CO3408 – Week 7 Lab Sheet

Modelling and Implementing Coordination in Concurrency

Learning Outcomes

By completing this lab, students will be able to:

- Create and manage threads in Java.

- Use `join()` to correctly wait for thread completion.

- Explain the difference between *synchronisation* and *coordination*.

- Implement coordination using `wait()`, `notify()`, and `notifyAll()`.

- Apply the Producer–Consumer pattern in Java.

Part 1 – Creating, Starting, and Waiting for Threads

Task 1.1 – Create a Simple Thread

Write a Java class that: - Implements `Runnable`. Create 3 threads and start them.

```java
class HelloTask implements Runnable {
    @Override
             public void run() {
         System.out.println("Hello from: " +
Thread.currentThread().getName());
     }
       }
public class Task1Test {
    public static void main(String[] args) {
        Thread t1 = new Thread(new HelloTask(), "T1");
        Thread t2 = new Thread(new HelloTask(), "T2");
        Thread t3 = new Thread(new HelloTask(), "T3");
        t1.start();
        t2.start();
        t3.start();
//        try {
//            t1.join();
//            t2.join();
//            t3.join();
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }
        System.out.println("All threads completed!");
    }
   }
```

Questions: 1. What happens to the order of output? 2. Why is thread scheduling unpredictable?

Task 1.2 – Using `join()`

Modify your code so the main thread waits for all 3 threads to finish. Use join() method and uncomment the above commented code and run.

Questions:

1. How does `join()` change the behaviour of the program?

2. Why might `join()` be important in real applications?

Part 2 – Thread Priorities (Demonstration)

Task 2.1 – Experiment with Priorities

Set random priorities for each thread using:

`thread.setPriority(1 + (int)(Math.random()*10));`

Run the program multiple times.

```
public class ThreadPriorityDemo {
    public static void main(String[] args) {
        // Create and start 5 threads with random priorities
        for (int i = 1; i <= 5; i++) {
            Thread t = new Thread(new Task(), "Thread-" + i);
            // Set a random priority between 1 and 10
            int priority = 1 + (int)(Math.random() * 10);
            t.setPriority(priority);
            System.out.println(t.getName() + " has priority: " + priority);
            t.start();
        }
    }

    // Simple task that prints the thread name and a loop counter
    static class Task implements Runnable {
        @Override
        public void run() {
            for (int i = 1; i <= 5; i++) {
                System.out.println(Thread.currentThread().getName() + "
running: " + i);
                try {
                    Thread.sleep(100); // Sleep briefly to simulate work
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
}
```

Questions:

1. Do high-priority threads always run first?

2. Why are priorities only *hints* for the scheduler?

Part 3 – Coordination vs Synchronisation

Task 3.1 – Analyse the Difference

In your own words, explain the difference between: - **Synchronisation** (e.g., using `synchronized`) - **Coordination** (e.g., using `wait()` / `notifyAll()`)

Give an example of when synchronisation alone is NOT enough.

Part 4 – Implementing Producer–Consumer in Java

Task 4.1 – Implement a Shared Buffer

Create a class `Buffer` with: - A fixed-size array (size = 5) - `produce(item)` method - `consume()` method

Use: - `synchronized` - `wait()` - `notifyAll()`

Task 4.2 – Implement Producer and Consumer Threads

Create: - `Producer implements Runnable` - `Consumer implements Runnable`

Producer: generates 10 items.

Consumer: consumes 10 items.

Start both threads from a `main()` method.

```java
public class ProducerAndConsumer {
      public static void main(String[] args) {
            Buffer buffer = new Buffer();
            new Thread(new Producer(buffer)).start();
            new Thread(new Consumer(buffer)).start();
      }
}
class Buffer {
      private String[] items = new String[5];
      private int count = 0;
      public synchronized void produce(String item) {
            while (count == items.length) { // buffer full
                  try { wait(); } catch (InterruptedException e) {}
            }
            items[count++] = item;
            notifyAll();
      }
```

```
    public synchronized String consume() {
        while (count == 0) { // buffer empty
            try { wait(); } catch (InterruptedException e) {}
        }
        String item = items[--count];
        notifyAll();
        return item;
    }
}
class Producer implements Runnable {
    private Buffer buffer;
    public Producer(Buffer b) { buffer = b; }
    public void run() {
        for (int i = 1; i <= 10; i++) {
            String item = "Item" + i;
            System.out.println("Producing: " + item);
            buffer.produce(item);
        }
    }
}
class Consumer implements Runnable {
    private Buffer buffer;
    public Consumer(Buffer b) { buffer = b; }
    public void run() {
        for (int i = 1; i <= 10; i++) {
            String item = buffer.consume();
            System.out.println("Consuming: " + item);
        }
    }
}
```

Task 4.3 – Run and Analyse Output

Observe the order of produced and consumed items.

Questions:

1. What happens if you remove the `wait()` calls?

2. What happens if you remove `notifyAll()`?

3. Why must `wait()` and `notifyAll()` be inside synchronised blocks?

Task 4.4 – Graceful Thread Stopping (volatile + interrupt)

Worker Thread with a stop flag

```
class Worker implements Runnable {
    private volatile boolean running = true;
    public void stop() { running = false; }
    @Override
    public void run() {
        while (running) {
            System.out.println("Working...");
```

```
                  try { Thread.sleep(400); } catch (InterruptedException
e) {}
            }
            System.out.println("Worker stopped.");
      }
}
public class StopWorkerDemo {
      public static void main(String[] args) throws InterruptedException
{
            Worker worker = new Worker();
            Thread t = new Thread(worker);
            t.start();
            Thread.sleep(2000); // let the worker run
            worker.stop();      // stop worker
      }
}
```