# Print Odd and Even Numbers Using Two Threads

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.

#### Requirements

* + 1. **Two Threads**: One thread prints odd numbers, and the other thread prints numbers.
    2. **Synchronization**: The threads must coordinate to print the numbers in sequence without any overlap or missing numbers.
    3. **Range**: The program should print numbers from 1 to a specified maximum value.

#### Implementation

* **OddEvenPrinter Class**: Manages the printing of odd and even numbers using two threads.
* **OddThread Class**: Represents the thread that prints odd numbers.
* **EvenThread Class**: Represents the thread that prints even numbers.

**CODE:**

package PrintOddEven;  
  
class OddEvenPrinter{  
 private final int max;  
 private int num=1;  
 private final Object lock=new Object();  
  
 public OddEvenPrinter(int max){  
 this.max=max;  
 }  
  
 public void printOdd(){  
 synchronized (lock){  
 while(num<=max){  
 if(num%2==0){  
 try{  
 lock.wait();  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }else{  
 System.*out*.println("Odd: "+num);  
 num++;  
 lock.notify();  
 }  
 }  
 }  
 }  
 public void printEven(){  
 synchronized (lock){  
 while(num<=max){  
 if(num%2!=0){  
 try {  
 lock.wait();  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }else{  
 System.*out*.println("Even: "+num);  
 num++;  
 lock.notify();  
 }  
 }  
 }  
 }  
  
}  
  
public class OddEvenPrint {  
 public static void main(String[] args) {  
 OddEvenPrinter oe=new OddEvenPrinter(10);  
 Thread ot1=new Thread(()->{oe.printOdd();});  
 Thread et1=new Thread(()->{oe.printEven();});  
  
 ot1.start();  
 et1.start();  
 }  
}

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

# Implement a multi-threaded application in Java to fetch stock prices from a CSV file

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.

**Requirements:**

1. **Input:**
   1. A CSV file containing stock symbols and their corresponding prices.
   2. A list of stock symbols to fetch prices for (e.g., ["AAPL", "GOOGL", "MSFT", "AMZN", "TSLA"]).
2. **Output:**
   1. A map or list of stock symbols with their corresponding prices.
3. **Constraints:**
   1. The list of stock symbols can be large (e.g., up to 10,000 symbols).
   2. The solution should efficiently utilize multiple threads to improve performance over a single-threaded implementation.
4. **Implementation Details:**
   1. Use Java's Thread class or the ExecutorService framework to manage threads.
   2. Implement a divide-and-conquer approach:
      1. Divide the list of stock symbols into smaller sub-lists.
      2. Fetch the stock prices for each sub-list in parallel using separate threads.
      3. Combine the results from all threads into a single map or list.

**CODE:**

package CSVFile;  
import java.io.\*;  
import java.util.\*;  
import java.util.concurrent.\*;  
  
public class StockPrice {  
  
 private static final int *THREAD\_POOL\_SIZE* = 4;  
  
 public static void main(String[] args) throws Exception {  
 // Step 1: Load stock data from CSV  
 Map<String, Double> stockData = *loadStockDataFromCSV*("src\\CSVFile\\stocks.csv");  
  
 // Step 2: List of stock symbols to fetch  
 List<String> symbolsToFetch = Arrays.*asList*("AAPL", "GOOGL", "MSFT", "AMZN", "TSLA", "NVDA");  
  
 // Step 3: Fetch prices using multiple threads  
 Map<String, Double> result = *fetchStockPricesConcurrently*(stockData, symbolsToFetch);  
  
 // Step 4: Print final results  
 System.*out*.println("Fetched Stock Prices:");  
 result.forEach((symbol, price) -> System.*out*.println(symbol + " => $" + price));  
 }  
  
 // Method to load CSV into a Map  
 private static Map<String, Double> loadStockDataFromCSV(String filePath) throws IOException {  
 Map<String, Double> stockMap = new HashMap<>();  
 try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {  
 String line;  
 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());  
 stockMap.put(symbol, price);  
 }  
 }  
 }  
 return stockMap;  
 }  
  
 // Concurrently fetch prices for requested symbols  
 private static Map<String, Double> fetchStockPricesConcurrently(Map<String, Double> stockData, List<String> symbols) throws InterruptedException, ExecutionException {  
 ExecutorService executor = Executors.*newFixedThreadPool*(*THREAD\_POOL\_SIZE*);  
 List<Future<Map<String, Double>>> futures = new ArrayList<>();  
  
 int chunkSize = (int) Math.*ceil*((double) symbols.size() / *THREAD\_POOL\_SIZE*);  
  
 for (int i = 0; i < symbols.size(); i += chunkSize) {  
 List<String> sublist = symbols.subList(i, Math.*min*(i + chunkSize, symbols.size()));  
  
 Callable<Map<String, Double>> task = () -> {  
 Map<String, Double> subResult = new HashMap<>();  
 for (String symbol : sublist) {  
 if (stockData.containsKey(symbol)) {  
 subResult.put(symbol, stockData.get(symbol));  
 }  
 }  
 return subResult;  
 };  
  
 futures.add(executor.submit(task));  
 }  
  
 // Combine results from all futures  
 Map<String, Double> finalResult = new HashMap<>();  
 for (Future<Map<String, Double>> future : futures) {  
 finalResult.putAll(future.get());  
 }  
  
 executor.shutdown();  
 return finalResult;  
 }  
}

**OUTPUT:**

**A screen shot of a computer

AI-generated content may be incorrect.**

# Problem Statement: Bridge Crossing with Shared Token

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.

#### Requirements

1. **Bridge**: A shared resource that only one person can use at a time.
2. **Token**: A token with one permit representing the token that controls access to the bridge.
3. **Direction Control**: A mechanism to ensure that the token is used by residents of City B to travel to City A first, and then by residents of City A to travel to City B.
4. **Implement Waiting Queue:** Implement queue for both cities so that people will get the turn to cross the city.

#### Implementation

1. **Bridge Class**: Manages the token and the direction control.
2. **Person Class**: Represents a person who wants to cross the bridge.
3. **BridgeManagement Class**: Creates instances of Person and starts their threads.

CODE:

package SharedToken;  
  
import java.util.concurrent.\*;  
import java.util.concurrent.locks.\*;  
  
public class Bridge {  
 private final Semaphore token = new Semaphore(1); // Only one token  
 private final Lock bridgeLock = new ReentrantLock(); // Only one person on bridge  
 private volatile String currentDirection = "B"; // Token starts in City B  
 private final BlockingQueue<Person> queueA = new LinkedBlockingQueue<>();  
 private final BlockingQueue<Person> queueB = new LinkedBlockingQueue<>();  
  
 public void requestToCross(Person person) throws InterruptedException {  
 if (person.getCity().equals("A")) {  
 queueA.put(person);  
 } else {  
 queueB.put(person);  
 }  
 }  
  
 public void cross(Person person) throws InterruptedException {  
 BlockingQueue<Person> myQueue = person.getCity().equals("A") ? queueA : queueB;  
 BlockingQueue<Person> oppositeQueue = person.getCity().equals("A") ? queueB : queueA;  
  
 // Wait for your turn  
 while (true) {  
 if (!myQueue.peek().equals(person)) {  
 Thread.*sleep*(50);  
 continue;  
 }  
  
 if (!currentDirection.equals(person.getCity())) {  
 Thread.*sleep*(50);  
 continue;  
 }  
  
 token.acquire(); // Take token  
 bridgeLock.lock(); // Enter bridge  
  
 try {  
 System.*out*.println(person + " is crossing the bridge...");  
 Thread.*sleep*(1000); // simulate crossing  
 System.*out*.println(person + " has crossed.");  
 } finally {  
 bridgeLock.unlock(); // Exit bridge  
 token.release(); // Release token  
 myQueue.poll(); // Remove self from queue  
 }  
  
 // If my queue is empty, switch direction  
 if (myQueue.isEmpty() && !oppositeQueue.isEmpty()) {  
 currentDirection = oppositeQueue.peek().getCity();  
 System.*out*.println("Direction switched to City " + currentDirection);  
 }  
  
 break;  
 }  
 }  
}

package SharedToken;  
  
public class Person implements Runnable {  
 private final String name;  
 private final String city;  
 private final Bridge bridge;  
  
 public Person(String name, String city, Bridge bridge) {  
 this.name = name;  
 this.city = city;  
 this.bridge = bridge;  
 }  
  
 public String getCity() {  
 return city;  
 }  
  
 @Override  
 public void run() {  
 try {  
 bridge.requestToCross(this);  
 bridge.cross(this);  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
  
 @Override  
 public String toString() {  
 return "Person[" + name + " from City " + city + "]";  
 }  
}

package SharedToken;  
  
public class BridgeManagement {  
 public static void main(String[] args) {  
 Bridge bridge = new Bridge();  
  
 // Create threads from both cities  
 for (int i = 1; i <= 5; i++) {  
 new Thread(new Person("B" + i, "B", bridge)).start();  
 }  
  
 for (int i = 1; i <= 5; i++) {  
 new Thread(new Person("A" + i, "A", bridge)).start();  
 }  
 }  
}

**OUTPUT:**

