Here is a quick start to get the gears turning...

ParkingLot is a class.

ParkingSpace is a class.

ParkingSpace has an Entrance.

Entrance has a location or more specifically, distance from Entrance.

ParkingLotSign is a class.

ParkingLot has a ParkingLotSign.

ParkingLot has a finite number of ParkingSpaces.

HandicappedParkingSpace is a subclass of ParkingSpace.

RegularParkingSpace is a subclass of ParkingSpace.

CompactParkingSpace is a subclass of ParkingSpace.

ParkingLot keeps array of ParkingSpaces, and a separate array of vacant ParkingSpaces in order of distance from its Entrance.

ParkingLotSign can be told to display "full", or "empty", or "blank/normal/partially occupied" by calling .Full(), .Empty() or .Normal()

Parker is a class.

Parker can Park().

Parker can Unpark().

Valet is a subclass of Parker that can call ParkingLot.FindVacantSpaceNearestEntrance(), which returns a ParkingSpace.

Parker has a ParkingSpace.

Parker can call ParkingSpace.Take() and ParkingSpace.Vacate().

Parker calls Entrance.Entering() and Entrance.Exiting() and ParkingSpace notifies ParkingLot when it is taken or vacated so that ParkingLot can determine if it is full or not. If it is newly full or newly empty or newly not full or empty, it should change the ParkingLotSign.Full() or ParkingLotSign.Empty() or ParkingLotSign.Normal().

HandicappedParker could be a subclass of Parker and CompactParker a subclass of Parker and RegularParker a subclass of Parker. (might be overkill, actually.)

In this solution, it is possible that Parker should be renamed to be Car.

**package** com.iris.thread;

**import** java.util.concurrent.atomic.AtomicInteger;

**public** **class** ThreeThreadsOrderedLockLess {

AtomicInteger sharedOutput = **new** AtomicInteger(0);

**private** Object object = **new** Object();

**public** **static** **void** main(String[] args) {

ThreeThreadsOrderedLockLess less = **new** ThreeThreadsOrderedLockLess();

MyThread t1 = less.**new** MyThread(0);

MyThread t2 = less.**new** MyThread(1);

MyThread t3 = less.**new** MyThread(2);

MyThread t4 = less.**new** MyThread(3);

MyThread t5 = less.**new** MyThread(4);

MyThread t6 = less.**new** MyThread(5);

t1.setName("Thread1");

t2.setName("Thread2");

t3.setName("Thread3");

t4.setName("Thread4");

t5.setName("Thread5");

t6.setName("Thread6");

t1.start();

t2.start();

t3.start();

t4.start();

t5.start();

t6.start();

}

**class** MyThread **extends** Thread{

**private** **int** finalPosition ;

**public** MyThread(**int** threadPosition) {

**this**.finalPosition = threadPosition;

}

@Override

**public** **void** run() {

**while**(sharedOutput.get() < 19) {

**if**(sharedOutput.get() % 6 == **this**.finalPosition) {

**synchronized** (object) {

System.***out***.println(Thread.*currentThread*().getName() + " "+ sharedOutput.getAndIncrement());

}

}

}

}

}

}

package com.iris.map;

import java.util.LinkedHashMap;

import java.util.Map;

public class LRUCacheImpl extends LinkedHashMap<Integer, String> {

private static final long serialVersionUID = 1L;

private int capacity;

public LRUCacheImpl(int capacity, float loadFactor){

super(capacity, loadFactor, true);

this.capacity = capacity;

}

/\*\*

\* removeEldestEntry() should be overridden by the user, otherwise it will not

\* remove the oldest object from the Map.

\*/

@Override

protected boolean removeEldestEntry(Map.Entry<Integer, String> eldest){

return size() > this.capacity;

}

public static void main(String arg[]){

LRUCacheImpl lruCache = new LRUCacheImpl(4, 0.75f);

lruCache.put(1, "Object1");

lruCache.put(2, "Object2");

lruCache.put(3, "Object3");

lruCache.get(1);

lruCache.put(4, "Object4");

System.out.println(lruCache);

lruCache.put(5, "Object5");

lruCache.get(3);

lruCache.put(6, "Object6");

System.out.println(lruCache);

lruCache.get(4);

lruCache.put(7, "Object7");

lruCache.put(8, "Object8");

System.out.println(lruCache);

}

}

public class ProducerConsumerService {

public static void main(String[] args) {

//Creating BlockingQueue of size 10

BlockingQueue<Message> queue = new ArrayBlockingQueue<>(10);

Producer producer = new Producer(queue);

Consumer consumer = new Consumer(queue);

//starting producer to produce messages in queue

new Thread(producer).start();

//starting consumer to consume messages from queue

new Thread(consumer).start();

System.out.println("Producer and Consumer has been started");

}

}

public class Consumer implements Runnable{

private BlockingQueue<Message> queue;

public Consumer(BlockingQueue<Message> q){

this.queue=q;

}

@Override

public void run() {

try{

Message msg;

//consuming messages until exit message is received

while((msg = queue.take()).getMsg() !="exit"){

Thread.sleep(10);

System.out.println("Consumed "+msg.getMsg());

}

}catch(InterruptedException e) {

e.printStackTrace();

}

}

}

public class Producer implements Runnable {

private BlockingQueue<Message> queue;

public Producer(BlockingQueue<Message> q){

this.queue=q;

}

@Override

public void run() {

//produce messages

for(int i=0; i<100; i++){

Message msg = new Message(""+i);

try {

Thread.sleep(i);

queue.put(msg);

System.out.println("Produced "+msg.getMsg());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

//adding exit message

Message msg = new Message("exit");

try {

queue.put(msg);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class Message {

private String msg;

public Message(String str){

this.msg=str;

}

public String getMsg() {

return msg;

}

}

**public** **class** ProducerConsumerInJava {

**public** **static** **void** main(String args[]) {

System.***out***.println("How to use wait and notify method in Java");

System.***out***.println("Solving Producer Consumper Problem");

Queue<Integer> buffer = **new** LinkedList<>();

**int** maxSize = 10;

Thread producer = **new** Producer(buffer, maxSize, "PRODUCER");

Thread consumer = **new** Consumer(buffer, maxSize, "CONSUMER");

producer.start();

consumer.start();

}

}

**class** Consumer **extends** Thread {

**private** Queue<Integer> queue;

**private** **int** maxSize;

**public** Consumer(Queue<Integer> queue, **int** maxSize, String name) {

**super**(name);

**this**.queue = queue;

**this**.maxSize = maxSize;

}

@Override

**public** **void** run() {

**while** (**true**) {

**synchronized** (queue) {

**while** (queue.isEmpty()) {

System.***out***.println("Queue is empty," + "Consumer thread is waiting"

+ " for producer thread to put something in queue");

**try** {

queue.wait();

} **catch** (Exception ex) {

ex.printStackTrace();

}

}

System.***out***.println("Consuming value : " + queue.remove());

queue.notifyAll();

}

}

}

}

**class** Producer **extends** Thread {

**private** Queue<Integer> queue;

**private** **int** maxSize;

**public** Producer(Queue<Integer> queue, **int** maxSize, String name) {

**super**(name);

**this**.queue = queue;

**this**.maxSize = maxSize;

}

@Override

**public** **void** run() {

**while** (**true**) {

**synchronized** (queue) {

**while** (queue.size() == maxSize) {

**try** {

System.***out***.println("Queue is full, " + "Producer thread waiting for "

+ "consumer to take something from queue");

queue.wait();

} **catch** (Exception ex) {

ex.printStackTrace();

}

}

Random random = **new** Random();

**int** i = random.nextInt();

System.***out***.println("Producing value : " + i);

queue.add(i);

queue.notifyAll();

}

}

}

}