Overview of the System

This ticketing system has two main types of components:

1. **Producers** (Ticket Producers): These simulate event organizers who release batches of tickets in a controlled manner.
2. **Consumers** (Ticket Consumers): These represent customers trying to purchase tickets.

The ticketing system uses a shared resource, a `BlockingQueue`, as a buffer to hold tickets. The queue handles concurrency so that producers and consumers don't interfere with each other, ensuring that each ticket is sold only once and that no customer receives duplicate tickets.

Here's a breakdown of how each part of the code works.

---

1. Defining the Shared Resource: `BlockingQueue`

java Copy code

```java
private static final int TICKET_CAPACITY = 50; // Maximum number of tickets available at a time private static final BlockingQueue<String> ticketQueue = new ArrayBlockingQueue<>(TICKET_CAPACITY);
```

- `TICKET_CAPACITY`: This constant defines the maximum capacity of the queue (i.e., the number of tickets that can be held in the queue at any given time). In this example, we set it to 50, simulating a situation where there can only be 50 tickets available in the buffer.
- `ticketQueue`: This is an instance of `ArrayBlockingQueue` with a maximum capacity (`TICKET_CAPACITY`). The `BlockingQueue` is a thread-safe structure that allows threads to add and remove elements while handling concurrency. It will:
  - Block a producer if it tries to add more tickets when the queue is full.
  - Block a consumer if it tries to remove a ticket when the queue is empty.

---

2. Producer Class: TicketProducer

java Copy code

```
static class TicketProducer implements Runnable { @Override public void run() { int
ticketId = 1; try { while (true) { for (int i = 0; i < 10; i++) { String ticket =
"Ticket-" + ticketId++; ticketQueue.put(ticket); // Adds ticket to the queue, waits if
full System.out.println("Released " + ticket); } Thread.sleep(5000); // Wait before
releasing the next batch of tickets } } catch (InterruptedException e) {
Thread.currentThread().interrupt(); } } }
```

- **Ticket Creation**: The `TicketProducer` simulates an event organizer releasing tickets. It generates a unique ticket ID and creates a string representing each ticket (e.g., "Ticket-1", "Ticket-2").
- **Adding Tickets to the Queue**: The `put()` method adds tickets to the `ticketQueue`. If the queue is full (50 tickets already in the queue), `put()` will automatically block until there's space, ensuring that the producer cannot overflow the queue.
- **Batch Release Simulation**: After each batch of 10 tickets is added, the producer waits (sleeps) for 5 seconds before releasing the next batch. This simulates a real-world scenario where tickets are released in waves rather than all at once.
- **Infinite Loop**: The `while (true)` loop keeps the producer thread running indefinitely, periodically releasing new tickets.

---

3. Consumer Class: TicketConsumer

java Copy code

```java
static class TicketConsumer implements Runnable { private final String customerName;
public TicketConsumer(String customerName) { this.customerName = customerName; }
@Override public void run() { try { while (true) { String ticket = ticketQueue.take(); //
Takes a ticket from the queue, waits if empty System.out.println(customerName + "
purchased " + ticket); Thread.sleep(1000); // Simulate time taken to process the purchase
} } catch (InterruptedException e) { Thread.currentThread().interrupt(); } } }
```

- **Customer Identity**: Each `TicketConsumer` instance has a unique `customerName`, simulating different customers. This name helps distinguish between different consumers in the output logs.
- **Ticket Purchase**: The `take()` method removes a ticket from `ticketQueue`. If no tickets are available, `take()` will block the consumer thread until a new ticket is added by the producer.
- **Purchase Processing Time**: After each purchase, the consumer waits (sleeps) for 1 second to simulate the time it might take to process the purchase, such as handling payment or generating a receipt.
- **Infinite Loop**: The consumer runs in an infinite loop, attempting to purchase tickets as they become available.

4. Main Method: Running the System

java Copy code

```java
public static void main(String[] args) { // Initialize producer and consumer threads
Thread producerThread = new Thread(new TicketProducer()); Thread consumerThread1 = new
Thread(new TicketConsumer("Customer1")); Thread consumerThread2 = new Thread(new
TicketConsumer("Customer2")); Thread consumerThread3 = new Thread(new
TicketConsumer("Customer3")); // Start the producer and consumer threads
producerThread.start(); consumerThread1.start(); consumerThread2.start();
consumerThread3.start(); }
```

- **Producer Thread**: This thread continuously releases tickets by creating and adding them to `ticketQueue`.
- **Consumer Threads**: Three separate consumer threads are created, each simulating a different customer. They independently attempt to buy tickets as they become available in the queue.
- **Thread Start**: The `start()` method initiates all threads, allowing them to run concurrently.

Key Features of This System

1. **Real-Time Ticket Release and Purchase**:
   - Tickets are released in batches by the producer, and customers attempt to purchase them as soon as they're available. This simulates a real-world scenario where tickets for an event become available, and customers compete to purchase them.
2. **BlockingQueue as the Buffer**:
   - The `BlockingQueue` provides automatic blocking behavior, handling both the queue's full and empty states. This ensures that the producer and consumer threads are safely synchronized without needing to manage `wait()` or `notify()` calls explicitly.
3. **Concurrency**:
   - Multiple consumer threads simulate a high-demand environment where many customers try to buy tickets simultaneously. Each consumer independently tries to access the queue, and the `BlockingQueue` ensures each ticket is only accessed by one consumer.
4. **Flow Control**:
   - The `BlockingQueue` prevents the producer from adding tickets if the queue reaches its maximum capacity, and it prevents consumers from removing tickets if none are available. This keeps the system stable and prevents overselling or lost tickets.

Real-World Enhancements and Practicality

In a real-world ticketing system, additional features would be added, such as:

- **Transaction Management**: Payment processing could be included in the consumer class, ensuring customers only get a ticket if their payment is successful. Failed transactions could return the ticket to the queue.
- **Customer Limits**: We could add checks so that each customer is limited to a certain number of tickets per transaction, which is common in ticketing systems.
- **Scalability**: More producer and consumer threads could be added as needed, allowing for multiple release agents and more customers to handle high ticket demand.

This example showcases how the Producer-Consumer pattern is ideal for real-time ticketing scenarios. By leveraging the `BlockingQueue` , this system provides thread-safe ticket distribution, efficient concurrency, and load balancing, making it a robust solution for managing simultaneous ticket releases and purchases.