Assignment Day-18

Core Java with DS and Algorithms

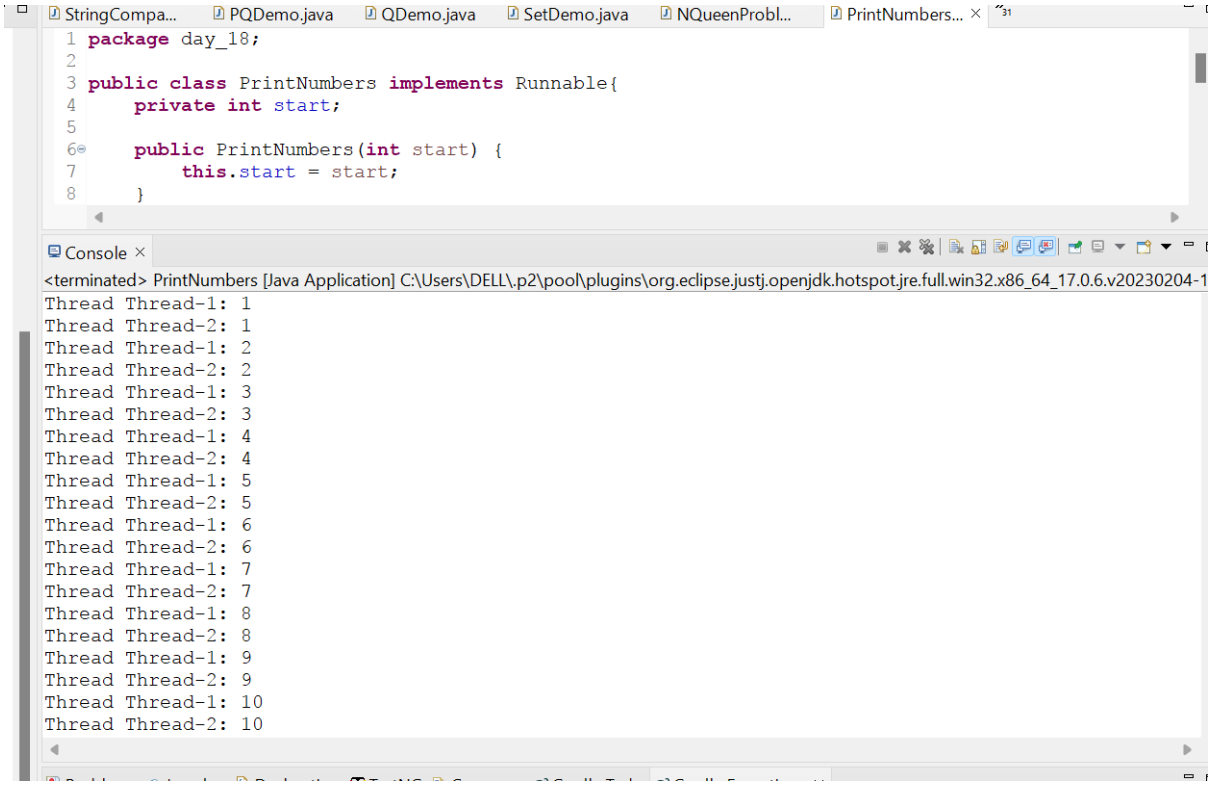
Name: Joshnitha Rangolu

Task 1: Creating and Managing Threads

Write a program that starts two threads, where each thread prints numbers from 1 to 10 with a 1-second delay between each number.

```
package day_18;

public class PrintNumbers implements Runnable{

private int start;

public PrintNumbers(int start) {

this.start = start;

}

@Override

public void run() {

for (int i = start; i <= start + 9; i++) {

System.out.println("Thread " + Thread.currentThread().getName() + ": " +
i);

try {

Thread.sleep(1000); // Sleep for 1 second

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void main(String[] args) {

Thread thread1 = new Thread(new PrintNumbers(1));

Thread thread2 = new Thread(new PrintNumbers(1));
```

```
thread1.setName("Thread-1");

thread2.setName("Thread-2");

thread1.start();

thread2.start();

}

}
```



Task 2: States and Transitions

Create a Java class that simulates a thread going through different lifecycle states: NEW, RUNNABLE, WAITING, TIMED_WAITING, BLOCKED, and TERMINATED. Use methods like sleep(), wait(), notify(), and join() to demonstrate these states..

```
package day_18;

public class ThreadStateSimulator implements Runnable{

private final Object lock = new Object();

@Override

public void run() {
```

```
System.out.println("State: NEW - Thread created");
System.out.println("State: RUNNABLE - Thread started");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
synchronized (lock) {
System.out.println("State: WAITING - Waiting on lock");
try {
lock.wait();
} catch (InterruptedException e) {
e.printStackTrace();
}
}
System.out.println("State: TIMED_WAITING - Waiting for 2 seconds");
try {
synchronized (lock) {
lock.wait(2000);
}
} catch (InterruptedException e) {
e.printStackTrace();
}
try {
System.out.println("State: BLOCKED - Waiting to call notify()");
synchronized (lock) {
lock.wait();
}
```

```
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("State: RUNNABLE - More work after unblock");
try {
Thread.sleep(500);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("State: TERMINATED - Thread completed");
}
public static void main(String[] args) throws InterruptedException {
ThreadStateSimulator thread = new ThreadStateSimulator();
Thread simulatorThread = new Thread(thread);
simulatorThread.start();
Thread.sleep(1500);
synchronized (thread.lock) {
System.out.println("Main: Notifying the thread");
thread.lock.notify();
}
simulatorThread.join();
System.out.println("Main: Thread joined (terminated)");
}
}
```

```java
package day_18;

public class ThreadStateSimulator implements Runnable{
    private final Object lock = new Object();

    @Override
    public void run() {
        System.out.println("State: NEW - Thread created");
        System.out.println("State: RUNNABLE - Thread started");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (lock) {
            System.out.println("State: WAITING - Waiting on lock");
            try {
                lock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
ThreadStateSimulator [Java Application] C:\Users\DELL\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\b
State: NEW - Thread created
State: RUNNABLE - Thread started
State: WAITING - Waiting on lock
Main: Notifying the thread
State: TIMED_WAITING - Waiting for 2 seconds
State: BLOCKED - Waiting to call notify()
```

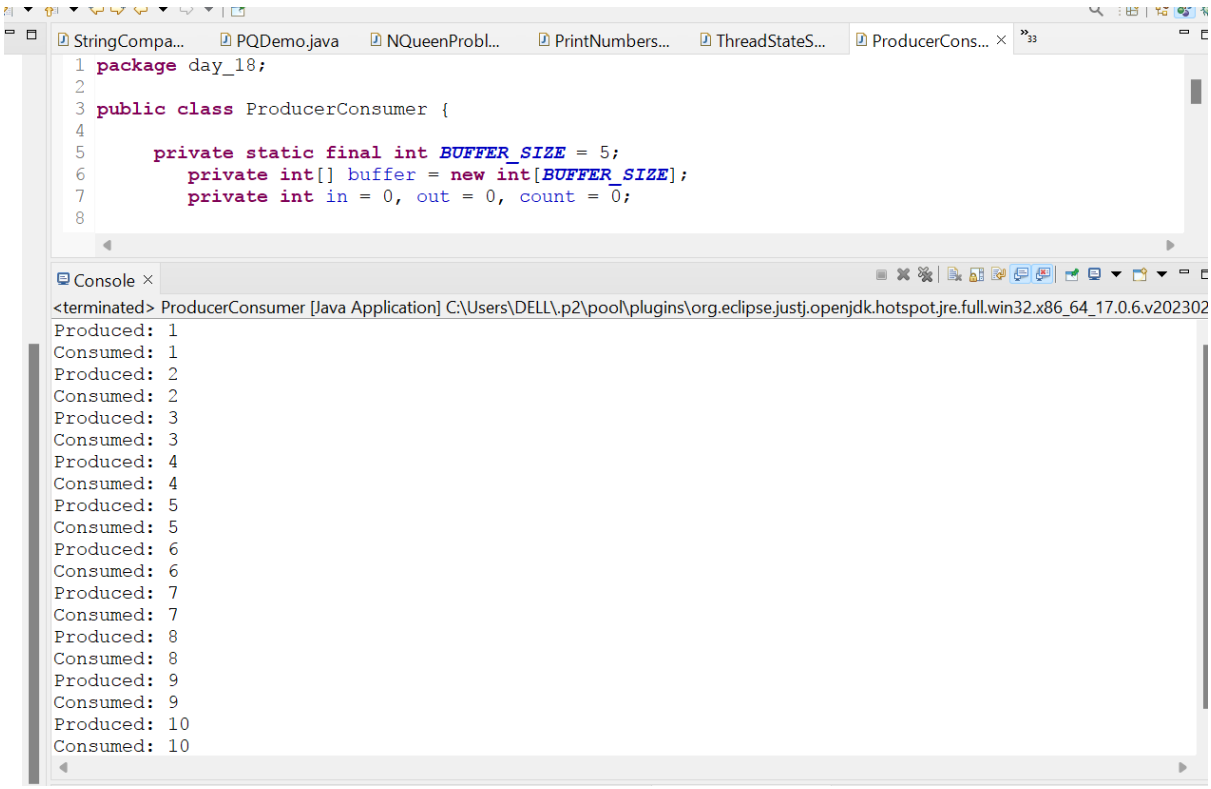
Task 3: Synchronization and Inter-thread Communication

Implement a producer-consumer problem using wait() and notify() methods to handle the correct processing sequence between threads.

```java
package day_18;

public class ProducerConsumer {

private static final int BUFFER_SIZE = 5;

private int[] buffer = new int[BUFFER_SIZE];

private int in = 0, out = 0, count = 0;

public synchronized void produce(int item) throws InterruptedException {

while (count == BUFFER_SIZE) {

wait();

}

buffer[in] = item;

in = (in + 1) % BUFFER_SIZE;

count++;
```

```
notify();
System.out.println("Produced: " + item);
}
public synchronized int consume() throws InterruptedException {
while (count == 0) {
wait();
}
int item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
count--;
notify();
System.out.println("Consumed: " + item);
return item;
}
public static void main(String[] args) throws InterruptedException {
ProducerConsumer pc = new ProducerConsumer();
Thread producer = new Thread(() -> {
for (int i = 1; i <= 10; i++) {
try {
pc.produce(i);
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
});
Thread consumer = new Thread(() -> {
for (int i = 1; i <= 10; i++) {
```

```
try {
pc.consume();
Thread.sleep(500);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
});
producer.start();
consumer.start();
producer.join();
consumer.join();
}
}
```

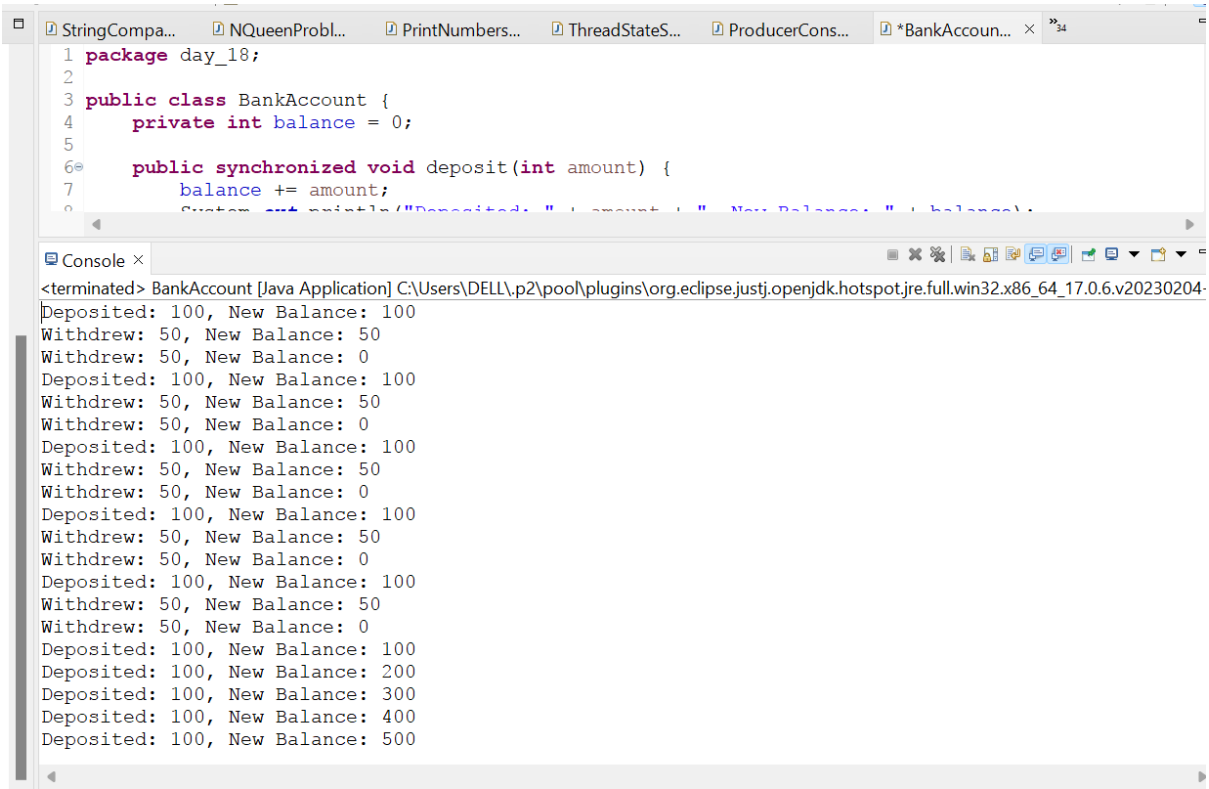


Task 4: Synchronized Blocks and Methods

Write a program that simulates a bank account being accessed by multiple threads to perform deposits and withdrawals using synchronized methods to prevent race conditions.

```
package day_18;

public class BankAccount {

private int balance = 0;

public synchronized void deposit(int amount) {

balance += amount;

System.out.println("Deposited: " + amount + ", New Balance: " + balance);

}

public synchronized void withdraw(int amount) throws InterruptedException {

if (balance < amount) {

System.out.println("Insufficient balance. Waiting for deposit...");

wait();

}

balance -= amount;

System.out.println("Withdrew: " + amount + ", New Balance: " + balance);

}

public static void main(String[] args) throws InterruptedException {

BankAccount account = new BankAccount();

Thread depositor = new Thread(() -> {

for (int i = 0; i < 10; i++) {

account.deposit(100);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}
```

```
}
});
Thread withdrawer = new Thread(() -> {
for (int i = 0; i < 10; i++) {
try {
account.withdraw(50);
Thread.sleep(500);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
});
depositor.start();
withdrawer.start();
depositor.join();
withdrawer.join();
}
}
```



Task 5: Thread Pools and Concurrency Utilities

Create a fixed-size thread pool and submit multiple tasks that perform complex calculations or I/O operations and observe the execution.

```
package day_18;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class FixedThreadPoolDemo {

public static void main(String[] args) throws InterruptedException {

int poolSize = 3;

ExecutorService executor = Executors.newFixedThreadPool(poolSize);

Runnable task = () -> {

long startTime = System.currentTimeMillis();

System.out.println("Thread " + Thread.currentThread().getName() + " started
calculation");
```

```java
try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

long endTime = System.currentTimeMillis();

System.out.println("Thread " + Thread.currentThread().getName() + "
finished calculation in " + (endTime - startTime) + " ms");

};

for (int i = 0; i < 10; i++) {

executor.submit(task);

}

executor.shutdown();

executor.awaitTermination(1, TimeUnit.MINUTES);

System.out.println("All tasks submitted");

}

}
```

```java
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class FixedThreadPoolDemo {
    public static void main(String[] args) throws InterruptedException {
        int poolSize = 3;
        ExecutorService executor = Executors.newFixedThreadPool(poolSize);
        Runnable task = () -> {
            long startTime = System.currentTimeMillis();
            System.out.println("Thread " + Thread.currentThread().getName() + " started cal
            try {
                Thread.sleep(2000);
```

Console ×

<terminated> FixedThreadPoolDemo [Java Application] C:\Users\DELL\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v202

```
Thread pool-1-thread-3 finished calculation in 2003 ms
Thread pool-1-thread-3 started calculation
Thread pool-1-thread-1 finished calculation in 2001 ms
Thread pool-1-thread-1 started calculation
Thread pool-1-thread-2 finished calculation in 2015 ms
Thread pool-1-thread-2 started calculation
Thread pool-1-thread-3 finished calculation in 2014 ms
Thread pool-1-thread-3 started calculation
Thread pool-1-thread-1 finished calculation in 2015 ms
Thread pool-1-thread-1 started calculation
Thread pool-1-thread-2 finished calculation in 2002 ms
Thread pool-1-thread-3 finished calculation in 2013 ms
Thread pool-1-thread-1 finished calculation in 2012 ms
All tasks submitted
```

Task 6: Executors, Concurrent Collections, CompletableFuture

Use an ExecutorService to parallelize a task that calculates prime numbers up to a given number and then use CompletableFuture to write the results to a file asynchronously.

```
package day_18;

import java.io.FileWriter;

import java.io.IOException;

import java.util.ArrayList;

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class PrimeNumberParallel {

private static final int MAX_NUMBER = 1000;

public static boolean isPrime(int num) {

if (num <= 1) return false;

if (num <= 3) return true;

if (num % 2 == 0 || num % 3 == 0) return false;

for (int i = 5; i * i <= num; i += 6) {

if (num % i == 0 || num % (i + 2) == 0) {

return false;

}

}

return true;

}

public static void main(String[] args) throws IOException,
InterruptedException {

int numThreads = Runtime.getRuntime().availableProcessors();

ExecutorService executor = Executors.newFixedThreadPool(numThreads);

ArrayList<Integer> primes = new ArrayList<>();
```

```
CompletableFuture<Void> task = CompletableFuture.runAsync(() -> {
for (int i = 2; i <= MAX_NUMBER; i++) {
if (isPrime(i)) {
primes.add(i);
}
}
}, executor);
task.thenAcceptAsync(unused -> {
try (FileWriter writer = new FileWriter("primes.txt")) {
for (int prime : primes) {
writer.write(prime + "\n");
}
writer.flush();
System.out.println("Primes written to primes.txt");
} catch (IOException e) {
e.printStackTrace();
}
}, executor);
executor.shutdown();
executor.awaitTermination(1, TimeUnit.MINUTES);
System.out.println("Prime number calculation finished");
}
}
```

```java
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class PrimeNumberParallel {
    private static final int MAX_NUMBER = 1000;

    public static boolean isPrime(int num) {
        if (num <= 1) return false;
        if (num <= 3) return true;
        if (num % 2 == 0 || num % 3 == 0) return false;
        for (int i = 5; i * i <= num; i += 6) {
            if (num % i == 0 || num % (i + 2) == 0) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) throws IOException, InterruptedException {
        int numThreads = Runtime.getRuntime().availableProcessors();
        ExecutorService executor = Executors.newFixedThreadPool(numThreads);
        ArrayList<Integer> primes = new ArrayList<>();
        CompletableFuture<Void> task = CompletableFuture.runAsync(() -> {
            for (int i = 2; i <= MAX_NUMBER; i++) {
                if (isPrime(i)) {
```

```
Console ×
<terminated> PrimeNumberParallel [Java Application] C:\Users\DELL\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v202302
Primes written to primes.txt
Prime number calculation finished
```

Task 7: Writing Thread-Safe Code, Immutable Objects

Design a thread-safe Counter class with increment and decrement methods. Then demonstrate its usage from multiple threads. Also, implement and use an immutable class to share data between threads.

```java
package day_18;

import java.util.concurrent.atomic.AtomicInteger;

public class ThreadSafeCounter {

public static void main(String[] args) throws InterruptedException {

Counter counter = new Counter();

Thread incrementer1 = new Thread(() -> {

for (int i = 0; i < 1000; i++) {

counter.increment();

}

});

Thread incrementer2 = new Thread(() -> {

for (int i = 0; i < 1000; i++) {
```

```
counter.increment();
}
});
Thread decrementer = new Thread(() -> {
for (int i = 0; i < 1000; i++) {
counter.decrement();
}
});
incrementer1.start();
incrementer2.start();
decrementer.start();
incrementer1.join();
incrementer2.join();
decrementer.join();
System.out.println("Final count: " + counter.getCount());
}
}
class Counter {
private final AtomicInteger count = new AtomicInteger(0);
public void increment() {
count.incrementAndGet();
}
public void decrement() {
count.decrementAndGet();
}
public int getCount() {
return count.get();
}
```

```
}

class Point {

private final int x;

private final int y;

public Point(int x, int y) {

this.x = x;

this.y = y;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

}
```

```
package day_18;

import java.util.concurrent.atomic.AtomicInteger;

public class ThreadSafeCounter {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Thread incrementer1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread incrementer2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread decrementer = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.decrement();
            }
        });

        incrementer1.start();
```

Console

```
<terminated> ThreadSafeCounter [Java Application] C:\Users\DELL\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v2023
Final count: 1000
```