NOTES

Multithreading in Java with the Producer-Consumer Problem 🧵

Introduction to Multi-Threading

Multithreading is a process of executing multiple threads simultaneously. A thread is a lightweight sub-process, the smallest unit of processing.

Both multiprocessing and multithreading are used to achieve multitasking. Multithreading is preferred over multiprocessing because: Threads share memory, reducing memory usage. Context switching between threads is faster than between processes. Uses of Multithreading in Java:
Games (for handling multiple actions simultaneously).
Animations (to smoothly update graphics).
Real-time applications (e.g., chat applications, web servers)

Example :- Gaming Applications

- In multiplayer online games, multiple tasks happen simultaneously:
- The game physics are calculated (jumping, running, shooting).
- The graphics are being rendered.
- The network communication updates the game state for multiple players.
- The sound effects play in the background.
- Each of these tasks is handled by different threads, ensuring smooth gameplay.

NOTES

Way to Create a Thread in Java

1. **Extending the Thread class**

In this method, we create a class that extends the Thread class and override its run() method. Then, we create an object of this class and call the start() method to begin the execution of the thread.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread(); // Creating thread object
        t1.start(); // Starting the thread
    }
}
```

NOTES

2. Implementing the Runnable interface

In this method, we create a class that implements the Runnable interface and override its run() method. Then, we create an instance of this class and pass it to a Thread object, which is then started using start().

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running...");
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t1 = new Thread(myRunnable); // Passing runnable instance to thread
        t1.start(); // Starting the thread
    }
}
```

3. Daemon Thread

A Daemon Thread in Java is a background thread that runs in the JVM and provides support to non-daemon (user) threads. Daemon threads are typically used for tasks like garbage collection, background monitoring, and other lowpriority jobs that should not prevent the JVM from shutting down. One key feature of daemon threads is that they automatically terminate when all user threads in the program finish execution. This means that the JVM does not wait for daemon threads to complete before shutting down. We can convert a thread into a daemon thread by calling the setDaemon(true) method before starting the thread.

NOTES

Producer-Consumer Problem

In this problem, there are two types of threads: a Producer, which generates data and adds it to a shared buffer, and a Consumer, which takes data from the buffer and processes it. The challenge arises because both threads work on the same buffer, which has a limited size. If the buffer is full, the Producer must wait until the Consumer removes some data, and if the buffer is empty, the Consumer must wait for the Producer to add new data.

Understanding the Problem with a Real-Life Example

Example = A Bakery Shop Imagine a bakery where:

- A baker (producer) bakes cakes and puts them on the counter.
- A customer (consumer) buys cakes from the counter.

Rules in the bakery:

1. The baker should not bake a cake if there's already a cake on the counter (avoids overproduction).
2. The customer should not take a cake if there's no cake on the counter (avoids consuming nothing).
3. Synchronization is required to ensure that the customer takes a cake only when it is available and the baker bakes a new one only when the counter is empty.

NOTES

Algorithm for Producer-Consumer Problem

Step 1: Define Shared Resource (Company Class)

1. Create a class Company with: o An integer n to store the produced item. o A boolean Turn to track whether Producer or Consumer should execute.
   - o Two synchronized methods:
     - produce_item(n): Produces an item.
     - consume_item(): Consumes an item.

Step 2: Implement Producer Thread

1. Create a class Producer that extends Thread.
2. In the run() method:
   - o Continuously produce items (1, 2, 3, ...).
   - o Call produce_item(n) from Company.
   - o Use Thread.sleep(1000) to simulate production delay.

Step 3: Implement Consumer Thread

1. Create a class Consumer that extends Thread.
2. In the run() method:
   - o Continuously consume items.
   - o Call consume_item() from Company.
   - o Use Thread.sleep(1000) to simulate consumption delay.

NOTES

Step 4: Implement Synchronization in Shared Resource (Company)
Produce Item
1. If Turn == true (meaning an item is already produced), the Producer:
    o Calls wait() to pause execution.
2. Else:
    o Store the produced item in n.
    o Print "Produced: n".
    o Set Turn = true (indicating Consumer should execute).
    o Call notify() to wake up the Consumer. Consume Item

1. If Turn == false (meaning no item is produced), the Consumer:
    o Calls wait() to pause execution.
2. Else:
    o Print "Consumed: n".
    o Set Turn = false (indicating Producer should execute).
    o Call notify() to wake up the Producer.

Step 5: Create and Start Threads

1. In Main class:
o Create an instance of Company.
o Create instances of Producer and Consumer, passing the Company instance.
o Start both threads.

NOTES

Main.java

```java
public class Main {
    public static void main(String[] args) {
        Company obj = new Company();
        Producer pro = new Producer(obj);
        Consumer con = new Consumer(obj);
        pro.start();
        con.start();
    }
}
```

NOTES

Company.java

```java
public class Company {
    int n;
    boolean Turn = false; // false → Producer can produce, true → Consumer can consume

    synchronized public void produce_item(int n) {
        if (Turn) {
            try {
                wait(); // Producer waits if item is already produced
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        this.n = n;
        System.out.println("Produced: " + this.n);
        Turn = true; // Now, Consumer can consume
        notify(); // Wake up Consumer
    }

    synchronized public int consume_item() {
        if (!Turn) {
            try {
                wait(); // Consumer waits if no item is available
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        System.out.println("Consumed: " + this.n);
        Turn = false; // Now, Producer can produce
        notify(); // Wake up Producer
        return this.n;
    }
}
```

NOTES

Producer.java

```java
public class Producer extends Thread {
    Company c;

    Producer(Company c) {
        this.c = c;
    }

    public void run() {
        int i = 1;
        while (true) {
            this.c.produce_item(i);
            try {
                Thread.sleep(1000); // Simulate production time
            } catch (Exception e) {
                e.printStackTrace();
            }
            i++;
        }
    }
}
```

NOTES

Consumer.java

```
public class Consumer extends Thread {
    Company c;

    Consumer(Company c) {
        this.c = c;
    }

    public void run() {
        while (true) {
            this.c.consume_item();
            try {
                Thread.sleep(1000); // Simulate consumption time
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```