**Lab 23**

**Exercise 1**

Create a new package called exercise1.

Create a new class called Buffer.java and place the following code in this class:

import java.util.ArrayList;

public class Buffer {

private final ArrayList<Integer> taskQueue;

private final int MAX\_CAPACITY;

public Buffer(ArrayList<Integer> sharedQueue, int c) {

this.taskQueue = sharedQueue;

MAX\_CAPACITY = c;

}

public int getMAX\_CAPACITY() {

return MAX\_CAPACITY;

}

public void produce(int i) {

{

synchronized (taskQueue) {

while (taskQueue.size() == MAX\_CAPACITY) {

System.out.println("Queue is full " + Thread.currentThread().getName() + " is waiting , size: " + taskQueue.size());

try {

taskQueue.wait();

} catch (InterruptedException ex) {

}

}

try {

Thread.sleep(1000);

} catch (InterruptedException ex) {

}

taskQueue.add(i);

System.out.println("Produced: " + i);

taskQueue.notifyAll();

}

}

}

public void consume() {

{

synchronized (taskQueue) {

while (taskQueue.isEmpty()) {

System.out.println("Queue is empty " + Thread.currentThread().getName() + " is waiting , size: " + taskQueue.size());

try {

taskQueue.wait();

} catch (InterruptedException ex) {

}

}

try {

Thread.sleep(1000);

} catch (InterruptedException ex) {

}

int j = taskQueue.remove(0);

System.out.println("Consumed: " + j);

taskQueue.notifyAll();

}

}

}

}

Create a new class called Producer.java and place the following code inside this class:

class Producer implements Runnable {

private Buffer buf;

public Producer(Buffer buf) {

this.buf = buf;

}

@Override

public void run() {

int counter = 0;

while (true) {

buf.produce(counter++);

}

}

}

Create a new class called Consumer.java and place the following code inside this class:

class Consumer implements Runnable {

private Buffer buf;

public Consumer(Buffer buf) {

this.buf = buf;

}

@Override

public void run() {

while (true) {

buf.consume();

}

}

}

Create a new class called TestProducerConsumer.java and place the following code inside this class:

import java.util.ArrayList;

public class TestProducerConsumer {

public static void main(String[] args) {

final ArrayList<Integer> taskQueue = new ArrayList<>();

final int SIZE = 5;

Buffer b = new Buffer(taskQueue, SIZE);

int MAX\_CAPACITY = 5;

Thread tProducer = new Thread(new Producer(b), "Producer");

Thread tConsumer = new Thread(new Consumer(b), "Consumer");

tProducer.start();

tConsumer.start();

}

}

Run the test program to view the output. The output will be different each time you run the program.

**Exercise 2**

The Java API contains a set of classes for dealing with concurrency. These are available in the java.util.concurrent package. The benefit of using classes in this package for thread programming is that they are thread-safe. In other words, we don’t have to worry about synchronisation. In the following exercise you will build an application that contains thread synchronisation without having to use the low level thread synchronisation techniques.

The class ArrayBlockingQueue is a bounded blocking queue backed by an array. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

The class ArrayBlockingQueue implements an interface called BlockingQueue which is basically a Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element. It is a queue which is thread safe to insert or retrieve elements from it. Also, it provides a mechanism which blocks requests for inserting new elements when the queue is full or requests for removing elements when the queue is empty, with the additional option to stop waiting when a specific timeout passes.

This functionality makes BlockingQueue a nice way of implementing the Producer-Consumer pattern, as the producing thread can insert elements until the upper limit of BlockingQueue while the consuming thread can retrieve elements until the lower limit is reached and of course with the support of the aforementioned blocking functionality.

Create a new package called exercise2

Create a new class called Producer.java and place the following code inside this class:

import java.util.Random;

import java.util.concurrent.BlockingQueue;

public class Producer implements Runnable {

private BlockingQueue queue = null;

public Producer(BlockingQueue queue) {

this.queue = queue;

}

@Override

public void run() {

Random rand = new Random();

int res = 0;

try {

res = add(rand.nextInt(100), rand.nextInt(50));

System.out.println("Produced: " + res);

queue.put(res);

Thread.sleep(1000);

res = add(rand.nextInt(100), rand.nextInt(50));

System.out.println("Produced: " + res);

queue.put(res);

Thread.sleep(1000);

res = add(rand.nextInt(100), rand.nextInt(50));

System.out.println("Produced: " + res);

queue.put(res);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

public int add(int x, int y) {

int result = 0;

result = x + y;

return result;

}

}

The Producer adds random integers and puts the result of the addition to a shared BlockingQueue.

Create a new class called Consumer.java and place the following code inside this class:

import java.util.concurrent.BlockingQueue;

public class Consumer implements Runnable {

protected BlockingQueue queue = null;

public Consumer(BlockingQueue queue) {

this.queue = queue;

}

@Override

public void run() {

try {

System.out.println("Consumed: " + queue.take());

System.out.println("Consumed: " + queue.take());

System.out.println("Consumed: " + queue.take());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

The Consumer takes the integers and prints them to the output. The Consumer uses the method take() which retrieves and removes the head of the queue and in case an element is not available, it blocks until this element becomes available.

Create a new class called TestProducerConsumer.java and place the following code inside this class:

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;

public class TestProducerConsumer {

public static void main(String[] args) throws Exception {

BlockingQueue bq = new ArrayBlockingQueue(1);

Producer producer = new Producer(bq);

Consumer consumer = new Consumer(bq);

Thread p = new Thread(producer);

Thread c = new Thread(consumer);

p.start();

c.start();

Thread.sleep(4000);

}

}

Run the test program to view the output. The output will be different each time you run the program.