LLD: Concurrency 3

Locks and Atomic Data Types

(We will start at 9:10PM)

BMS

RWLock

Concurrent HashMap

Distributed Cache

Agenda

Synchronized

Object → Only one thread will be allowed

methods → Across all Sync methods only 1 thread will come

① Cons of Synchronized ⇒ Perf issues

② Locks → Reentrant Lock
→ Read Write Lock
→ Semaphores

③ Volatile

④ Concurrent Data Structures

⇒ Atomic Integer

⇒ Concurrent Hash Map

Sync

Sync

Sync

Sync

subtractj (int x){}

addi (int x) { }

subtract i (int x){}

(1) Performance impact

(2) Third Party Class if it doesn't ~~support~~ concurrency, if we use it handle in a multithreaded environment we will have problems

Locks

Class A {

sync doA()

sync doB()

sync doC()

}

if one thread was trying to access a sync method, other threads are locked (prevented) from

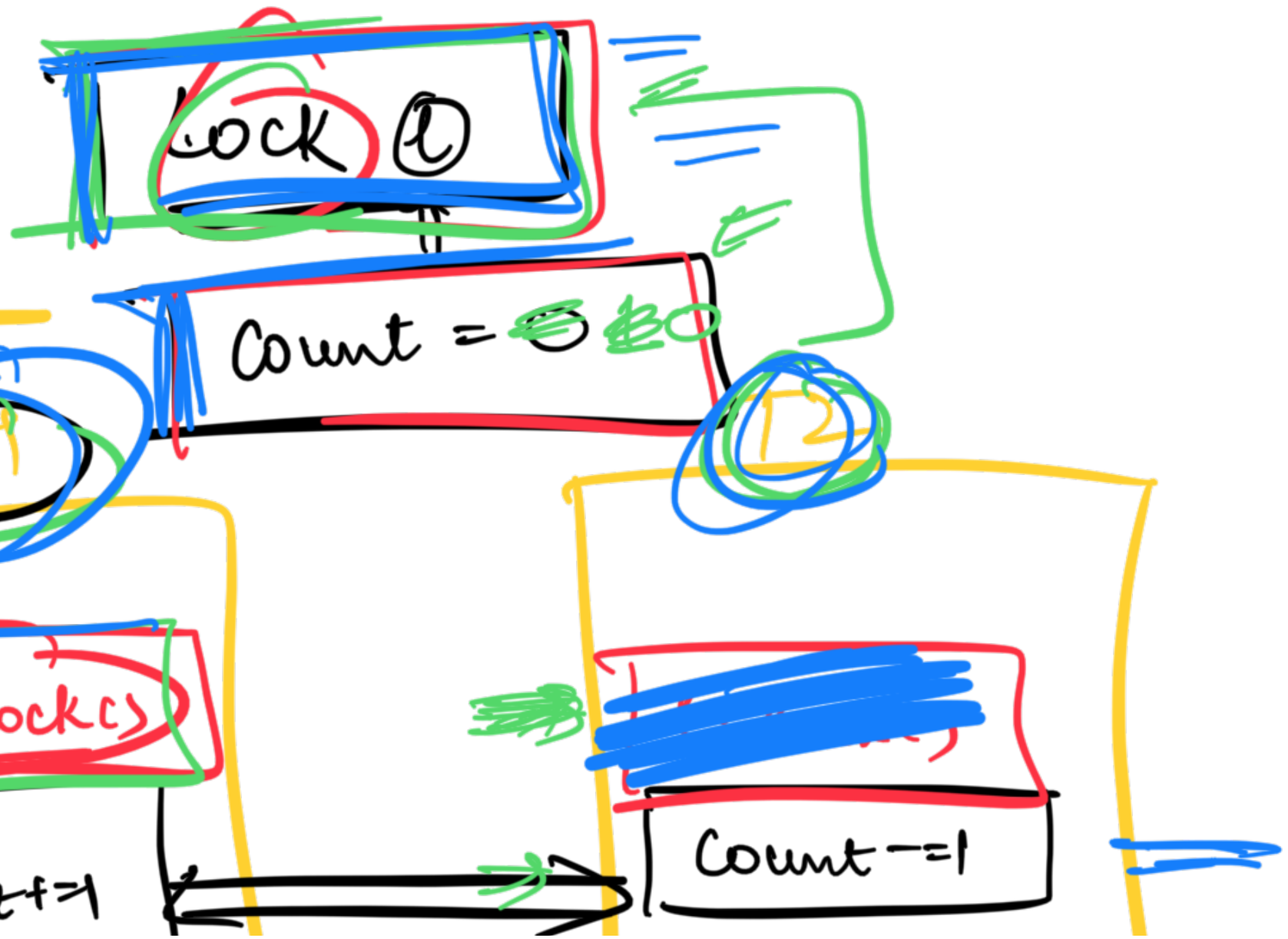
// sync is not being handled

by the class

⇒ that will be handled at client side.

accessing other sync methods.

Lock interface

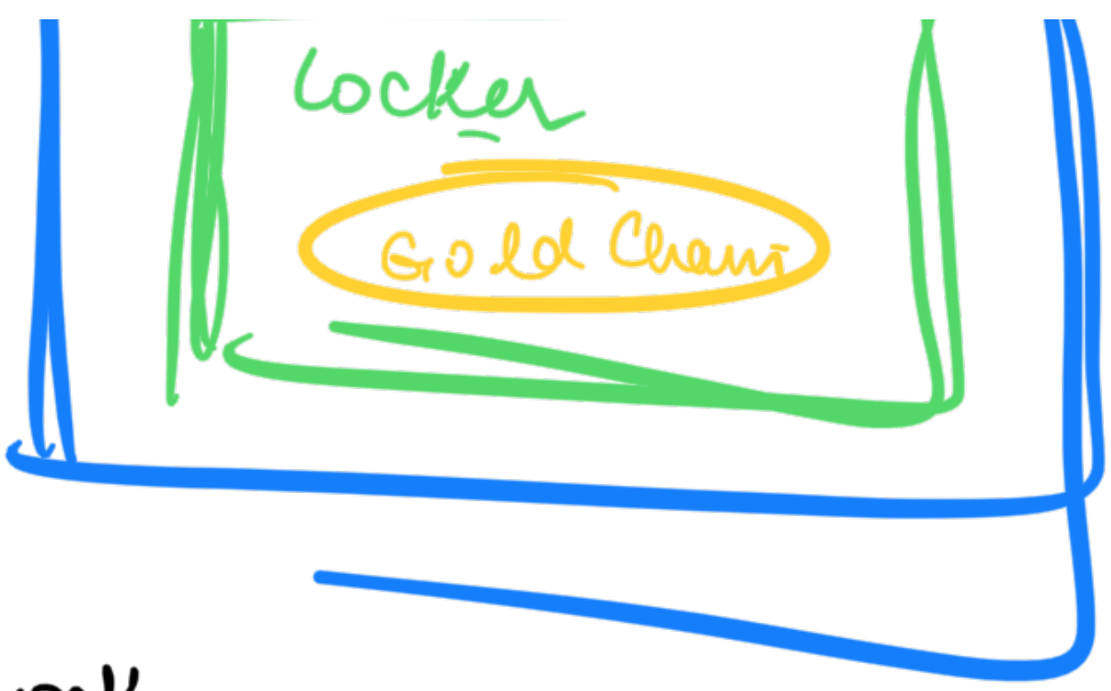


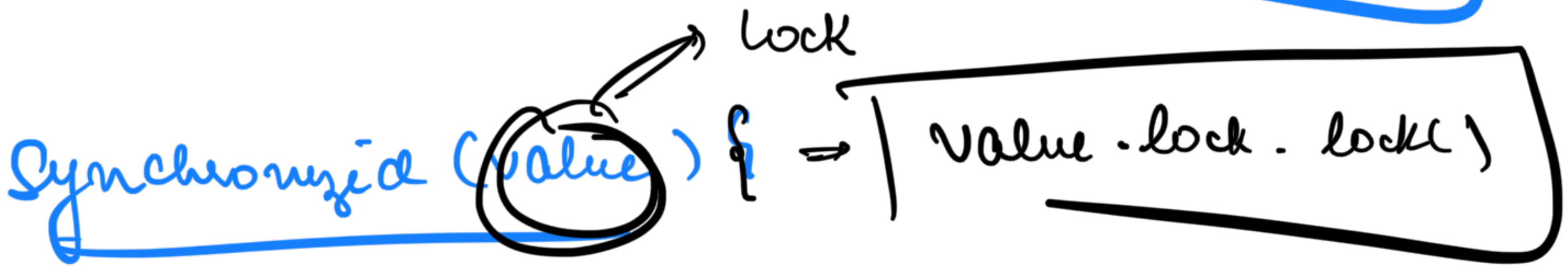
l.unlock()

internally within a lock object, language has

a waiting queue

Put our own lock


locker
Gold Chain


lock
synchronized (value) {
value.lock.lock()

value - lock . unlock()

Lock Two locks

A

B

T1

Synchronized (A) {

lock for A and B locally

Synchronized (B) {

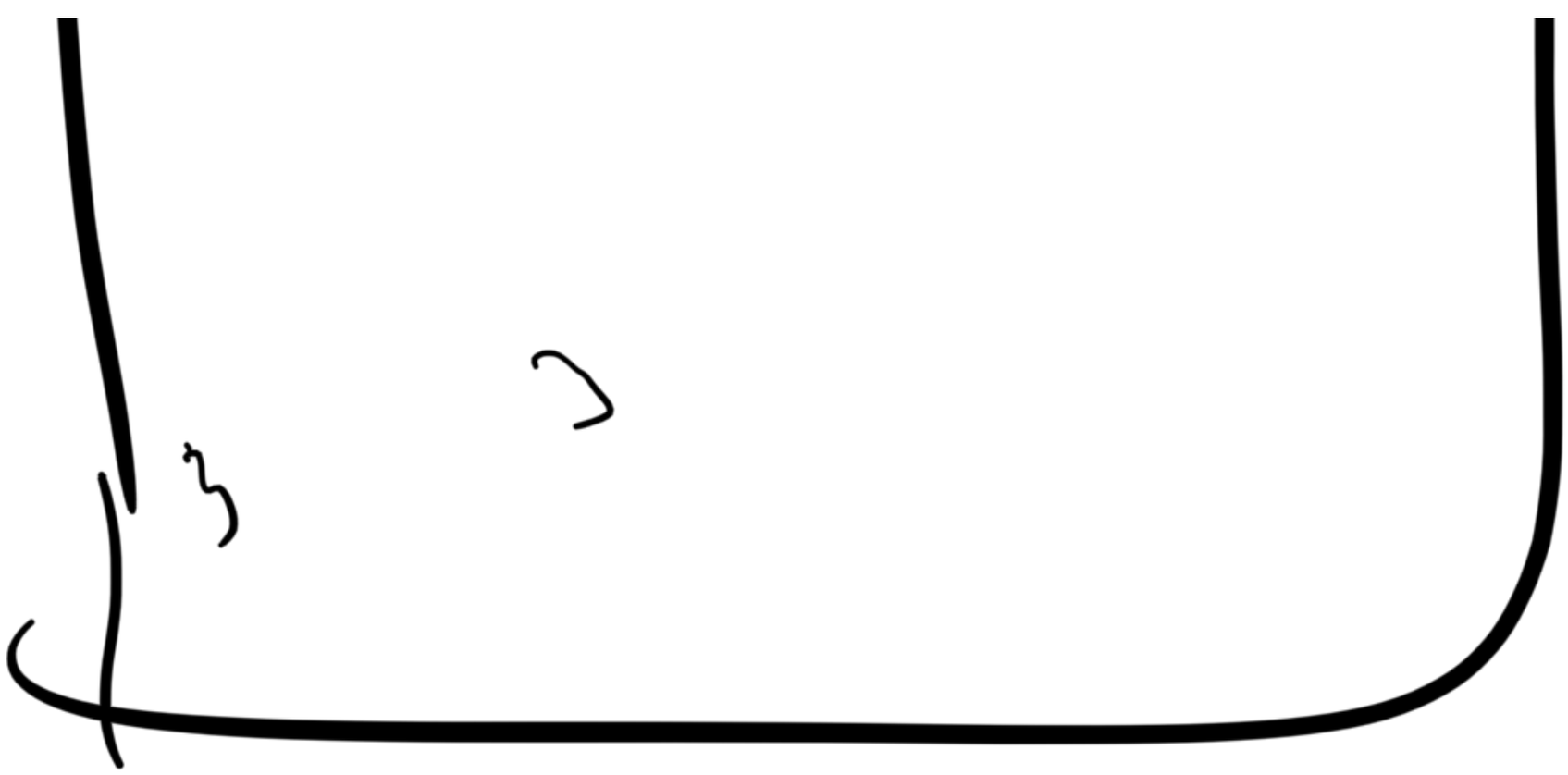
}

look for A and B. unlock

T2

Synchronize (B) {

Synchronize (A) {



Use Case :

2 objects A and B

I always access A and B together.

Lock Anything's

Lock roman

whenever I try to access A and B

I will just take a lock on

anything

anything.lock()

getA()

getBC)

()

getB()

getA()

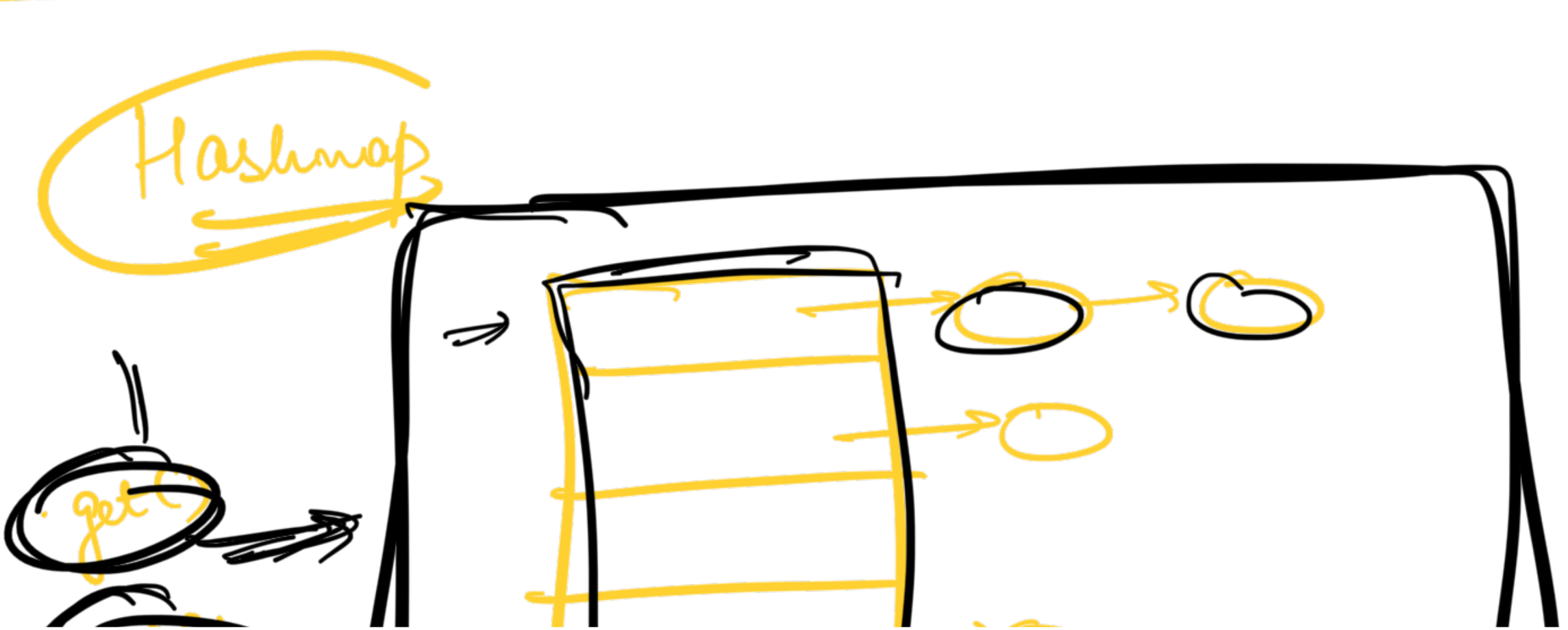
anything.unlock()

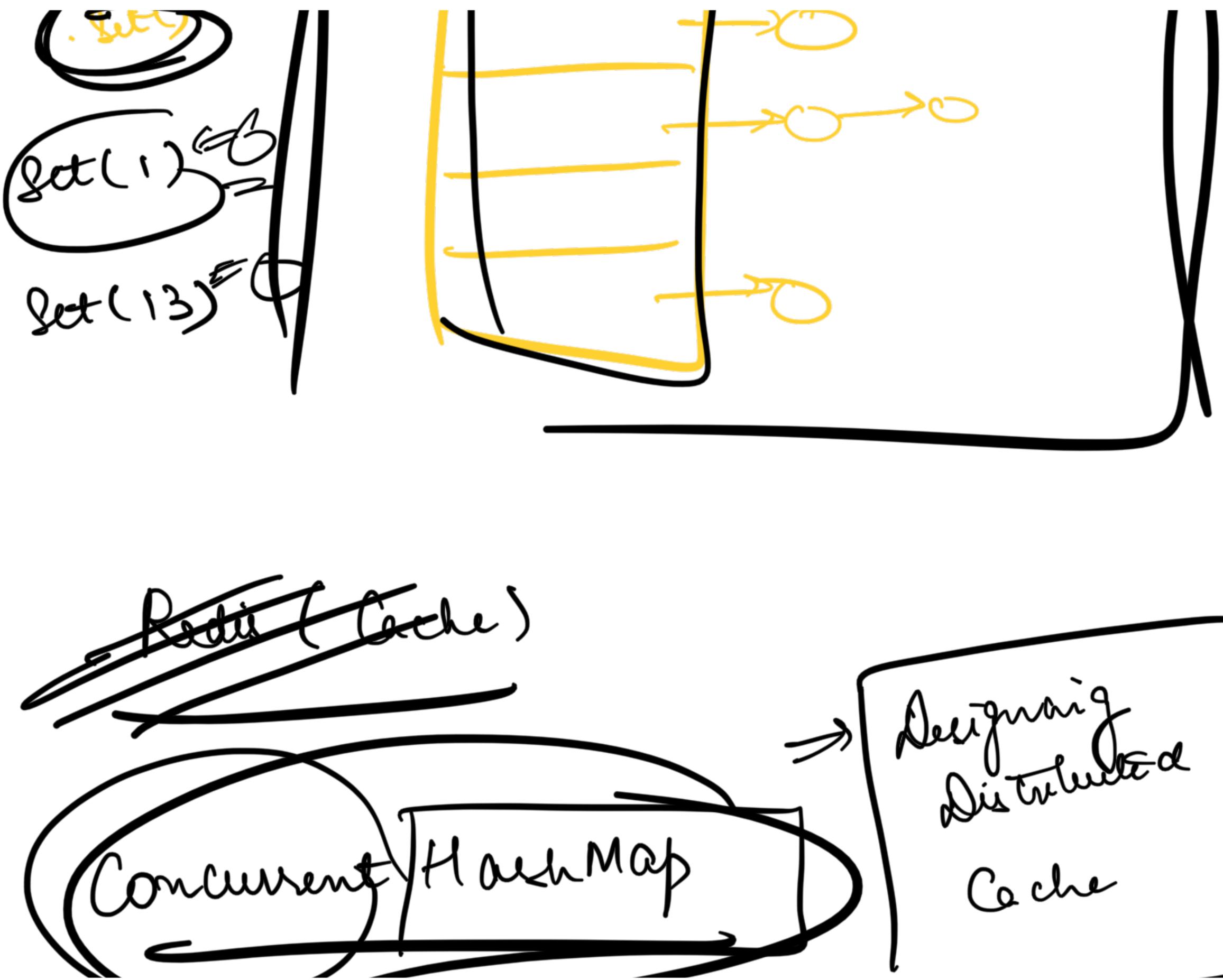
Why did we need to handle Concurrency?

Getting valu

1

Atomic Data Types





Redis (Cache)

Concurrent HashMap

Designing Distributed Cache

1 - 50

①

51 - 100

②

>100

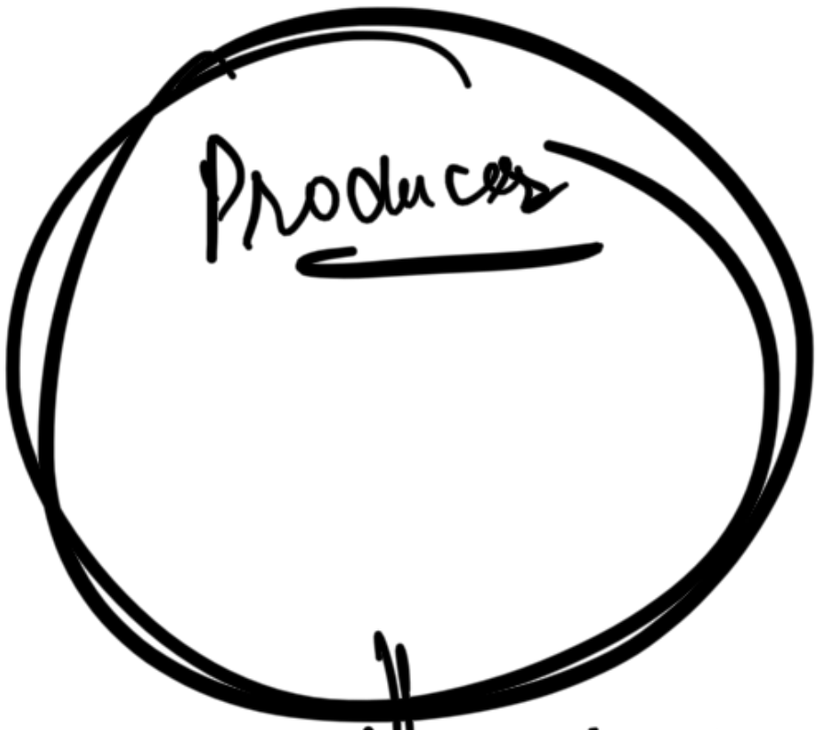
③

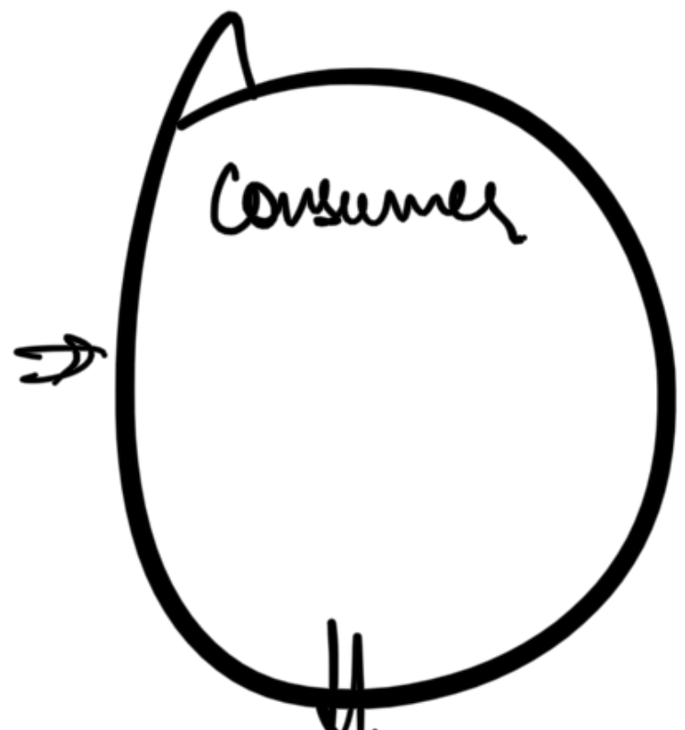
map. let (51, "hello")

Million rows

map. set (57, "hi")

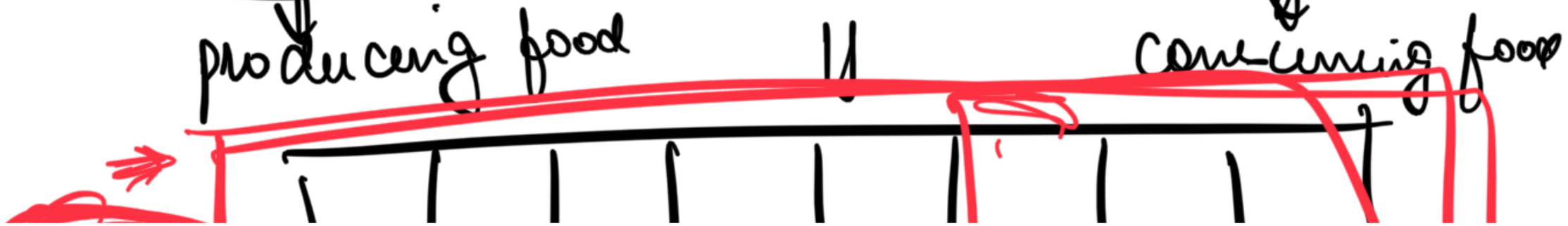
map. set (1, "hey")





producing food

consuming food



Block with a given count

Count

Semaphore s = new Semaphore ( )

at Max ⇒ s. acquire ( ) // lock

Count

s release ( ) // unlock.

threads

can

access

at

some

time

Problem Statements

① 2 types of people

a.) Producers

(Each producer is a separate thread)

b) Consumers

(Each consumer is a separate thread)

② We should not allow a producer to put something on tray if tray is full

③ We should not allow a consumer to eat something from tray if tray is empty