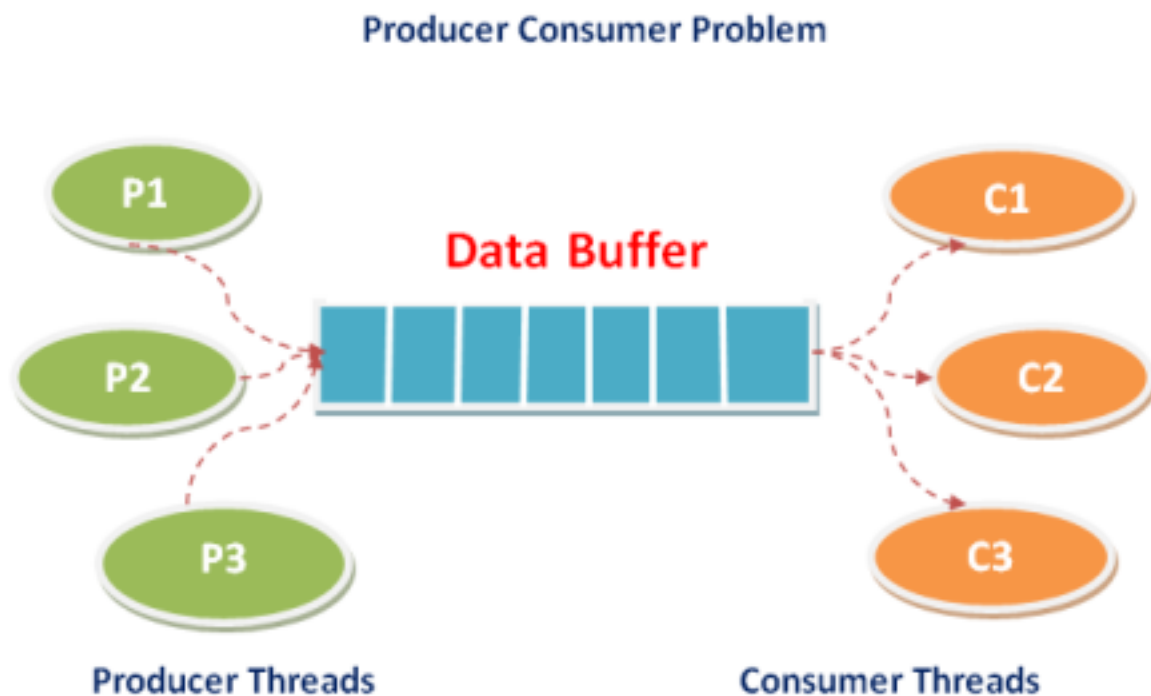


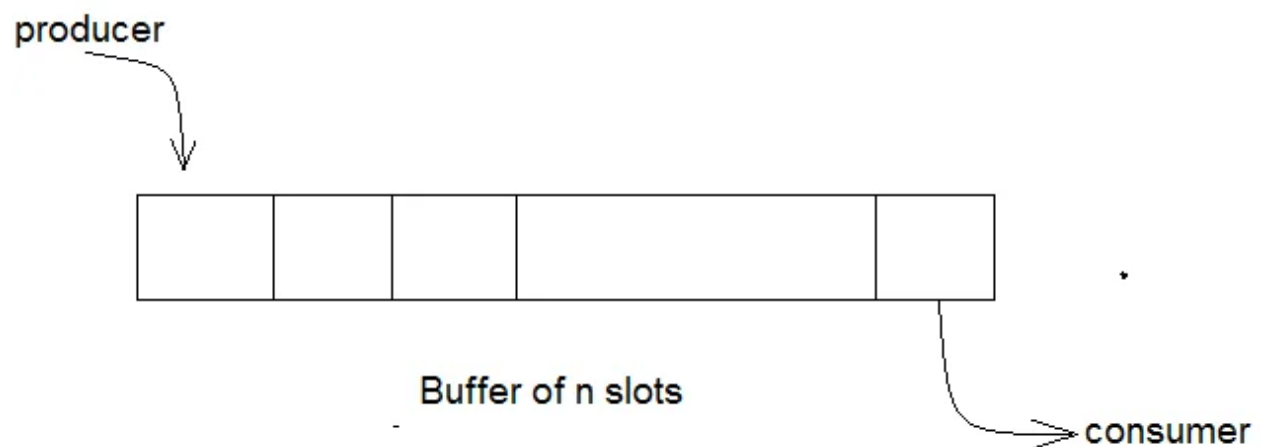
Bounded Buffer Problem



Problem Description:

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization.

There is a buffer of **N** slots, and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Bounded Buffer Problem Structure:

- The structure of the producer process

```
while (true) {  
.../* produce an item in next produced*/  
...  
wait(empty);  
wait(mutex);  
.../* add next produced to the buffer */  
...  
signal(mutex);  
signal(full);  
}
```

- The structure of the consumer process

```
while (true) {  
wait(full);  
wait(mutex);  
.../* remove an item from buffer to next  
consumed*/  
...  
signal(mutex);  
signal(empty);  
.../* consume the item in next consumed */  
...}
```

Bounded Buffer Problem Pseudocode:

Producer	Consumer
<pre>repeat ... #produce an item in nextp ... wait(empty); wait(mutex); ... #add nextp to buffer ... signal(mutex); signal(full); until false;</pre>	<pre>repeat wait(full); wait(mutex); ... # remove item from buffer to nextc ... signal(mutex); signal(empty); ... #consume the item in nextc ... until false;</pre>

Semaphore Pseudocode:

Semaphore operations on variable S are now defined as,

wait(S){

S.value--;

if (S.value <= 0) {

add this process to S.List; block(); //suspends,
waiting state
}

}

signal(S) { S.value++;

if (S.value > 0) {

remove a process P from S.List; wakeup(P); //resumes,
ready state
}

}

Solution:

Here's a Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:

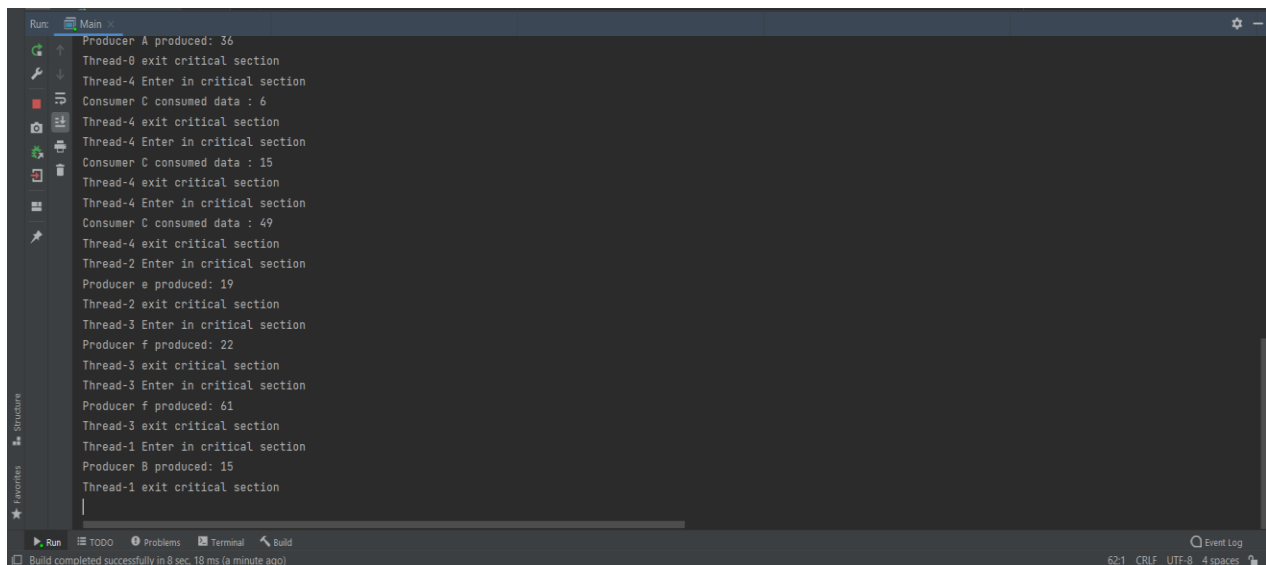
mutex, a **binary semaphore** which is used to acquire and release the lock, it's act like a gate which let one thread to enter.(initialized to value 1)

- **empty**: a **counting semaphore** whose initial value is the number of slots in the buffer, since initially all slots are empty. (Initialized to value n)
- **full**, a **counting semaphore** whose (initial value is 0) .

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

Deadlock Problem:

While using semaphores the concept of ownership of mutex(lock) and the order of increment and decrement of the semaphores should be kept in mind . Any change in order may lead to deadlock so, Order of Wait's are very important?



```
Run: Main
Producer A produced: 36
Thread-8 exit critical section
Thread-4 Enter in critical section
Consumer C consumed data : 6
Thread-4 exit critical section
Thread-4 Enter in critical section
Consumer C consumed data : 15
Thread-4 exit critical section
Thread-4 Enter in critical section
Consumer C consumed data : 49
Thread-4 exit critical section
Thread-2 Enter in critical section
Producer e produced: 19
Thread-2 exit critical section
Thread-3 Enter in critical section
Producer f produced: 22
Thread-3 exit critical section
Thread-3 Enter in critical section
Producer f produced: 61
Thread-3 exit critical section
Thread-1 Enter in critical section
Producer B produced: 15
Thread-1 exit critical section
```

Build completed successfully in 8 sec, 18 ms (a minute ago)

Explanation:

If we replace order of `wait(empty)` with `wait(mutex)` or replace order of `wait(full)` with `wait(mutex)`


It will prevent other producer or consumer to get in critical section because if we replace it that means we take the lock(mutex) before check the buffer is empty or full, so the consumer will get into critical section with (lock) and may be the buffer is empty so consumer will be wait in the queue with the lock and will not take item, so it's prevent other consumer to get into critical section because he have the key

From the producer view he will enter the critical section with lock and may be the buffer is full so he will not add something and wait in queue, so other producers cannot get into critical section because he has the key So, it will lead to **deadlock**

It's same problem if we replace `signal(mutex)` with `signal(full)`

Because producer must return the (lock) before leaving the critical section to let another consumer enter critical section

So, it will lead to **deadlock**

```
while(true) {  
    Buffer.mutex.acquire();   
    Buffer.full.acquire();   
    System.out.println(Thread.currentThread().getName () + " Enter in critical section");  
    data = buffer.removeItem();  
    System.out.println("Consumer " + this.name + " consumed data : " + data);  
    System.out.println(Thread.currentThread().getName () + " exit critical section");  
    sleep((r.nextInt( bound: 100)*10));  
    Buffer.mutex.release();  
    Buffer.empty.release();  
}  
}
```

```
while(true) {  
    Buffer.mutex.acquire();   
    Buffer.empty.acquire();   
    System.out.println(Thread.currentThread().getName () + " Enter in critical section");  
    int item = (int) (Math.random()*100);  
    System.out.println("Producer " + this.name + " produced: " + item);  
    sleep((r.nextInt( bound: 100)*10));  
    buffer.addItem(item);  
    System.out.println(Thread.currentThread().getName () + " exit critical section");  
    Buffer.mutex.release();  
    Buffer.full.release();  
}  
}
```


Solution of Deadlock :

We must verify the order of the wait and signal commands before deployment which is the responsible of programmers the semaphores should be kept in mind . Any change in order may lead to deadlock

```
while(true) {  
    Buffer.empty.acquire();  
    Buffer.mutex.acquire();  
    System.out.println(Thread.currentThread().getName () + " Enter in critical section");  
    int item = (int) (Math.random()*100);  
    System.out.println("Producer " + this.name + " produced: " + item);  
    sleep((r.nextInt( bound: 100)*10));  
    buffer.addItem(item);  
    System.out.println(Thread.currentThread().getName () + " exit critical section");  
    Buffer.mutex.release();  
    Buffer.full.release();  
}
```

```
while(true) {  
    Buffer.full.acquire();  
    Buffer.mutex.acquire();  
    System.out.println(Thread.currentThread().getName () + " Enter in critical section");  
    data = buffer.removeItem();  
    System.out.println("Consumer " + this.name + " consumed data : " + data);  
    System.out.println(Thread.currentThread().getName () + " exit critical section");  
    sleep((r.nextInt( bound: 100)*10));  
    Buffer.mutex.release();  
    Buffer.empty.release();  
}
```

Starvation Problem:

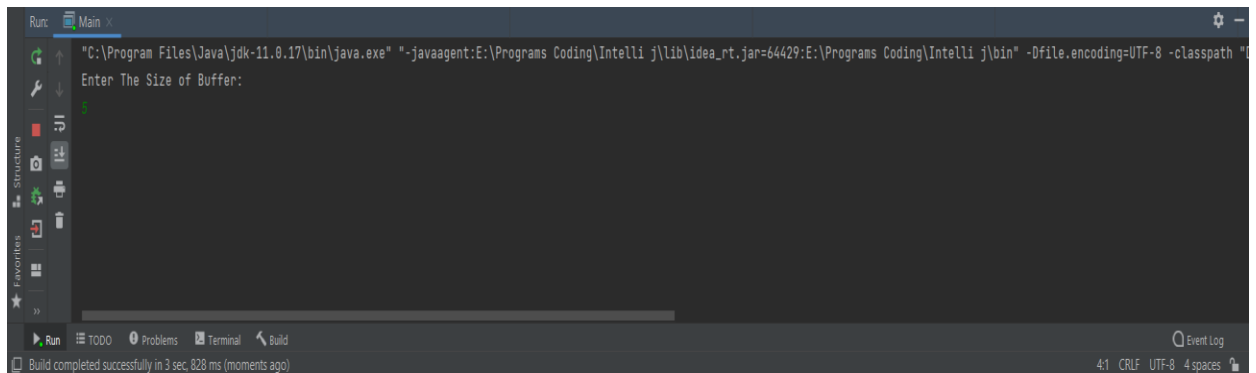
- If we put in producer

```
wait(full);  
wait(mutex);  
...  
#add to buffer  
...  
signal(mutex);  
signal(full);
```

- if we put in consumer:

```
wait(full);  
wait(mutex);  
...  
# remove item from buffer  
...  
signal(mutex);  
signal(empty);
```

```
while(true) {  
    Buffer.full.acquire(); ←  
    Buffer.mutex.acquire();  
    System.out.println(Thread.currentThread().getName () + " Enter in critical section");  
    int item = (int) (Math.random()*100);  
    System.out.println("Producer " + this.name + " produced: " + item);  
    sleep((r.nextInt( bound: 100)*10));  
    buffer.addItem(item);  
    System.out.println(Thread.currentThread().getName () + " exit critical section");  
    Buffer.mutex.release();  
    Buffer.full.release();  
}
```



so if we replace wait condition in producer and consumer it results in starvation. Starvation is the problem that occurs when high-priority processes keep executing and low priority processes get blocked for an indefinite time. Here producer and consumer process is not allowed.

another scenario:

- If we put in producer

```
wait(Empty);
```

```
wait(mutex);
```

```
...
```

```
#add to buffer
```

```
...
```

```
signal(mutex);
```

```
signal(Empty);
```

- if we put in consumer:

```
wait(full);
```

```
wait(mutex);
```

```
...
```

```
# remove item from buffer
```

```
...
```

```
signal(mutex);
```

```
signal(full);
```

```
17 while(true) {
18     Buffer.empty.acquire();
19     Buffer.mutex.acquire();
20     System.out.println(Thread.currentThread().getName () + " Enter in critical section");
21     int item = (int) (Math.random()*100);
22     System.out.println("Producer " + this.name + " produced: " + item);
23     sleep((r.nextInt( bound: 100)*10));
24     buffer.addItem(item);
25     System.out.println(Thread.currentThread().getName () + " exit critical section");
26     Buffer.mutex.release();
27     Buffer.empty.release();
28 }
29 }
```

Run: Main

Thread-1 Enter in critical section
Producer B produced: 12
Thread-1 exit critical section
Thread-1 Enter in critical section
Producer B produced: 79
Thread-1 exit critical section
Thread-1 Enter in critical section
Producer B produced: 80
Thread-1 exit critical section
Thread-1 Enter in critical section
Producer B produced: 62
Thread-1 exit critical section
Thread-1 Enter in critical section
Producer B produced: 64
Process finished with exit code 130

Build completed successfully in 4 sec, 104 ms (a minute ago)

```
19 while(true) {
20     Buffer.full.acquire();
21     Buffer.mutex.acquire();
22     System.out.println(Thread.currentThread().getName () + " Enter in critical section");
23     data = buffer.removeItem();
24     System.out.println("Consumer " + this.name + " consumed data : " + data);
25     System.out.println(Thread.currentThread().getName () + " exit critical section");
26     sleep((r.nextInt( bound: 100)*10));
27     Buffer.mutex.release();
28     Buffer.full.release();
29 }
30 }
31 catch (InterruptedException ex)
```

Run: Main

Thread-0 Enter in critical section
Producer A produced: 60
Thread-0 exit critical section
Thread-0 Enter in critical section
Producer A produced: 17
Thread-0 exit critical section
Thread-0 Enter in critical section
Producer A produced: 57
Thread-0 exit critical section
Thread-0 Enter in critical section
Producer A produced: 26
Process finished with exit code 130

Build completed successfully in 4 sec, 142 ms (moments ago)

results in starvation. producer at condition wait(empty) blocks the process that there are no items to give and wait(full) in consumer conditions blocks that there is no item to consume because it is already full it means something blocking high-priority processes keep executing and low priority processes get blocked for an indefinite time.

Conclusion:

- Consumer on certain condition is not returning the previously consumed buffer to empty buffer queue and continuing to wait for next ready buffer ready to be consumed.
- Or Producer on certain condition is not returning produced buffer to ready buffer queue and continuing to wait for empty buffer to produce.
- Then eventually this kind of situation will lead to starvation.

Solution of Starvation:

The solution of starvation in this situation we must use (FIFO)queue to prevent starvation as first come first served to avoid thread from waiting

Real World Problem Description:

Pizza restaurant

Suppose that we have cook shelf in pizza restaurant

That we put the meals on it ,so cook shelf has a capacity

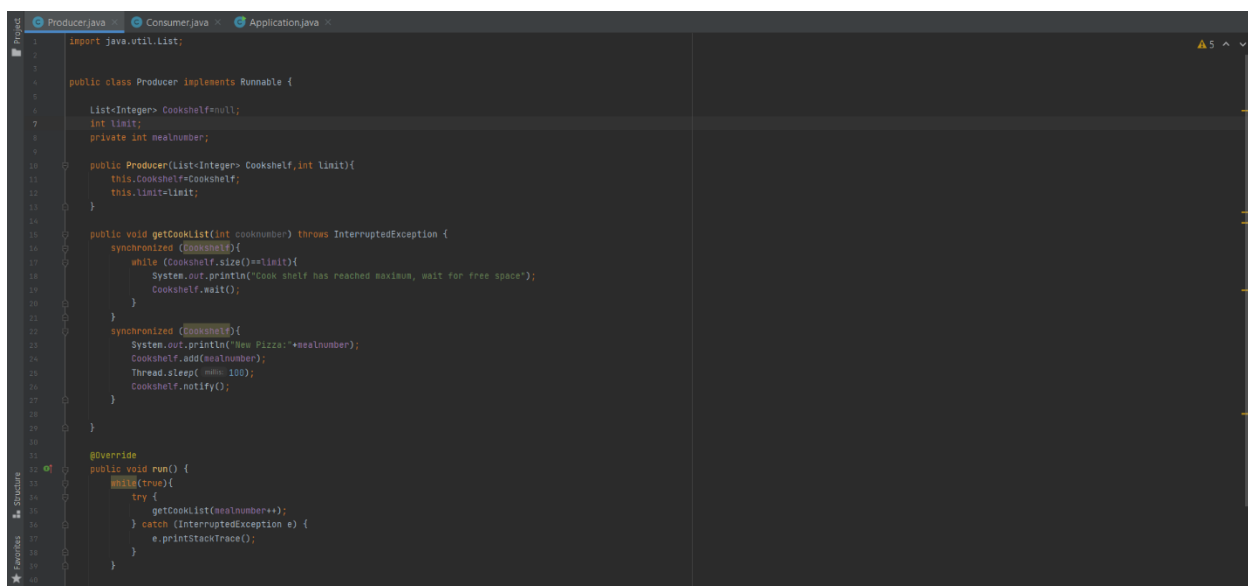
That can take for meals, and also we have here cooker which he cook the meal and servant which is serve the meal, so producer here is the cooker which he produce the meals and servant is the consumer which is serve the meal to customers, so when the cook shelf reach the limit the cooker(producer) must wait until free space is available after servant(consumer) take meal to customer to free space ,as same as for servant(consumer) he must wait for the cooker (producer) to make meals if the cook shelf is empty .

Producer class:

We have here cook shelf(shared variable between producer and consumer) variable which is initially null(free) and limit variable which is initially indicated the max size that can the shelf take

And meal number which describe the number of meal is produced

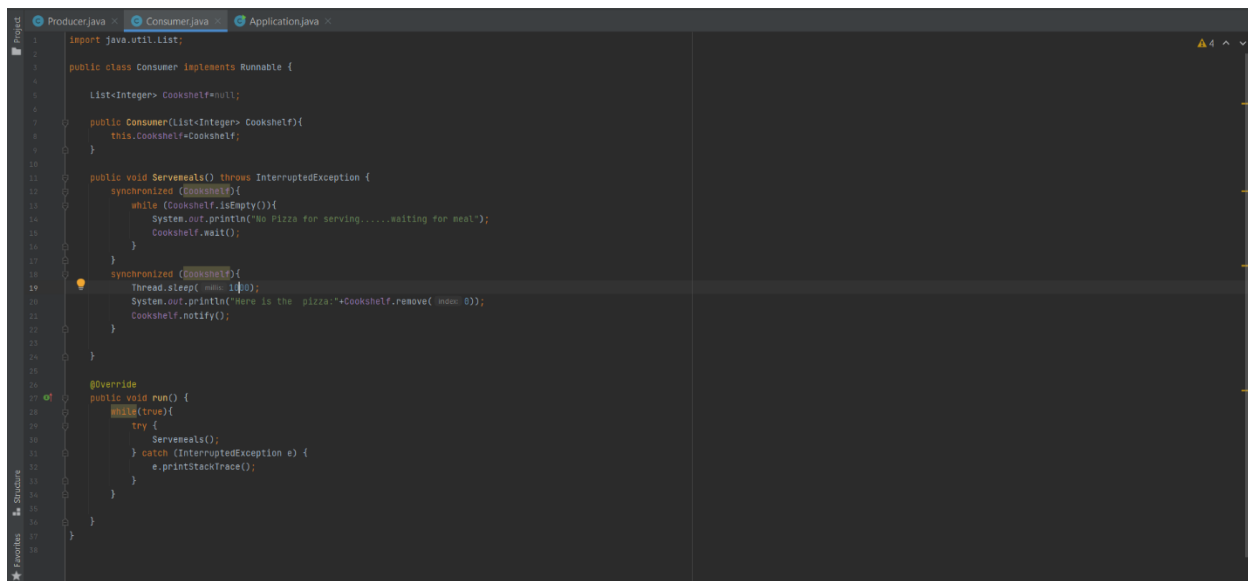
So in the constructor we send our producer to initial the list. Function get cook list take cook number that that will produce and we put synchronized block to protect our shared variable and make a while loop if it's reached the max size (limit) of the shelf ..print that the shelf is full and use wait() to cook shelf which mean that wait until the consumer to serve customers ,so print the number of pizza is produced and put it to shelf to be served then then notify(wakeup) consumer to serve the meals And wait until he serve some meals



```
1 import java.util.List;
2
3
4 public class Producer implements Runnable {
5
6     List<Integer> Cookshelf=null;
7     int limit;
8     private int mealNumber;
9
10    public Producer(List<Integer> Cookshelf,int limit){
11        this.Cookshelf=Cookshelf;
12        this.limit=limit;
13    }
14
15    public void getCookList(int cooknumber) throws InterruptedException {
16        synchronized (Cookshelf){
17            while (Cookshelf.size()==limit){
18                System.out.println("Cook shelf has reached maximum, wait for free space");
19                Cookshelf.wait();
20            }
21        }
22        synchronized (Cookshelf){
23            System.out.println("New Pizza:"+mealNumber);
24            Cookshelf.add(mealNumber);
25            Thread.sleep(1000);
26            Cookshelf.notify();
27        }
28    }
29
30    @Override
31    public void run() {
32        while(true){
33            try {
34                getCookList(mealNumber++);
35            } catch (InterruptedException e) {
36                e.printStackTrace();
37            }
38        }
39    }
40 }
```


Consumer class:

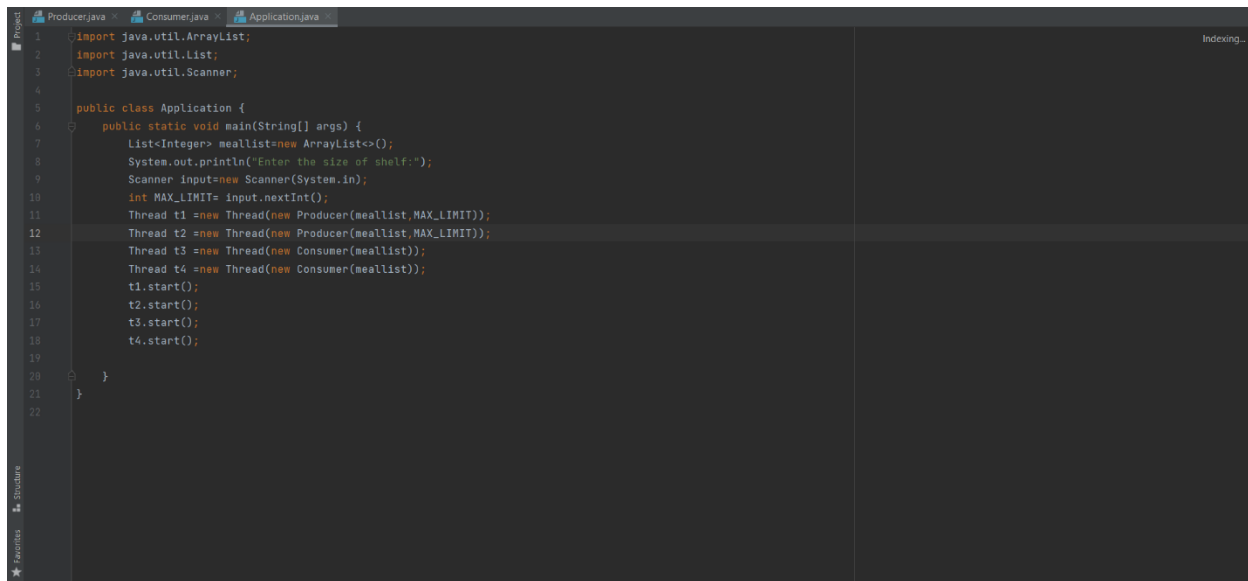
We have here cook shelf(shared variable between producer and consumer) variable which is initially null(free) So in the constructor we send our consumer to initial the list. Function get Serve meals ,we put synchronized block to protect our shared variable and make a while loop if it's empty print that the shelf is empty and there isn't pizza and use wait() to cook shelf which mean that wait until the producer to produce another pizza Then wait for (5seconds) that indicate the number of seconds the servant take to serve the customer Then notify(wakeup) the producer to produce another pizza For customers



```
1 import java.util.List;
2
3 public class Consumer implements Runnable {
4
5     List<Integer> Cookshelf=null;
6
7     public Consumer(List<Integer> Cookshelf){
8         this.Cookshelf=Cookshelf;
9     }
10
11     public void ServeMeals() throws InterruptedException {
12         synchronized (Cookshelf){
13             while (Cookshelf.isEmpty()){
14                 System.out.println("No Pizza for serving.....waiting for meal");
15                 Cookshelf.wait();
16             }
17         }
18         synchronized (Cookshelf){
19             Thread.sleep(5000);
20             System.out.println("Here is the pizza:"+Cookshelf.remove(0));
21             Cookshelf.notify();
22         }
23     }
24
25     @Override
26     public void run() {
27         while(true){
28             try {
29                 ServeMeals();
30             } catch (InterruptedException e) {
31                 e.printStackTrace();
32             }
33         }
34     }
35 }
```

The application class:

That is our main class that create the threads that will handle the tasks for cooker and servant(producer and consumer)



```
1  import java.util.ArrayList;
2  import java.util.List;
3  import java.util.Scanner;
4
5  public class Application {
6      public static void main(String[] args) {
7          List<Integer> meallist=new ArrayList<>();
8          System.out.println("Enter the size of shelf:");
9          Scanner input=new Scanner(System.in);
10         int MAX_LIMIT= input.nextInt();
11         Thread t1=new Thread(new Producer(meallist,MAX_LIMIT));
12         Thread t2=new Thread(new Producer(meallist,MAX_LIMIT));
13         Thread t3=new Thread(new Consumer(meallist));
14         Thread t4=new Thread(new Consumer(meallist));
15         t1.start();
16         t2.start();
17         t3.start();
18         t4.start();
19
20     }
21 }
22
```