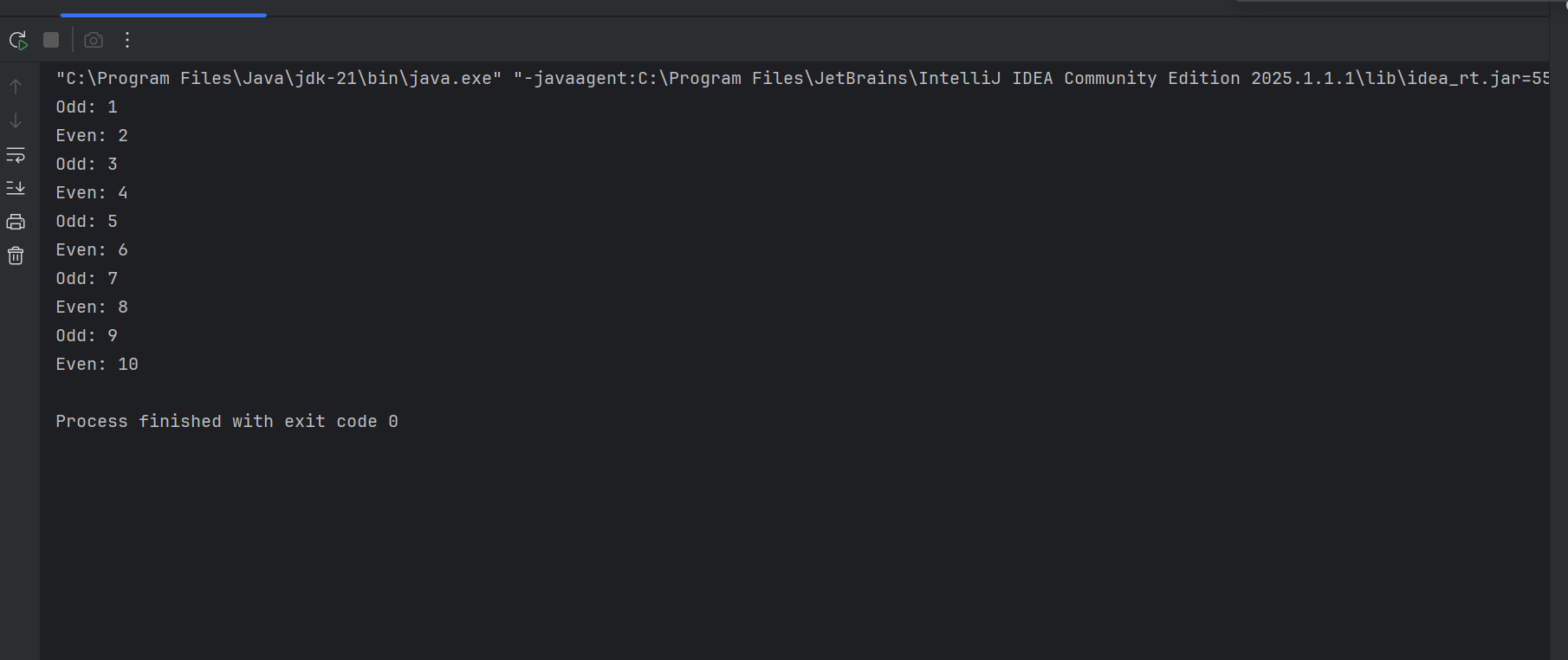
1. You are tasked with writing a Java program that prints odd and even numbers in sequence using two separate threads. One thread should print odd numbers, and the other thread should print even numbers. The threads should coordinate to ensure the numbers are printed in the correct order.Requirements
2. **Two Threads**: One thread prints odd numbers, and the other thread prints numbers.
3. **Synchronization**: The threads must coordinate to print the numbers in sequence without any overlap or missing numbers.
4. **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.
* class OddEvenPrinter {  
   private int number = 1;  
   private final int max;  
   public OddEvenPrinter(int max) {  
   this.max = max;  
   }  
    
   public synchronized void printOdd() throws InterruptedException {  
   while (number <= max) {  
   if (number % 2 == 0) {  
   wait();  
   } else {  
   System.*out*.println("Odd: " + number);  
   number++;  
   notify();  
   }  
   }  
   }  
    
   public synchronized void printEven() throws InterruptedException {  
   while (number <= max) {  
   if (number % 2 == 1) {  
   wait();  
   } else {  
   System.*out*.println("Even: " + number);  
   number++;  
   notify();  
   }  
   }  
   }  
    
   public static void main(String[] args) {  
   OddEvenPrinter printer = new OddEvenPrinter(10);  
   Thread oddThread = new Thread(() -> {  
   try {  
   printer.printOdd();  
   } catch (InterruptedException e) {  
   Thread.*currentThread*().interrupt();  
   }  
   });  
   Thread evenThread = new Thread(() -> {  
   try {  
   printer.printEven();  
   } catch (InterruptedException e) {  
   Thread.*currentThread*().interrupt();  
   }  
   });  
   oddThread.start();  
   evenThread.start();  
   }  
  }

  
  
  
  
  
  
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.

import java.util.\*;  
import java.util.concurrent.\*;  
import java.util.concurrent.atomic.\*;  
  
class Bridge {  
 private String tokenCity = "B"; // Initially in City B  
 private boolean isBridgeOccupied = false;  
  
 private final Queue<Person> queueA = new LinkedList<>();  
 private final Queue<Person> queueB = new LinkedList<>();  
  
 public synchronized void arrive(Person person) {  
 Queue<Person> queue = person.city.equals("A") ? queueA : queueB;  
 queue.offer(person);  
 System.*out*.println(person.name + " from City " + person.city + " arrives and waits.");  
  
 while (true) {  
 if (tokenCity.equals(person.city) &&  
 queue.peek() == person &&  
 !isBridgeOccupied) {  
 isBridgeOccupied = true;  
 queue.poll(); // Remove from queue  
 break;  
 }  
  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
  
 System.*out*.println(person.name + " from City " + person.city + " is crossing the bridge.");  
  
 try {  
 Thread.*sleep*(ThreadLocalRandom.*current*().nextInt(500, 1500));  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
  
 person.city = person.city.equals("A") ? "B" : "A";  
 System.*out*.println(person.name + " has crossed to City " + person.city + ".");  
  
 isBridgeOccupied = false;  
  
 // If current city's queue is empty, move the token  
 if ((tokenCity.equals("A") && queueA.isEmpty()) ||  
 (tokenCity.equals("B") && queueB.isEmpty())) {  
 tokenCity = tokenCity.equals("A") ? "B" : "A";  
 System.*out*.println("Token moved to City " + tokenCity + ".");  
 }  
  
 notifyAll(); // Wake up all waiting threads  
 }  
}  
  
class Person extends Thread {  
 public String name;  
 public String city;  
 private final Bridge bridge;  
  
 public Person(String name, String city, Bridge bridge) {  
 this.name = name;  
 this.city = city;  
 this.bridge = bridge;  
 }  
  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(ThreadLocalRandom.*current*().nextInt(100, 1000)); // Random arrival  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 bridge.arrive(this);  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Bridge bridge = new Bridge();  
  
 List<Person> people = new ArrayList<>();  
  
 int numFromA = 3;  
 int numFromB = 3;  
  
 for (int i = 1; i <= numFromA; i++) {  
 people.add(new Person("Person\_A\_" + i, "A", bridge));  
 }  
  
 for (int i = 1; i <= numFromB; i++) {  
 people.add(new Person("Person\_B\_" + i, "B", bridge));  
 }  
  
 for (Person p : people) {  
 p.start();  
 }  
  
 for (Person p : people) {  
 try {  
 p.join();  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
  
 System.*out*.println("All people have crossed the bridge.");  
 }  
}

