Agenda

1) Synchronised keyword

2) Semaphores

⇒ SYNCHRONISED KEYWORD :-

Synchronised keyword helps us to solve synchronisation issues.

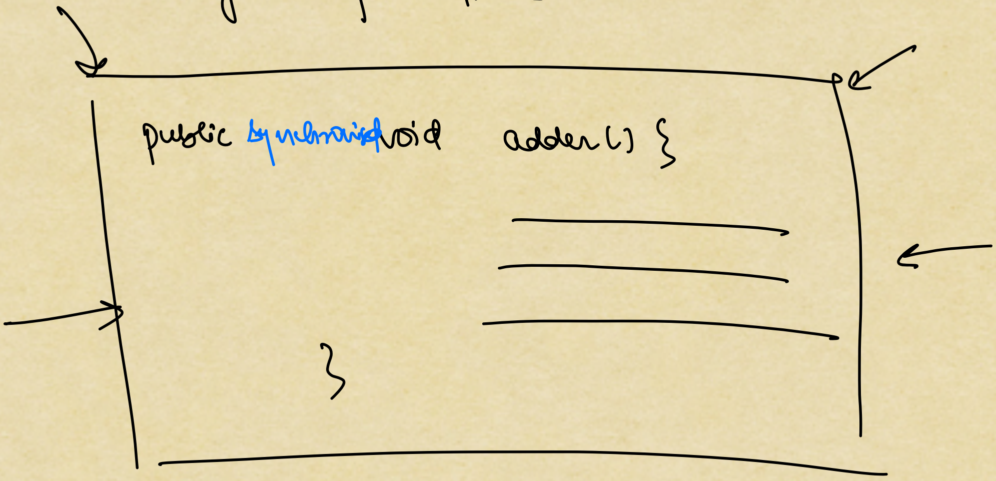
There are 2 ways :-

1) Synchronised method

2) Synchronised block

1) Synchronised method

We put synchronised keyword in the method name, so that the method is not accessible by multiple threads at the same time.



```
public synchronised void adder() {
        ________
        ________
        ________
    }
```

ii) <u>Synchronised block</u>

We put a block of code inside "synchronised", to make the code accessible only by a single thread at a time.

```
public void adder() {
        ____________
         ____________
         ____________
    synchronised(______) {
         data.add();
         ____________
    }
         ____________
}
```

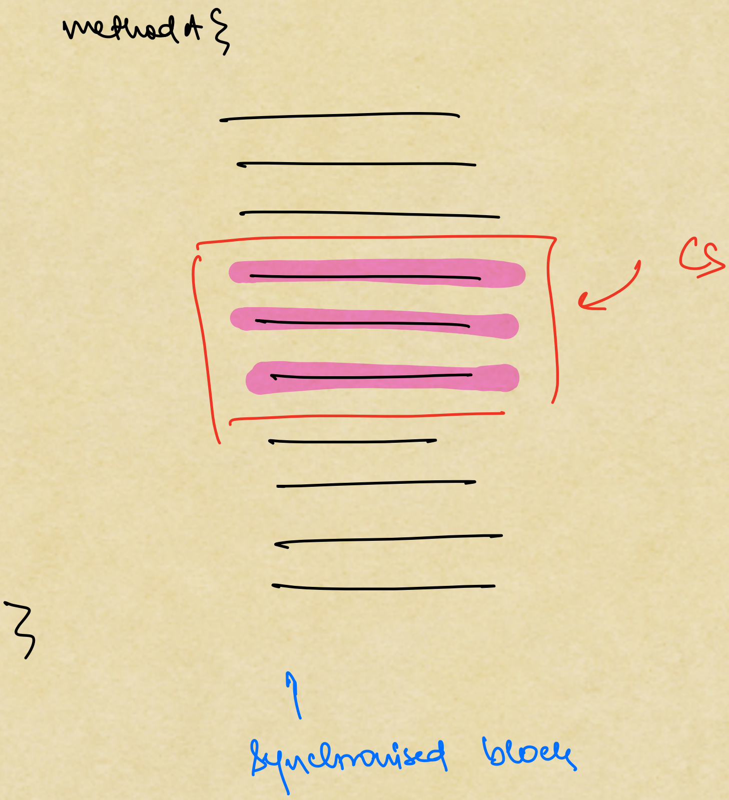
→ object name | classname
(this)   (Adder.class)

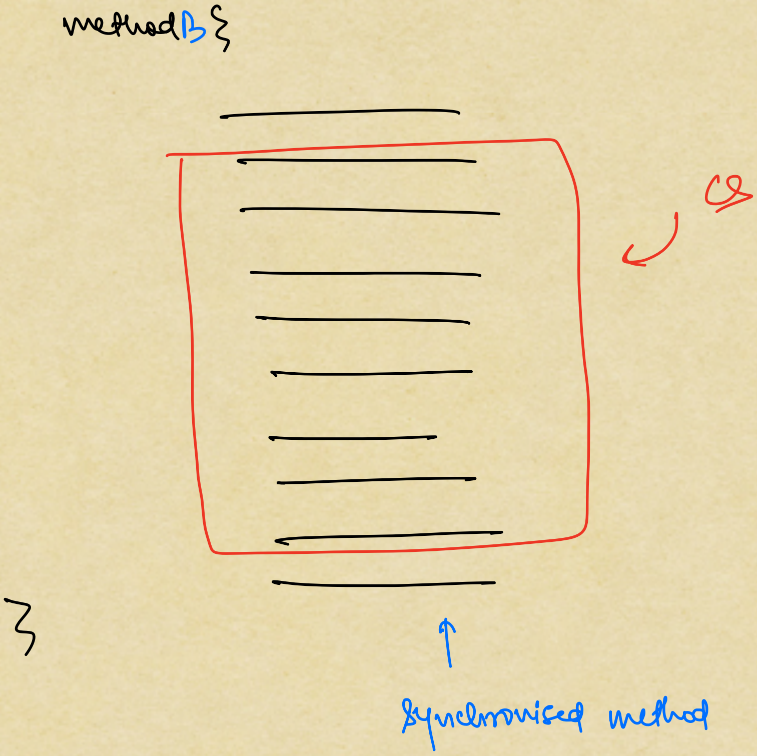
→ Use case wise both are same,

→ Synchronised method puts the entire method under lock, we cant optimise on lines of code

→ Prefer sync. block until atleast 90% of part of the method requires synchronisation.

ex ⇒


methodA {
CS
}
synchronised block


methodB {
CS
}
synchronised method

Class → Classname → Object [meta-data of class]

↓

Objects [instances]

[Synchronised(ClassName · class)

→ lock taken on entire class object

→ all methods calls via any object will be synchronised

```
class Count
        incrementCount

              synchronised ( Count.class)

        print() {
              synchronised ( Count.class)
        }
   class Counterter.      implements   Runnable {

              Count c;

              run() {
                    for( ———— ) {
                              c. incrementCount();
                    }
              }

   }
```

→ Count c1 = new Count();
→ c2

→ CounterFunc Cin1 (c1) ←

→ Cin2 (c2) ←

T1 → Cin1

T2 → Cin2

t1.start(); → 10000
t2.start(); → 10000

t1 ⟹ C1
t2 ⟹ C2

↓

Sequential

class Printer implement Runnable () {

run() {

print();

}

C3

C4

P1 → (C3)

p2 → (C4)

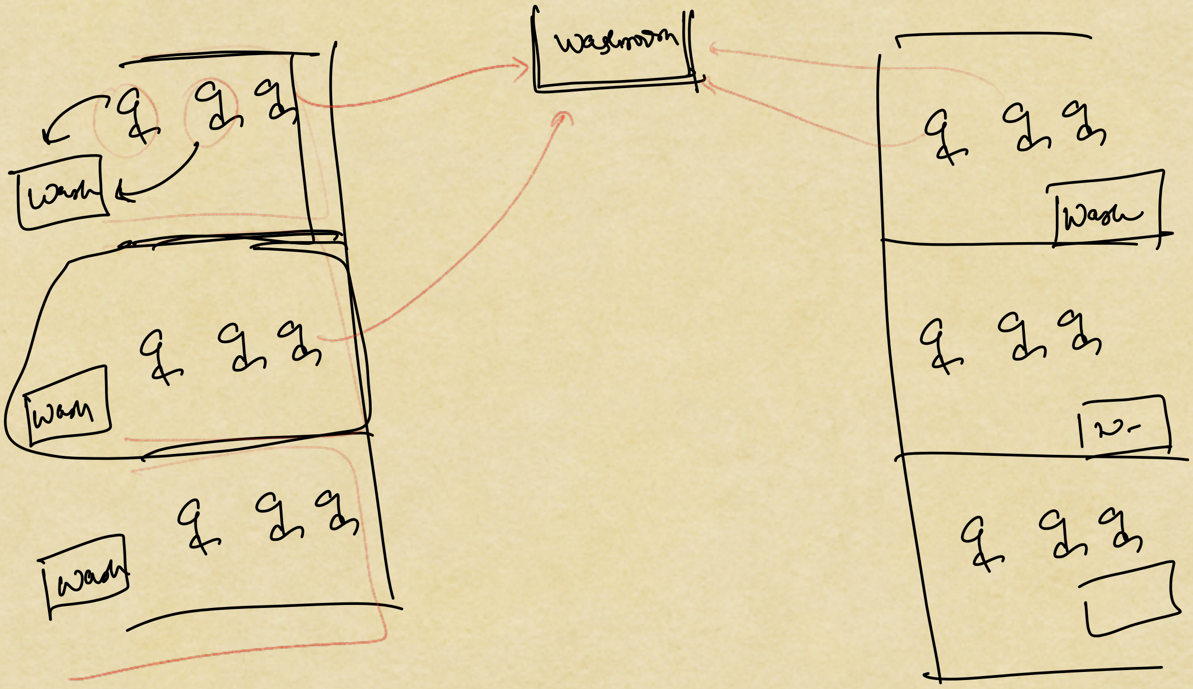
t3 → p1

t4 → p2

t3. start

t4. start()

} all 4 will execute sequential

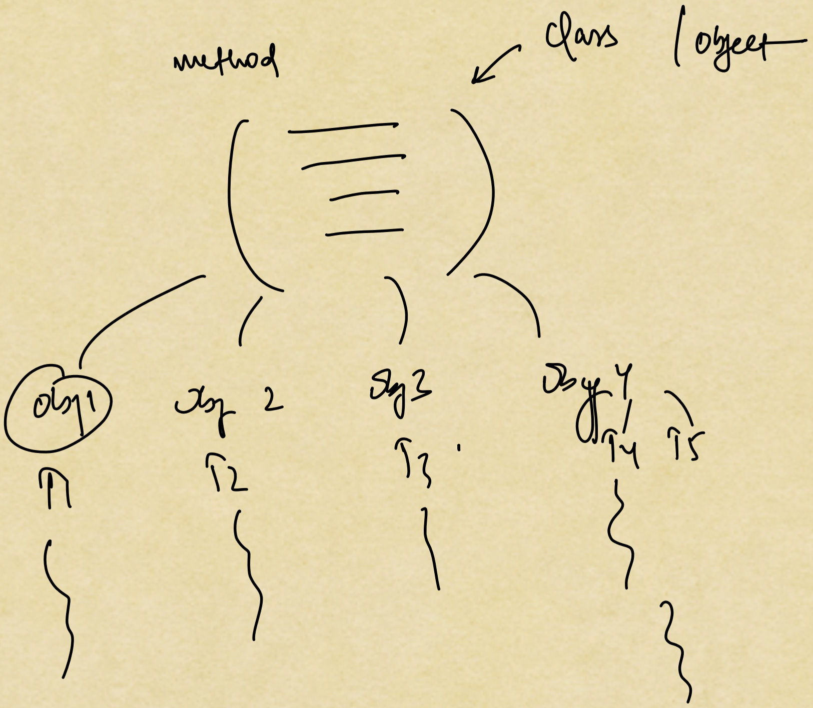


obj name

synchronised(this) {

}

Class



⇒ Semaphores

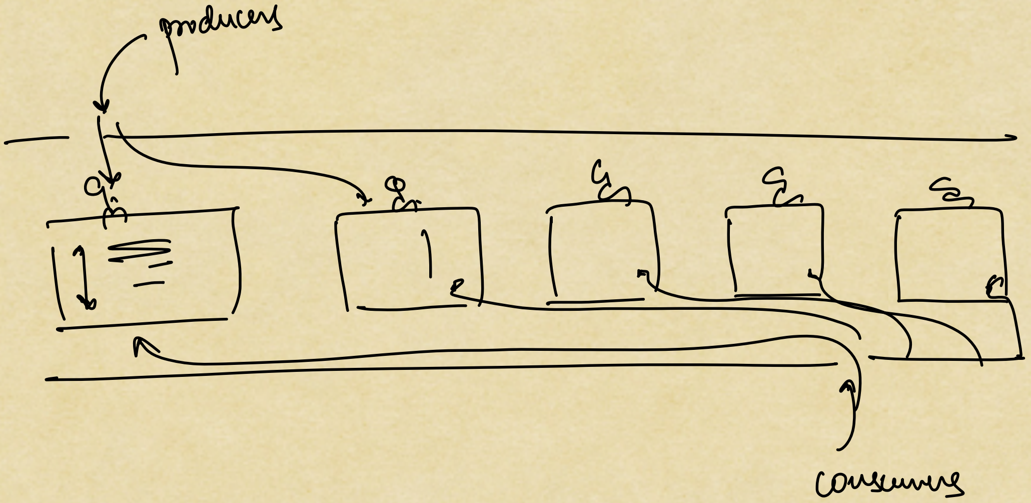
Producer - Consumer problem

eg ⇒ where/what ⇒ H.W

Tailor store sell the shirts that they have made, the shop has counters, where each tailor can sell a shirt

- each tailor can sell 1 shirt at a time at a particular counter
- customer only enters the shop if atleast 1 shirt available for them.

=> as fast as possible



→ No. of producers that can enter the store
= no. of empty slots

→ No. of consumers that can enter the store
= no. of filled slots

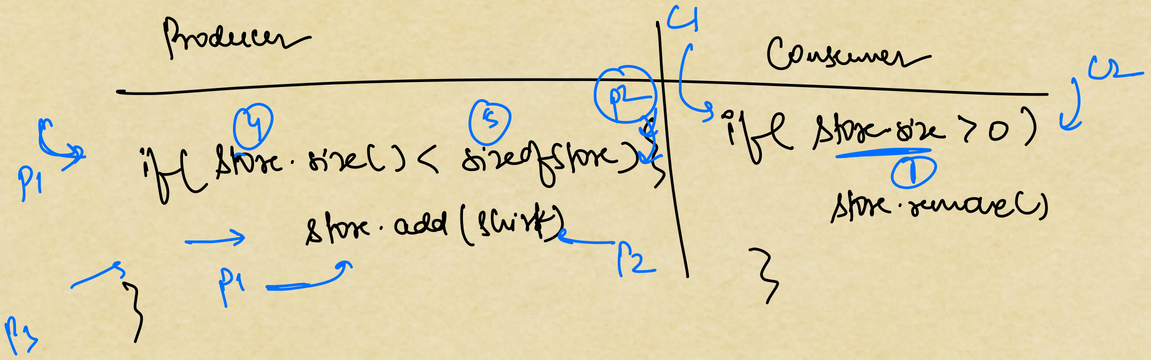
=> Implementation :-

```
Store => List<Shirt> store;
         int sizeofStore = 5;
```

Producer	Consumer

if(store.size() < sizeofStore){
    store.add(shirt)
}
	if(store.size > 0){
	    store.remove();
	}

There are 1000s of parallely running producers & consumers.



Producer

```
if(store.size() < sizeofstore){
    store.add(shirt)
}
```

Consumer

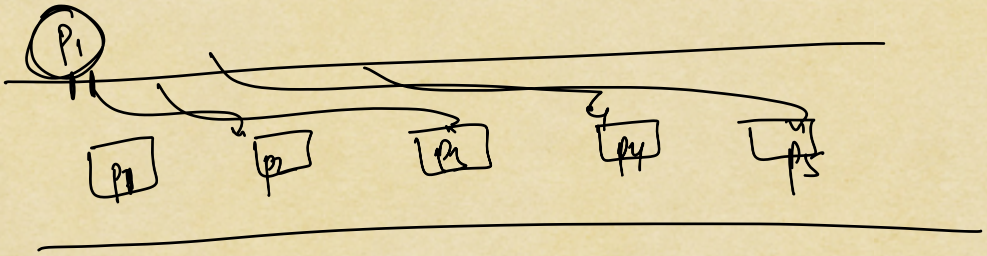
```
if(store.size > 0){
    store.remove()
}
```

Solution

1) Mutex | Synchronised :

At a time only 1 thread would be allowed to

$\underbrace{\text{add a shirt}}_{\text{producer}}$ | $\underbrace{\text{remove a shirt}}_{\text{consumer}}$



H.W find scenario

→ ability → ~~not to allow 100s threads~~

~~not to only allow 1 thread~~

allow a certain number of threads executing parallely

↓

Semaphore

Semaphore s = new Semaphore(N);

↓

no. of threads

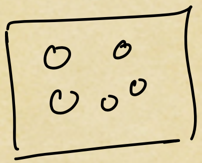
Mutex

Semaphore s = new Semaphore(1)

Mutex | Synchronised

Election Booth

Semaphore



GD room