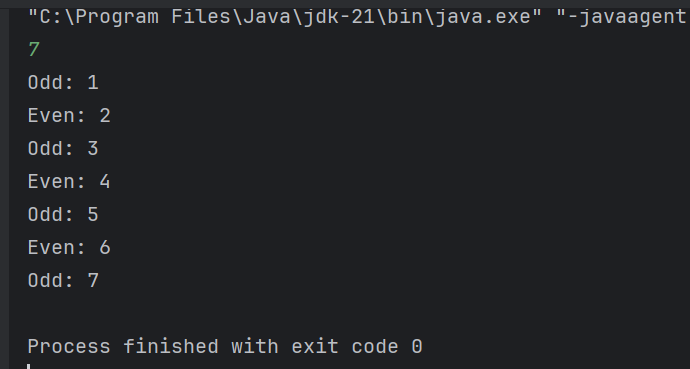
You are tasked with writing a Java program that prints odd and even numbers in sequence using two separate threads. One thread should print odd numbers, and the other thread should print even numbers. The threads should coordinate to ensure the numbers are printed in the correct order.

***Code –***

package Multithreading;  
import java.util.\*;  
  
public class OddEven {  
 static class OddEvenPrinter {  
 private int number = 1;  
 private final int max;  
 private boolean isOddTurn = true;  
 public OddEvenPrinter(int max) {  
 this.max = max;  
 }  
 public synchronized void printOdd() throws InterruptedException {  
 while (number < max) {  
 while (!isOddTurn) {  
 wait();  
 }  
 if (number % 2 == 1) {  
 System.*out*.println("Odd: " + number);  
 number++;  
 isOddTurn = false;  
 notifyAll();  
 }  
 }  
 }  
 public synchronized void printEven() throws InterruptedException {  
 while (number < max) {  
 while (isOddTurn) {  
 wait();  
 }  
 if (number % 2 == 0) {  
 System.*out*.println("Even: " + number);  
 number++;  
 isOddTurn = true;  
 notifyAll();  
 }  
 }  
 }  
 }  
  
 static class OddThread extends Thread {  
 private final OddEvenPrinter printer;  
 public OddThread(OddEvenPrinter printer) {  
 this.printer = printer;  
 }  
 public void run() {  
 try {  
 printer.printOdd();  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
 }  
 static class EvenThread extends Thread {  
 private final OddEvenPrinter printer;  
 public EvenThread(OddEvenPrinter printer) {  
 this.printer = printer;  
 }  
 public void run() {  
 try {  
 printer.printEven();  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
 }  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
 int max = sc.nextInt();  
 OddEvenPrinter printer = new OddEvenPrinter(max);  
 Thread oThread = new OddThread(printer);  
 Thread eThread = new EvenThread(printer);  
 oThread.start();  
 eThread.start();  
 }  
}

***Output –***

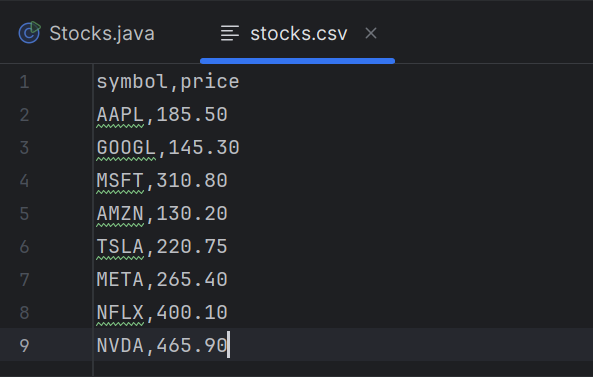
******

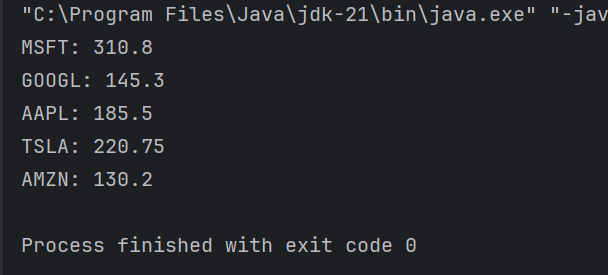
Implement a multi-threaded application in Java to fetch stock prices from a CSV file. The goal is to divide the list of stock symbols into smaller sub-lists, fetch their prices concurrently using multiple threads, and then combine the results.

***Code –***

package Multithreading;  
import java.io.\*;  
import java.util.\*;  
import java.util.concurrent.\*;  
  
public class Stocks {  
  
 private static Map<String, Double> loadStockData(String filePath) throws IOException {  
 Map<String, Double> stockData = new HashMap<>();  
 BufferedReader reader = new BufferedReader(new FileReader(filePath));  
 String line = reader.readLine(); // skip header  
 while ((line = reader.readLine()) != null) {  
 String[] parts = line.split(",");  
 if (parts.length == 2) {  
 String symbol = parts[0].trim();  
 double price = Double.*parseDouble*(parts[1].trim());  
 stockData.put(symbol, price);  
 }  
 }  
 reader.close();  
 return stockData;  
 }  
  
 private static Map<String, Double> fetchPricesConcurrently(List<String> symbols, Map<String, Double> stockData, int threadCount) throws InterruptedException, ExecutionException {  
 ExecutorService executor = Executors.*newFixedThreadPool*(threadCount);  
 List<Callable<Map<String, Double>>> tasks = new ArrayList<>();  
 int batchSize = symbols.size() / threadCount;  
 for (int i = 0; i < threadCount; i++) {  
 int start = i \* batchSize;  
 int end = (i == threadCount - 1) ? symbols.size() : (i + 1) \* batchSize;  
 List<String> subList = symbols.subList(start, end);  
 tasks.add(() -> {  
 Map<String, Double> result = new HashMap<>();  
 for (String symbol : subList) {  
 if (stockData.containsKey(symbol)) {  
 result.put(symbol, stockData.get(symbol));  
 }  
 }  
 return result;  
 });  
 }  
 List<Future<Map<String, Double>>> futures = executor.invokeAll(tasks);  
 Map<String, Double> finalResult = new HashMap<>();  
 for (Future<Map<String, Double>> future : futures) {  
 finalResult.putAll(future.get());  
 }  
 executor.shutdown();  
 return finalResult;  
 }  
  
 public static void main(String[] args) throws Exception {  
 String filePath = "C:\\Users\\DixitRaj\\IdeaProjects\\Exception Handling\\src\\Multithreading\\stocks.csv";  
 Map<String, Double> stockData = *loadStockData*(filePath);  
 List<String> symbolsToFetch = Arrays.*asList*("AAPL", "GOOGL", "MSFT", "AMZN", "TSLA");  
 int threadCount = 4;  
 Map<String, Double> result = *fetchPricesConcurrently*(symbolsToFetch, stockData, threadCount);  
 for (Map.Entry<String, Double> entry : result.entrySet()) {  
 System.*out*.println(entry.getKey() + ": " + entry.getValue());  
 }  
 }  
}

***Output –***

******

******

There are two cities, City A and City B, connected by a bridge. Only one person can cross the bridge at a time. To cross the bridge, a person must take a token from one end and deposit it at the other end. There is only one token available, and it must be shared by all residents of both cities. Initially, the token is in City B. Residents of City B must use the token to travel to City A first, and only then can residents of City A use the token to travel to City B.

***Code –***

package Multithreading;  
import java.util.LinkedList;  
import java.util.Queue;  
  
class Bridge {  
 private boolean tokenInCityB = true; // Token starts in City B  
 private final Queue<Person> cityBQueue = new LinkedList<>();  
 private final Queue<Person> cityAQueue = new LinkedList<>();  
 public synchronized void requestCrossing(Person person) throws InterruptedException {  
 if (person.getCity().equals("B")) {  
 cityBQueue.add(person);  
 } else {  
 cityAQueue.add(person);  
 }  
 while (!canCross(person)) {  
 wait();  
 }  
 System.*out*.println(person.getPersonName() + " is crossing the bridge from City " + person.getCity() + " to " +  
 (person.getCity().equals("B") ? "A" : "B"));  
 tokenInCityB = !tokenInCityB;  
 if (person.getCity().equals("B")) {  
 cityBQueue.poll();  
 } else {  
 cityAQueue.poll();  
 }  
 notifyAll();  
 }  
  
 private boolean canCross(Person person) {  
 return (person.getCity().equals("B") && tokenInCityB && cityBQueue.peek() == person) ||  
 (person.getCity().equals("A") && !tokenInCityB && cityAQueue.peek() == person);  
 }  
}  
  
class Person extends Thread {  
 private final String personName;  
 private final String city;  
 private final Bridge bridge;  
 public Person(String personName, String city, Bridge bridge) {  
 this.personName = personName;  
 this.city = city;  
 this.bridge = bridge;  
 }  
 public String getCity() {  
 return city;  
 }  
 public String getPersonName() { // Renamed method to avoid conflict with Thread class  
 return personName;  
 }  
 @Override  
 public void run() {  
 try {  
 bridge.requestCrossing(this);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}  
  
public class Bridgepass {  
 public static void main(String[] args) {  
 Bridge bridge = new Bridge();  
 Person p1 = new Person("Aman", "B", bridge);  
 Person p2 = new Person("Ankit", "B", bridge);  
 Person p3 = new Person("Nikhil", "B", bridge);  
 Person p4 = new Person("Dixit", "A", bridge);  
 Person p5 = new Person("Sanskar", "A", bridge);  
 Person p6 = new Person("Vijay", "A", bridge);  
 p1.start();  
 p2.start();  
 p3.start();  
 p4.start();  
 p5.start();  
 p6.start();  
 }  
}

***Output –***

