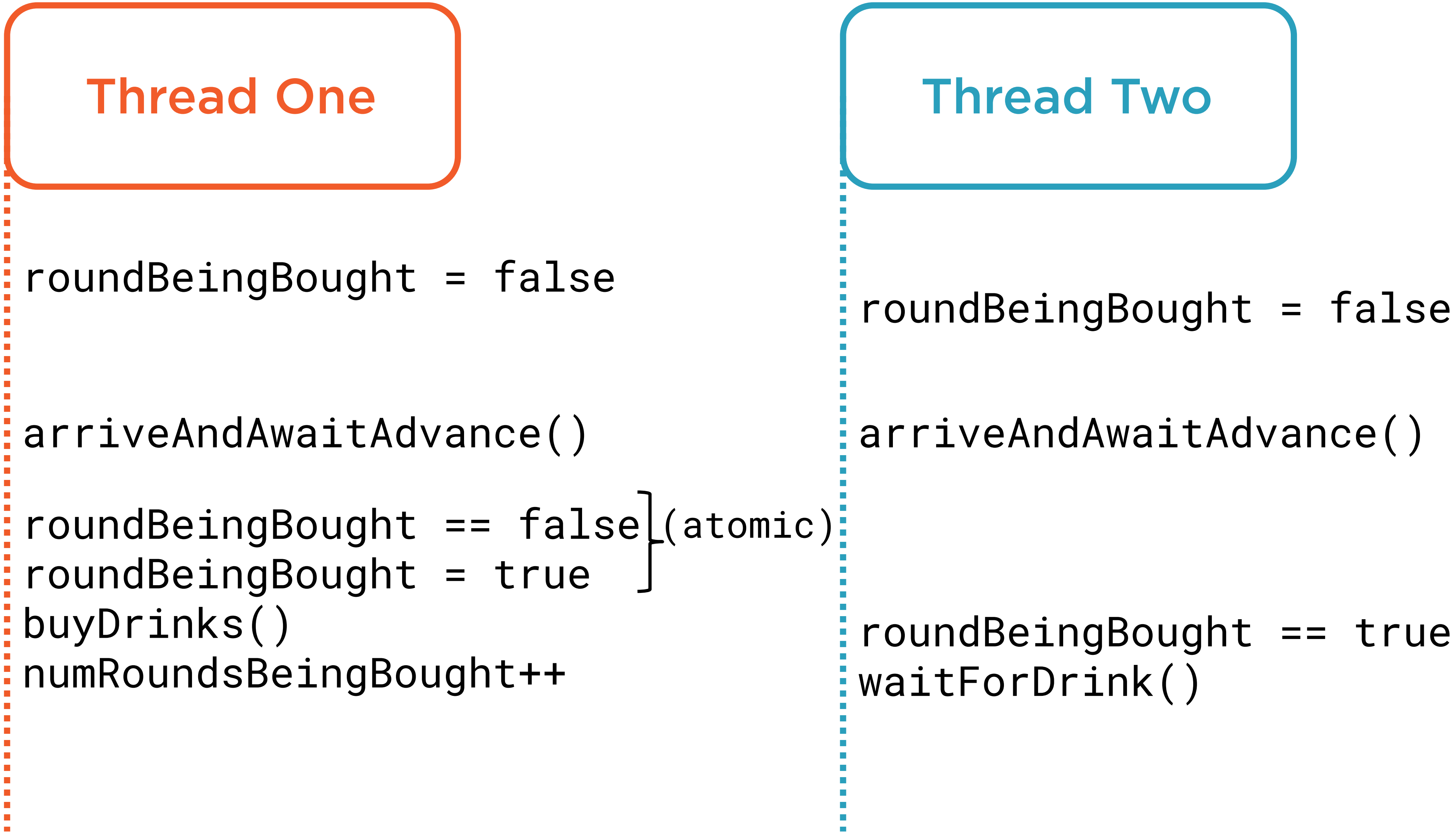
- But needs to be reset

Phaser

CyclicBarrier with multiple phases [avoids needing to be reset]

When all threads reach the barrier

- Phase is incremented
- Threads may proceed

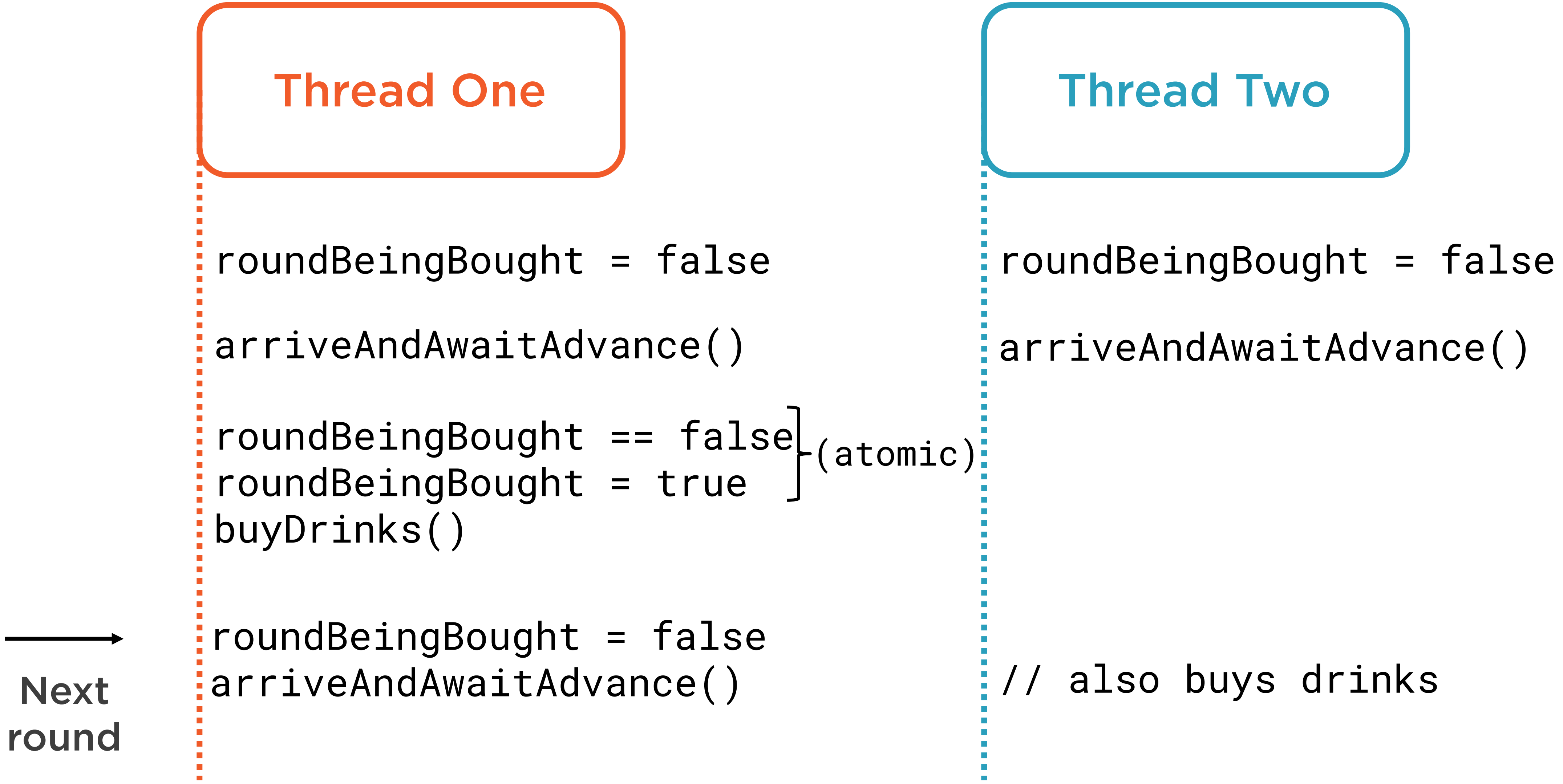


Thread One

```
roundBeingBought = false

arriveAndAwaitAdvance()

roundBeingBought == false  ⎤ (atomic)
roundBeingBought = true    ⎦
buyDrinks()
numRoundsBeingBought++
```

Thread Two

```
roundBeingBought = false

arriveAndAwaitAdvance()

roundBeingBought == true
waitForDrink()
```



Thread One

```
roundBeingBought = false

arriveAndAwaitAdvance()

roundBeingBought == false
roundBeingBought = true      (atomic)
buyDrinks()

roundBeingBought = false
arriveAndAwaitAdvance()
```

Thread Two

```
roundBeingBought = false

arriveAndAwaitAdvance()





// also buys drinks
```

Next round

Synchronized Using the Same Object

If using same object on multiple blocks

- Only one thread can execute in any of them at once
- They become part of the same critical section

Synchronized Keyword on Method Definitions

For instance - methods, object that 'this' refers to will be synchronized on

- All object's synchronized methods become a single critical section
- Controlled by the same monitor

Benefits:

- Visible
- Documentable
- No statements can be inserted before

Method Signatures and Synchronized

Synchronized does not form part of the method signature

Two forms (one synchronized, one not) not possible. Instead:

- Two versions of the class (e.g. StringBuffer & StringBuilder)
- Two differently named or parameterized methods

Synchronized version can call non-synchronized version

Hashtable

Thread safe HashMap with synchronized methods

- Doesn't actually call HashMap

Get and put can't be used simultaneously

- Get doesn't modify
- Synchronized prevents multiple readers
- May cause performance issues
- ConcurrentHashMap may give better performance

Static Methods and Synchronized

Synchronized on an instance method doesn't synchronize across all instances

- Problem if static variables are modified

Can use synchronized on static methods

- Class object (E.g. Foo.class) is passed to synchronized

Summary

A race condition is caused by threads executing code in an unintended order

- The code where this happens is a critical section

Mutexes protect critical sections

- Obtaining a mutex means only that thread can enter the critical section
- Other threads must wait until it is released

Summary

Java uses the synchronized keyword which is passed an object

- The block becomes a critical section
- Implemented by monitors

Use mutual exclusion to:

- Protect invariants
- Keep object fields consistent
- Prevent simultaneous reads and writes

Summary

AtomicInteger

- Atomic increment and decrement

AtomicBoolean

- ‘test-and-set’

Saw Hashtable to prevent concurrent access to a collection

Summary

Synchronized on a code block

- Object passed should be final and not shared elsewhere

Synchronized on a method

- Instance method uses object 'this' refers to
- Static method uses class objects

Summary

Java mutexes are reentrant

- Try to obtain one we're holding, we won't have to block

And now you should be able to answer the following riddle:

How many threads does it take to change a lightbulb?

How many threads does it take to change a lightbulb?

All of them… if you don’t use a mutex!