# Producer-Consumer without Blocking Queue using SQLite, Java, wait()/notify()

## Overview

The **Producer-Consumer problem** is a concurrency scenario where a **Producer** thread generates data and places it into a **shared buffer**, and a **Consumer** thread takes data from that buffer to process it. Synchronization is essential to avoid issues like data inconsistency or deadlocks.

In this project, we demonstrate the **Producer-Consumer pattern** by simulating how data is passed between two systems using threads. The core idea is to show how a **Producer** thread generates data and a **Consumer** thread processes it in real time, with both threads coordinating through Java's `wait()` and `notifyAll()` synchronization methods.

1. The **source** is the `orders.json` file. It contains a list of sales orders in a simple, readable format.
2. The **destination** is the SQLite database (`orders.db`). This is where we want the final data to live.
3. **Controlled flow with buffer:** By using a `SharedBuffer`, we control how data moves between the reader and the writer. It avoids race conditions and makes sure data is not lost. The poison pill ensures a clean shutdown after the last item is processed.

### Summary

- We read data from a JSON **source container** (temporary).
- We write data into a SQLite **destination container** (permanent).
- We use two threads to simulate real-time processing: a **Producer** and a **Consumer**.
- The **buffer** manages handoff between them, safely and cleanly.

This setup demonstrates how real systems handle data flow across components especially when timing, synchronization, and correctness matter.

## Project Structure

```
producer-consumer/
   pom.xml
   src/
      main/java/org/example/
         Main.java
```

```
        Producer.java
        Consumer.java
        SharedBuffer.java
        OrderEntity.java
        DBManager.java
    resources/
        orders.json

test/java/org/example/
DBManagerTest.java
OrderEntityTest.java
ProducerConsumerIntegrationTest.java
```

## Components Overview

### Main.java

Coordinates the application. It sets up the buffer, launches threads, and connects everything.

### Producer.java

Reads a JSON file of orders, parses them into `OrderEntity` objects, and places them into the buffer. Sends a "poison pill" (order ID -1) to signal completion.

### Consumer.java

Consumes orders from the buffer and inserts them into the SQLite database. Stops when it receives the poison pill.

### SharedBuffer.java

A synchronized one-slot buffer using `wait()` and `notifyAll()`. Implements producer-consumer logic manually.

### DBManager.java

Handles database connection, table creation, order insertion, and fetching. Uses SQLite via JDBC.

### OrderEntity.java

A plain Java object that holds a single order's data. Includes `toString()` override for easy logging.

## Running the Application

### Prerequisites

- Java 17+
- Maven installed
- SQLite JDBC driver (`sqlite-jdbc-3.44.1.0.jar`)

### Build and Run

```
mvn clean compile
mvn exec:java -Dexec.mainClass="org.example.Main"
```

This will:

- Load orders from `src/main/resources/orders.json`
- Start producer and consumer threads
- Insert data into `orders.db`

## Running Tests

JUnit 5 tests are included for each component.

```
mvn test
```

### Included Tests

- `DBManagerTest` verifies insert and fetch logic
- `OrderEntityTest` validates object structure and formatting
- `ProducerConsumerIntegrationTest` full end-to-end verification

## Sample JSON Format

```json
[
  {
    "orderId": 1,
    "customerName": "Alice",
    "status": "NEW",
    "amount": 120.0,
    "orderDate": "2025-01-01"
  }
]
```

## Output

Running Main.java

Running tests

alt text

Figure 1: alt text

alt text

Figure 2: alt text

### Assumptions

- Date format is `yyyy-MM-dd`
- No missing or malformed fields in the input JSON
- The SQLite table will be cleared on each run (via `clearTable()`)

### Concepts Demonstrated

- Java multithreading (`Runnable`, `wait/notify`)
- Manual shared buffer synchronization
- File I/O with JSON parsing (`org.json`)
- SQLite persistence using JDBC
- Unit and integration testing using JUnit 5
- Clean, modular Maven project structure

## Note

**Look at Producer__Consumer__Blocking__Queue Implementation to understand how this can be implmeneted in modern way using Blocking Queue.**