Semaphore

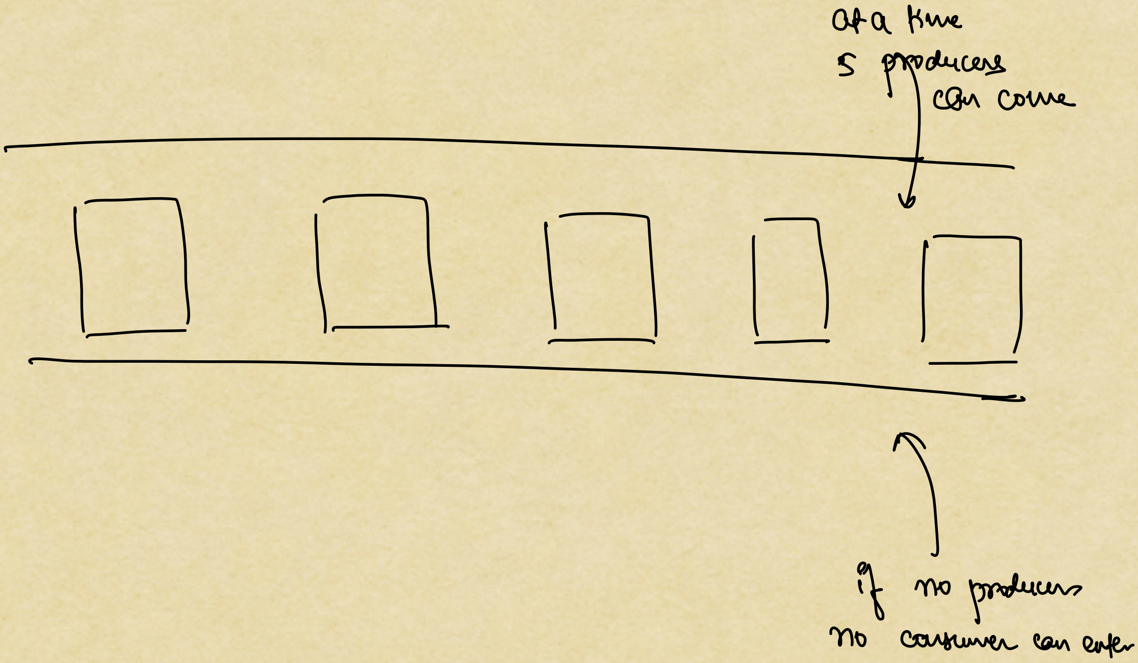
Atomic → AtomicInteger

Concurrent DS

=> Producer Consumer Problem:-

Imagine a store that allows tailors to sell the shirts they have made. The shop has counters, where each tailor can sell a shirt.

- * Each tailor can only sell 1 shirt at a time
- * Customer only enters the shop if atleast 1 shirt is available for sale



at a time 5 producers can come

if no producers no consumer can enter

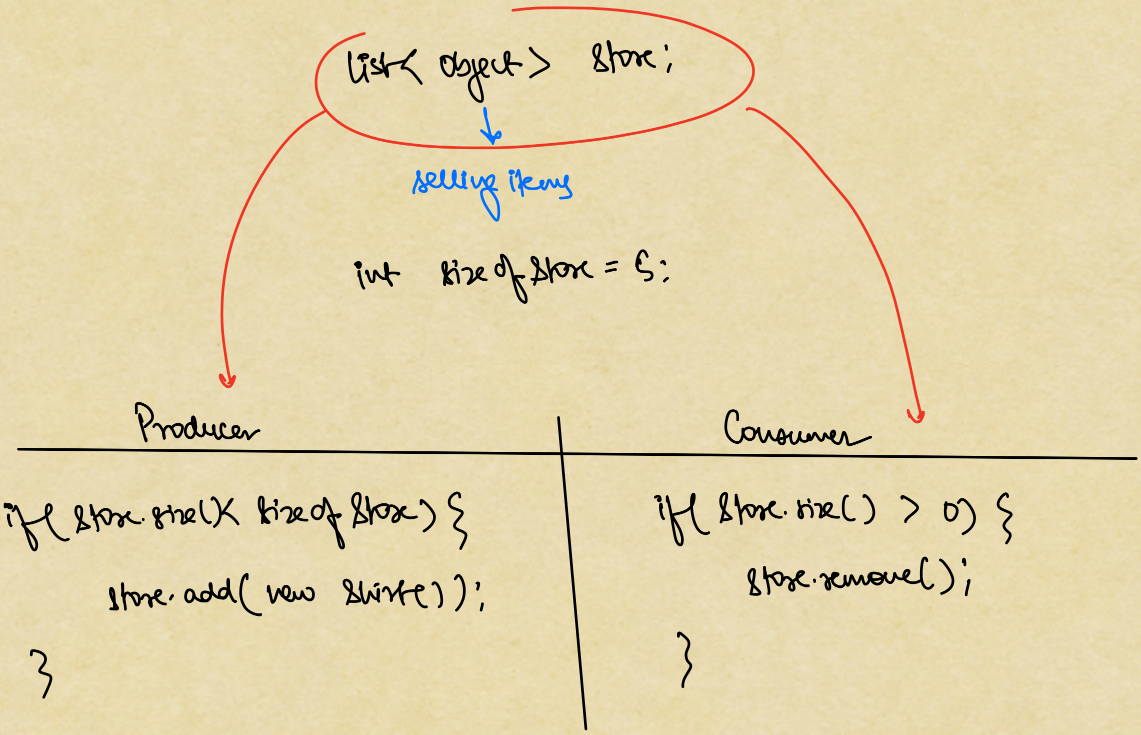
⇒ No. of producers that can enter the store

= no. of empty slots

⇒ No. of consumers that can enter the store

= no. of filled slots

⇒ Write the code for this, and make sure the store keeping running :-



```
list<object> store;
```

selling items

```
int sizeofStore = 5;
```

Producer

```
if(store.size()< sizeofStore){
    store.add(new Shirt());
}
```

Consumer

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

If I run 1000s of producers and consumers parallely

Producer

```
if( store.size() < sizeof store) {
CS → store.add( new Shirt());
}
```

Consumer

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

Solution :

i) Synchronised block | mutex lock

They will only 1 thread to go through the CS, ie, adding | removing from store.

Since, my store has multiple counters | slots I want multiple producers to be able sell and multiple consumers to be able to buy at the same time.

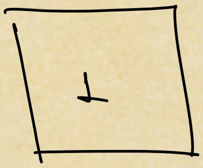
. to solve this problem, we will use semaphore,

```
Semaphore s = new Semaphore(N);
```

↳ will allow max N no. of threads, wherever semaphore is used.

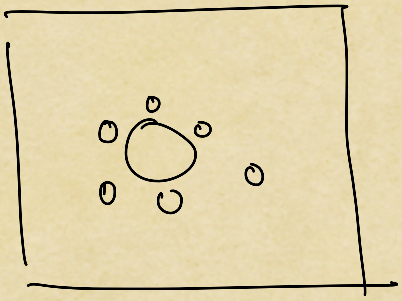
ex => Synchronised | Mutex

=>



Election Booth

Semaphore → GD Interview room [only fixed no. of people allowed in the room]



Lets assume, Store has 4 counters,

So we allow 4 threads to enter

possibility => a consumer but there is no producer

SEMAPHORE → integer counter + Synchronisation task => producer => add shirt to store

task => consumer => remove shirt from store

producer => acquires a threads

add a shirt to store

releases the thread ⟷ notify the consumer

consumer => acquires a threads

removes a shirt from shirt

releases the thread

initially we will give all 4 threads to producer :

- as soon as 1 producer acquires a threads,

  available threads = 3 (4-1)

- add a shirt

- as soon as the producer release the third t

  → notify the consumer, and

  available threads for consumer = 1

Producer

acquires()

add shirt

release()

no. of producers = ~~4~~ ~~3~~ 4

no. of consumers = ~~0~~ ~~1~~ 0

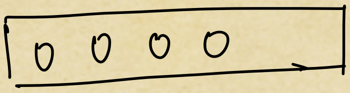
Store

Consumer

acquire()

removes shirt

releases()

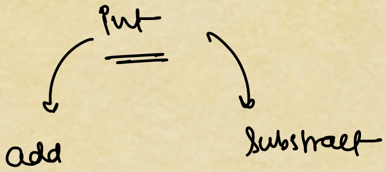


no. of producer allowed = ~~4~~ ~~3~~ ~~2~~ ~~1~~ 2

no. of consumers allowed = ~~0~~ ~~1~~ ~~0~~ ~~1~~ 2

P1	P2	P3	C
acquires	acquires	aquires	acquires
add	add	add	buys
releases	releases	releases	releases

⇒ ATOMIC DATA TYPES:-



* multiple threads can lead to inconsistency of data.

⇒ Some kind of Synchronisation is required:

Synchronised, Mutex, Semaphore

For every primitive datatype, we have an Atomic Datatype:

Atomic ⇒ 1 task on it at a time.

AtomicDatatype are inherently thread-safe [no inconsistency while multi-threading]

int ⇒ AtomicInteger

boolean ⇒ AtomicBoolean

long ⇒ AtomicLong