# 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 even 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.

# 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.

# 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.