

Understanding CASing and Atomic Variables



José Paumard

PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard <https://github.com/JosePaumard>



Agenda



CASing!

What does compare and swap mean?

And why is it useful?

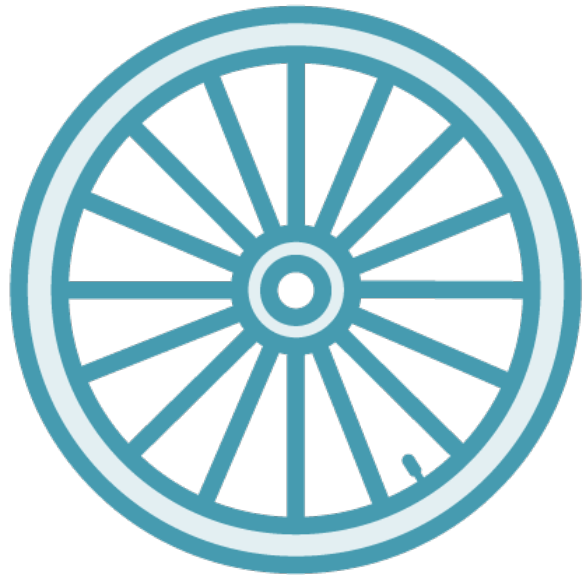
What is in the JDK to implement CASing?

✓ How and when to use it



✓ CASing = “Compare And Swap”





The starting point is a set of assembly instructions

Very low level functionalities given by the CPU

That are exposed at the API level so that we can use them in our applications

✓ The starting point is a set of assembly instructions that is very low level functionalities given by the CPU. Those low level functionalities have been exposed at the API level in the JDK on the many other languages so that we can leverage them in our applications.

What is CASing?



Concurrent Read / Write

The problem in concurrent programming is the concurrent access to shared memory

We used synchronization to handle that

But in certain cases, we have more tools



Concurrent Read / Write

Synchronization has a cost...

Is it really always essential to use it?

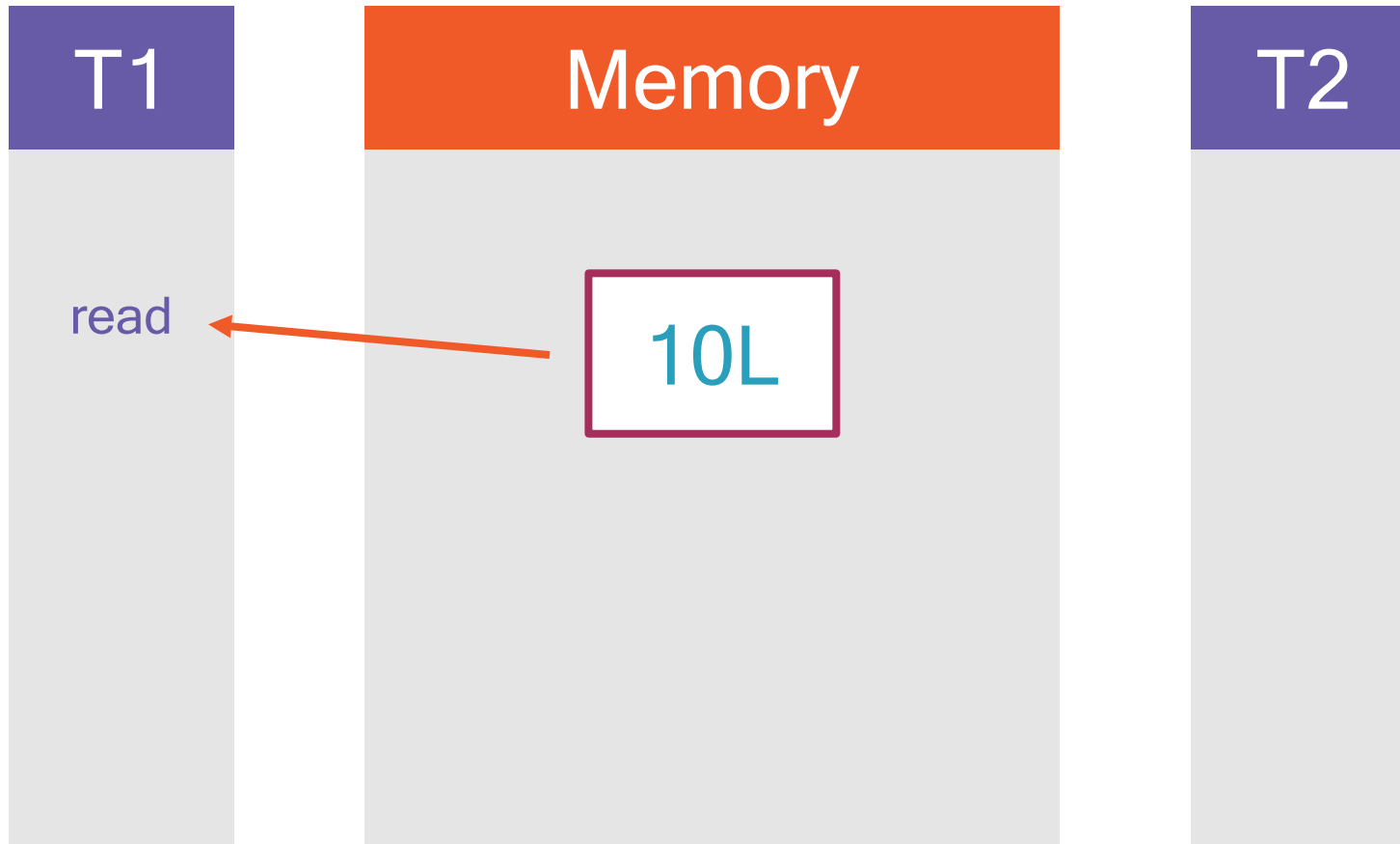
In fact we use it to be sure that our code is correct

What if in fact, real concurrency is rare?

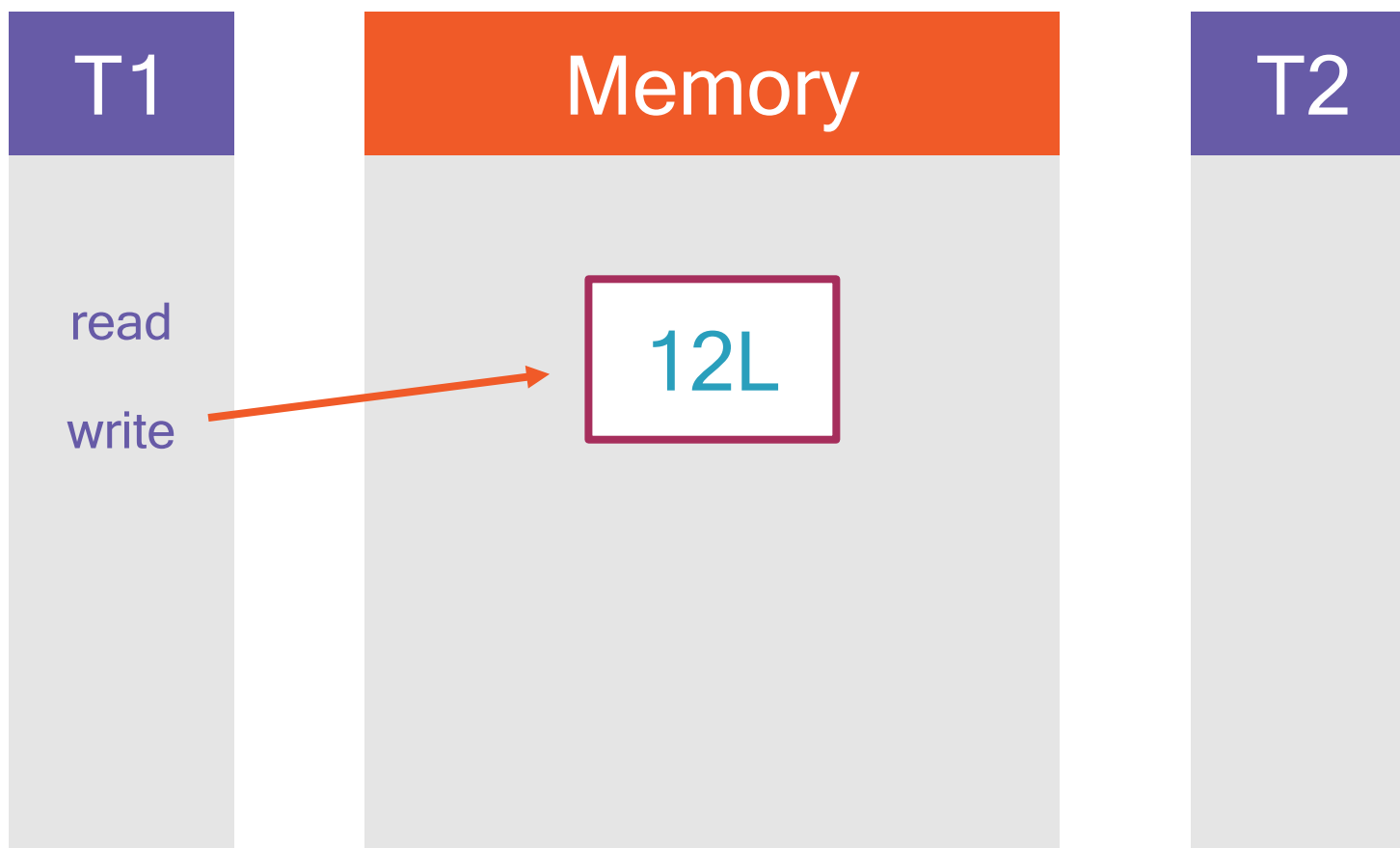
Synchronization has a cost. In fact, we use it to be sure that our code is correct. But if we didn't use it, are we really sure that our code would fail. In fact, there are many cases where people forget to synchronize the modification of memory and the code still works. What it means is that there are many cases where real concurrency is rare.



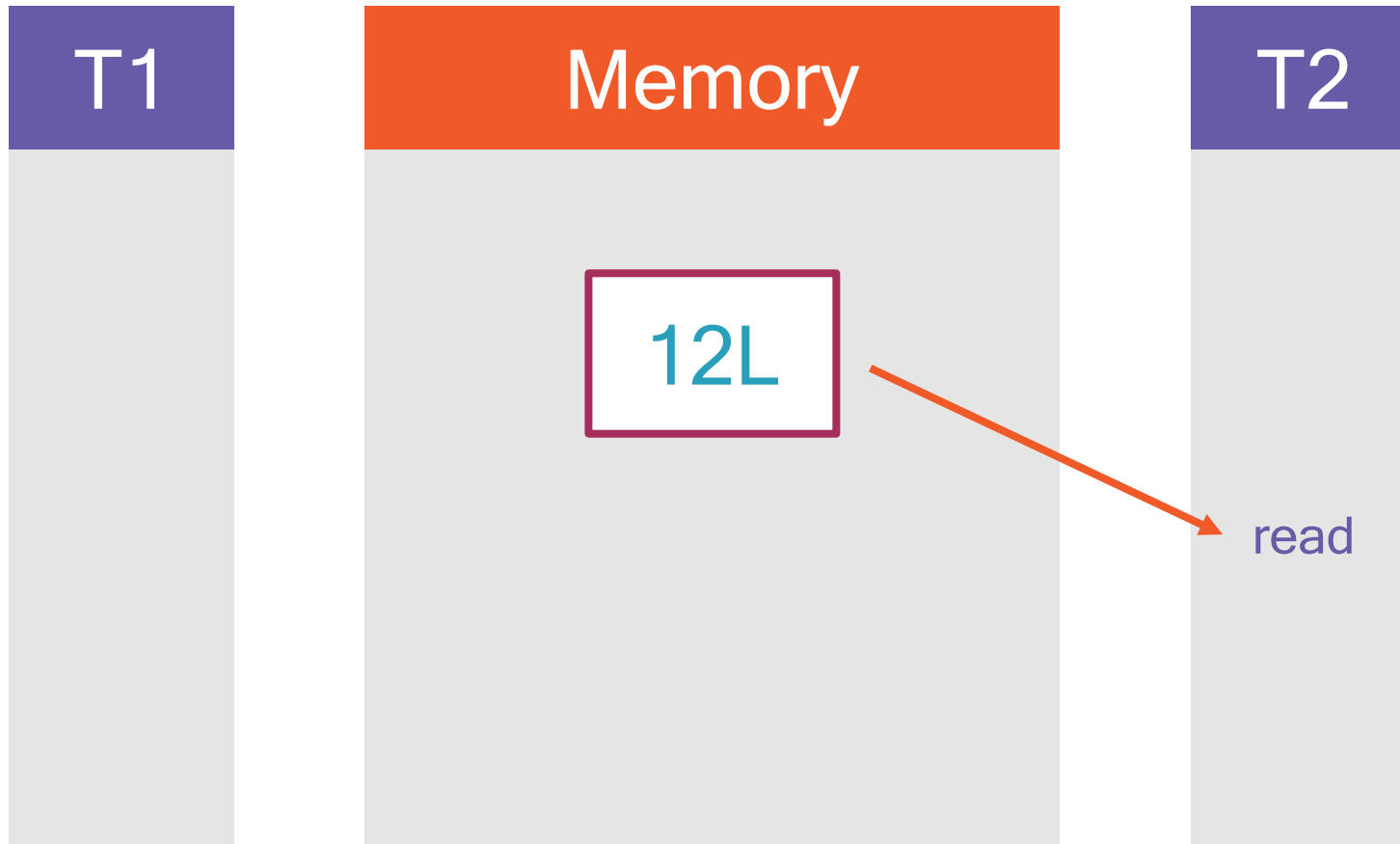
Synchronized Memory



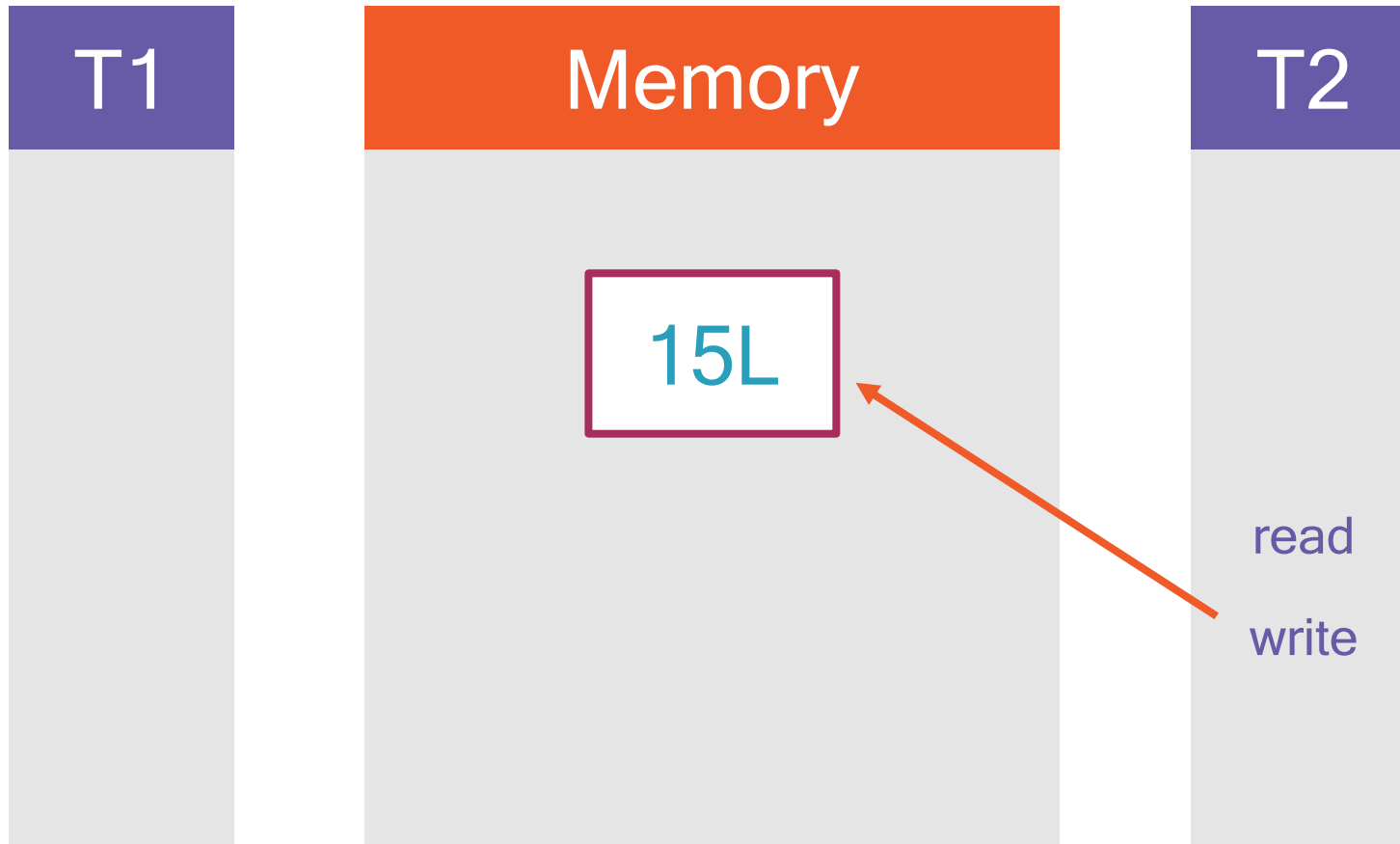
Synchronized Memory



Synchronized Memory



Synchronized Memory



Concurrent Read / Write

We need to write a correct code, so protection by lock is essential

But in fact, there is no real concurrency at runtime...

This is where CASing can be used



CASing

Compare and Swap works with three parameters:

- ✓ a location in memory
- ✓ - an existing value at that location
- ✓ - a new value to replace this existing value



CASing

If the current value at that address is the expected value, then it is replaced by the new value and returns true

If not, it returns false

All in a single, atomic assembly instruction



```
// Create an atomic long
AtomicLong counter = new AtomicLong(10L);

// Safely increment the value
long newValue = counter.incrementAndGet();
```

Example with AtomicLong

Safe incrementation of a counter without synchronization



Under the Hood

The Java API tries to apply the incrementation

The CASing tells the calling code if the incrementation failed

✓ If it did, the API tries again Imp

CASing works well when concurrency is not too high. In fact, if the concurrency is high, then the update operation of the memory will be tried again and again until it is accepted by all the thread, and at one given point of time, only one thread will win. All the other ones will be retrying again and again. That's why The behavior of a CASing system is very different from the behavior of a synchronized system.



If you synchronize a portion of memory, it means that all your threads between are going to wait to access this memory. In the case of the CASing, all the threads at the same time are going to access this memory, but only one will be the winner. So if CASing is not used in the right use case, it may create a very heavy load both on the memory and on the CPU.



Let us browse through the API

We have several classes

With different functionalities...





Atomically sets to the given value and returns the previous value.

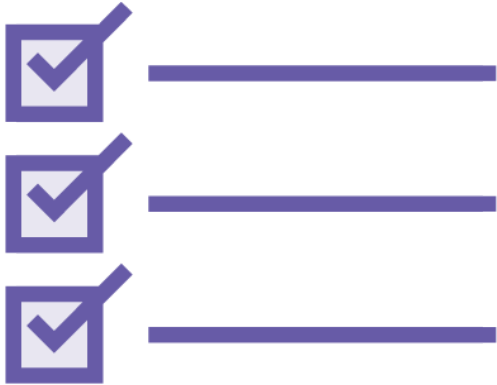
We have:

AtomicBoolean

- **get(), set()** → Unconditionally sets to the given value.
- **getAndSet(value)**
- **compareAndSet(expected, value)**

Atomically sets the value to the given updated value if the current value == the expected value.





We have:

AtomicInteger, AtomicLong

- get(), set()
- getAndSet(value)
- compareAndSet(expected, value)
- getAndUpdate(unaryOp),
updateAndGet(unaryOp)





We have:

`AtomicInteger`, `AtomicLong`

- `getAndIncrement()`, `getAndDecrement()`
- `getAndAdd(value)`, `addAndGet(value)`
- `getAndAccumulate(value, binOp)`,
`accumulateAndGet(value, binOp)`





We have:

`AtomicReference<V>`

- `get(), set()`
- `getAndSet(value)`
- `getAndUpdate(unaryOp),
updateAndGet(unaryOp)`
- `getAndAccumulate(value, binOp),
accumulateAndGet(value, binOp)`
- `compareAndSet(expected, value)`

About CASing

CASing works well when concurrency is not “too” high

CASing: many tries until it is accepted...

Synchronization: waiting threads until one can enter the synchronized block

✓ CASing may create load on the memory and / or CPU

group



Atomic Variables

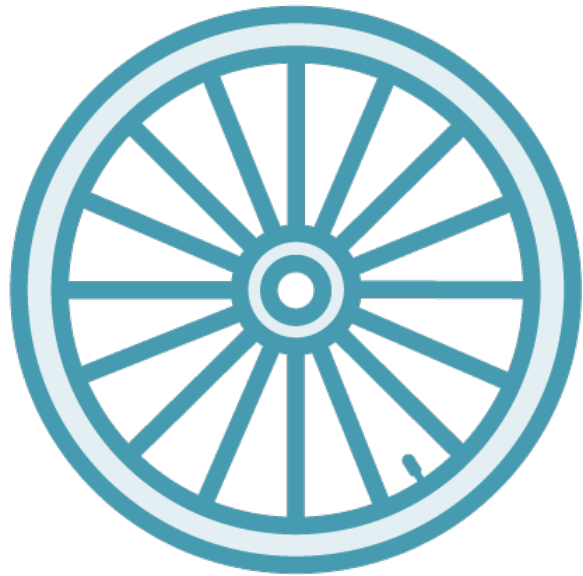
CASing is another tool to handle concurrent reads and writes

It is different from synchronization

- ✓ Can lead to better performances
- ✓ It should be used with care!

Adders and Accumulators





All the methods are built on the “modify and get” or “get and modify”

Sometimes we do not need the “get” part at each modification

Thus the LongAdder and LongAccumulator classes (Java 8)



LongAdder & LongAccumulator

✓ No get() but merge()

It work as an AtomicLong

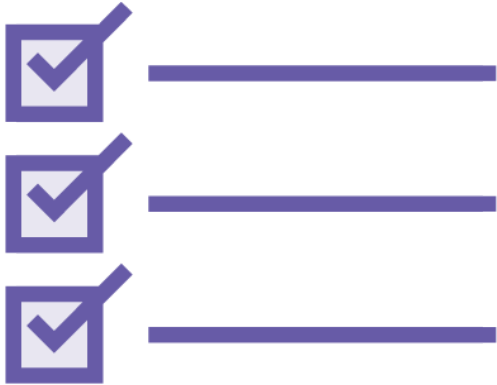
It does not return the updated value

So it can distribute the update on different cells

And merge the results on a get call

These are tailored for high concurrency





For the LongAdder:

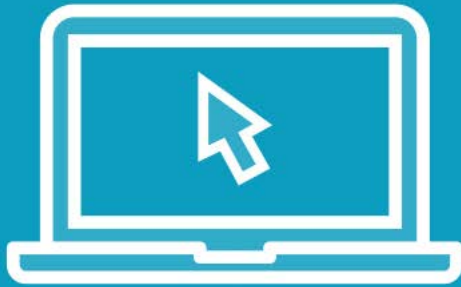
- increment(), decrement()
- add(long)
- sum(), longValue(), intValue()
- sumThenReset()



For the LongAccumulator:

- built on a binary operator
- accumulate(long)
- get()
- intValue(), longValue(), floatValue(), doubleValue()
- getThenReset()

Demo



Let us see some code!

Let us see some atomic counters in action



Demo Wrapup



What did we see?

How to create an atomic counter in a thread safe way without synchronization

Retrying is normal when using atomic operations, and should be expected



Module Wrapup



What did we learn?

When we need to update values or references in memory, CASing may be a better solution than locking

We have several tools with common operations

We still need to choose the right one, depending on the level of concurrency

