Manjula Nannuri

Day 18_Assignment

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 com.example;

class ManagingThreads implements Runnable {

private String threadName;

public ManagingThreads(String threadName) {

this.threadName = threadName;

}

@Override

public void run() {

for (int i = 1; i <= 10; i++) {

System.out.println(threadName + ": " + i);

try {

Thread.sleep(1000); // Delay of 1 second

} catch (InterruptedException e) {

System.out.println(threadName + " interrupted.");

}

}

System.out.println(threadName + " has finished.");

}

public static void main(String[] args) {

Thread thread1 = new Thread(new ManagingThreads("Thread 1"));
```

```
Thread thread2 = new Thread(new ManagingThreads("Thread 2"));

thread1.start();

thread2.start();

}

}
```

OUTPUT:

```
<terminated> ManagingThreads [Java Application] C:\Program Files\Java\jdk-17\bin\javaw
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 2: 3
Thread 1: 3
Thread 2: 4
Thread 1: 4
Thread 1: 5
Thread 2: 5
Thread 1: 6
Thread 2: 6
Thread 1: 7
Thread 2: 7
Thread 1: 8
Thread 2: 8
Thread 1: 9
Thread 2: 9
Thread 1: 10
Thread 2: 10
Thread 2 has finished.
Thread 1 has finished.
```

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 com.example;

class ThreadStatesDemo {

private static final Object lock = new Object();
```

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

Thread newThread = new Thread(new StateDemoRunnable(), "DemoThread");

System.out.println(newThread.getName() + " - State: " + newThread.getState());

newThread.start();

System.out.println(newThread.getName() + " - State: " + newThread.getState());

Thread.sleep(100);

synchronized (lock) {

lock.notify();

}

Thread.sleep(100);

System.out.println(newThread.getName() + " - State: " + newThread.getState());

newThread.join();

System.out.println(newThread.getName() + " - State: " + newThread.getState());

}

private static class StateDemoRunnable implements Runnable {

@Override

public void run() {

try {

System.out.println(Thread.currentThread().getName() + " - Going to sleep
(TIMED_WAITING)...");

Thread.sleep(200);

synchronized (lock) {

System.out.println(Thread.currentThread().getName() + " - Waiting for notification
(WAITING)...");

lock.wait();
```

```
}

synchronized (lock) {

System.out.println(Thread.currentThread().getName() + " - Acquired lock (RUNNABLE or BLOCKED)...");

}

System.out.println(Thread.currentThread().getName() + " - Finished execution (TERMINATED)");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}
```

OUTPUT:

```
ThreadStatesDemo [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe  (09-Jun-2024, 8:
DemoThread - State: NEW
DemoThread - State: RUNNABLE
DemoThread - Going to sleep (TIMED_WAITING)...
DemoThread - Waiting for notification (WAITING)...
DemoThread - State: WAITING
```

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.

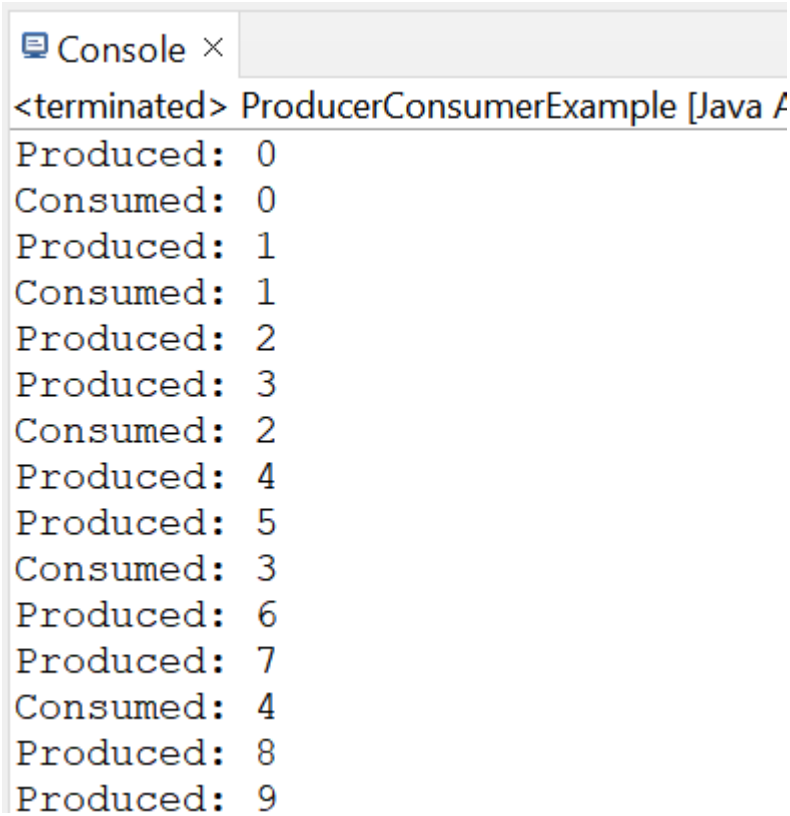
```
package com.example;

import java.util.LinkedList;

import java.util.Queue;

class Buffer {

private final int maxSize;
```

```
private final Queue<Integer> queue;

public Buffer(int maxSize) {

this.maxSize = maxSize;

this.queue = new LinkedList<>();

}

public synchronized void produce(int value) throws InterruptedException {

while (queue.size() == maxSize) {

wait();

}

queue.add(value);

System.out.println("Produced: " + value);

notifyAll();

}

public synchronized int consume() throws InterruptedException {

while (queue.isEmpty()) {

wait();

}

int value = queue.poll();

System.out.println("Consumed: " + value);

notifyAll();

return value;

}

}

class Producer implements Runnable {
```

```
private final Buffer buffer;

public Producer(Buffer buffer) {

this.buffer = buffer;

}

@Override

public void run() {

try {

int value = 0;

while (true) {

buffer.produce(value++);

Thread.sleep(500);

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

class Consumer implements Runnable {

private final Buffer buffer;

public Consumer(Buffer buffer) {

this.buffer = buffer;

}

@Override

public void run() {
```

```
try {
while (true) {
buffer.consume();
Thread.sleep(1000);
}
} catch (InterruptedException e) {
Thread.currentThread().interrupt();
}
}
}
public class ProducerConsumerExample {
public static void main(String[] args) {
Buffer buffer = new Buffer(5);
Thread producerThread = new Thread(new Producer(buffer));
Thread consumerThread = new Thread(new Consumer(buffer));
producerThread.start();
consumerThread.start();
}
}
```

OUTPUT:



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 com.example;

class BankAccount1 {

private double balance;

public BankAccount1(double initialBalance) {

this.balance = initialBalance;

}

public synchronized void deposit(double amount) {

balance += amount;

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

}
```

```
public synchronized void withdraw(double amount) {
if (balance >= amount) {
balance -= amount;
System.out.println("Withdrawn: " + amount + ", Balance: " + balance);
} else {
System.out.println("Insufficient funds for withdrawal: " + amount + ", Balance: " + balance);
}
}
}
class AccountHolder implements Runnable {
private BankAccount1 account;
public AccountHolder(BankAccount1 account) {
this.account = account;
}
@Override
public void run() {
for (int i = 0; i < 5; i++) {
account.deposit(100);
account.withdraw(50);
}
}
}
public class BankAccount{
public static void main(String[] args) {
```

```
BankAccount1 account = new BankAccount1(1000);

Thread thread1 = new Thread(new AccountHolder(account));

Thread thread2 = new Thread(new AccountHolder(account));

thread1.start();

thread2.start();

}

}
```

OUTPUT:

```
<terminated> BankAccount [Java Application] C:\Program Files\Java\j
Deposited: 100.0, Balance: 1100.0
Withdrawn: 50.0, Balance: 1050.0
Deposited: 100.0, Balance: 1150.0
Withdrawn: 50.0, Balance: 1100.0
Deposited: 100.0, Balance: 1200.0
Withdrawn: 50.0, Balance: 1150.0
Deposited: 100.0, Balance: 1250.0
Withdrawn: 50.0, Balance: 1200.0
Deposited: 100.0, Balance: 1300.0
Withdrawn: 50.0, Balance: 1250.0
Deposited: 100.0, Balance: 1350.0
Withdrawn: 50.0, Balance: 1300.0
Deposited: 100.0, Balance: 1400.0
Withdrawn: 50.0, Balance: 1350.0
Deposited: 100.0, Balance: 1450.0
Withdrawn: 50.0, Balance: 1400.0
Deposited: 100.0, Balance: 1500.0
Withdrawn: 50.0, Balance: 1450.0
Deposited: 100.0, Balance: 1550.0
Withdrawn: 50.0, Balance: 1500.0
```

Task 5: Thread Pools and Concurrency Utilities

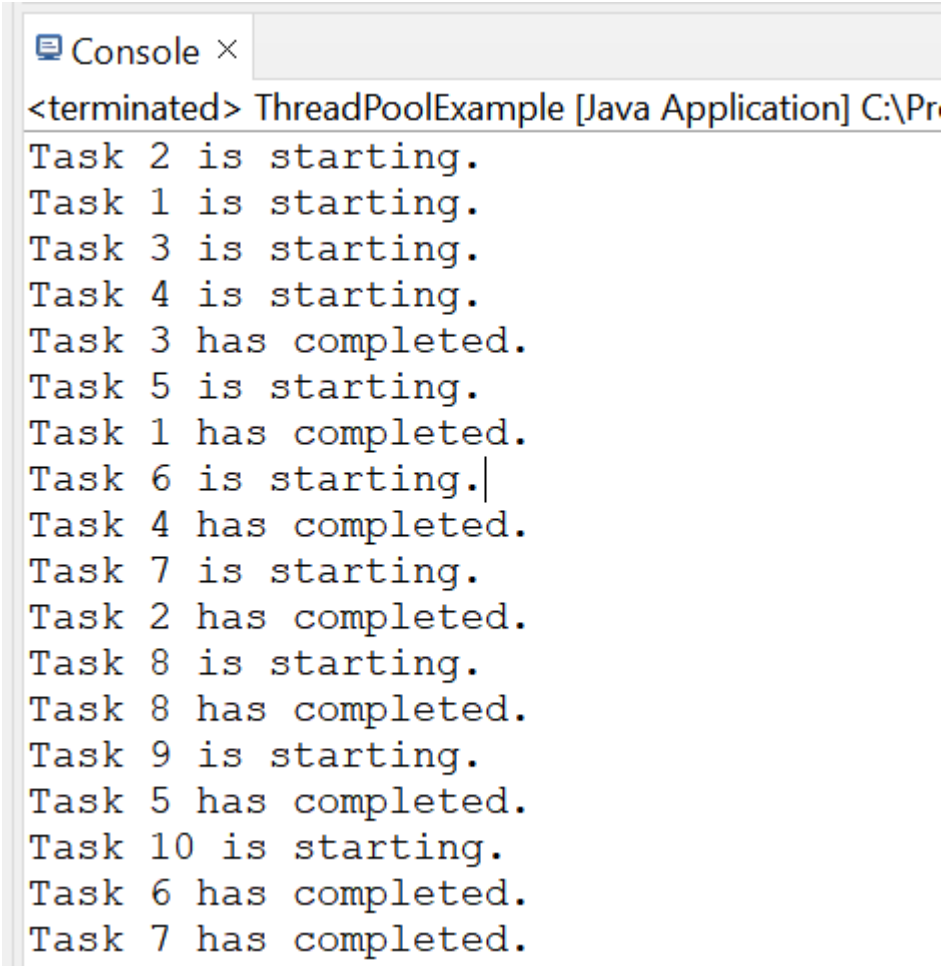
Create a fixed-size thread pool and submit multiple tasks that perform complex calculations or I/O operations and observe the execution.

```
package com.example;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

class ComplexTask implements Runnable {

    private final int taskId;

    public ComplexTask(int taskId) {

        this.taskId = taskId;

    }

    @Override

    public void run() {

        System.out.println("Task " + taskId + " is starting.");

        performComplexCalculation();

        System.out.println("Task " + taskId + " has completed.");

    }

    private void performComplexCalculation() {

        try {
```

```
            Thread.sleep((long) (Math.random() * 1000) + 500);
        } catch (InterruptedException e) {
            System.out.println("Task " + taskId + " was interrupted.");
        }
    }
}
public class ThreadPoolExample {
    public static void main(String[] args) {
        int numberOfTasks = 10;
        int poolSize = 4;
        ExecutorService executorService = Executors.newFixedThreadPool(poolSize);
        for (int i = 1; i <= numberOfTasks; i++) {
            executorService.submit(new ComplexTask(i));
        }
        executorService.shutdown();
        try {
            if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            executorService.shutdownNow();
        }
        System.out.println("All tasks have completed.");
    }
```

}

OUTPUT:


Console
<terminated> ThreadPoolExample [Java Application] C:\Pr
Task 2 is starting.
Task 1 is starting.
Task 3 is starting.
Task 4 is starting.
Task 3 has completed.
Task 5 is starting.
Task 1 has completed.
Task 6 is starting.
Task 4 has completed.
Task 7 is starting.
Task 2 has completed.
Task 8 is starting.
Task 8 has completed.
Task 9 is starting.
Task 5 has completed.
Task 10 is starting.
Task 6 has completed.
Task 7 has completed.

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 com.example;

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class PrimeNumberCalculator {

public static boolean isPrime(int number) {

if (number <= 1) return false;

if (number == 2) return true;

if (number % 2 == 0) return false;

for (int i = 3; i <= Math.sqrt(number); i += 2) {

if (number % i == 0) return false;

}

return true;

}
```

```
public static List<Integer> calculatePrimes(int maxNumber, int numThreads) throws
Exception {

ExecutorService executorService = Executors.newFixedThreadPool(numThreads);

List<Future<List<Integer>>> futures = new ArrayList<>();

int chunkSize = maxNumber / numThreads;

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

int start = i * chunkSize + 1;

int end = (i == numThreads - 1) ? maxNumber : (i + 1) * chunkSize;

futures.add(executorService.submit(() -> {

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

for (int j = start; j <= end; j++) {

if (isPrime(j)) {

primes.add(j);

}

}

return primes;

}));

}

List<Integer> allPrimes = new ArrayList<>();

for (Future<List<Integer>> future : futures) {

allPrimes.addAll(future.get());

}

executorService.shutdown();

return allPrimes;

}
```

```
public static CompletableFuture<Void> writePrimesToFile(List<Integer> primes, String
filename) {

return CompletableFuture.runAsync(() -> {

try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {

for (int prime : primes) {

writer.write(prime + "\n");

}

} catch (IOException e) {

e.printStackTrace();

}

});

}

public static void main(String[] args) {

int maxNumber = 100000;

int numThreads = 4;

String filename = "primes.txt";

try {

List<Integer> primes = calculatePrimes(maxNumber, numThreads);

CompletableFuture<Void> future = writePrimesToFile(primes, filename);

future.thenRun(() -> System.out.println("Prime numbers have been written to " +
filename));

future.get();

} catch (Exception e) {

e.printStackTrace();

}
```

```
}

}
```

OUTPUT:

```
<terminated> PrimeNumberCalculator [Java Application] C:\Program Files\Java\jdk
Prime numbers have been written to primes.txt
```

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.

```
package com.example;

class Counter {

private int count;

public Counter() {

this.count = 0;

}

public synchronized void increment() {

count++;

}

public synchronized void decrement() {

count--;

}

public synchronized int getCount() {

return count;

}
```

```
}

package com.example;

final class ImmutableData {

private final int value;

public ImmutableData(int value) {

this.value = value;

}

public int getValue() {

return value;

}

}

package com.example;

public class ThreadSafeDemo {

public static void main(String[] args) {

Counter counter = new Counter();

Thread incrementThread1 = new Thread(new CounterIncrementer(counter));

Thread incrementThread2 = new Thread(new CounterIncrementer(counter));

Thread decrementThread = new Thread(new CounterDecrementer(counter));

incrementThread1.start();

incrementThread2.start();

decrementThread.start();

try {

incrementThread1.join();

incrementThread2.join();
```

```
decrementThread.join();
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("Final count: " + counter.getCount());
ImmutableData immutableData = new ImmutableData(42);
Thread dataPrinter1 = new Thread(new DataPrinter(immutableData));
Thread dataPrinter2 = new Thread(new DataPrinter(immutableData));
dataPrinter1.start();
dataPrinter2.start();
}
}
class CounterIncrementer implements Runnable {
private Counter counter;
public CounterIncrementer(Counter counter) {
this.counter = counter;
}
@Override
public void run() {
for (int i = 0; i < 10; i++) {
counter.increment();
try {
Thread.sleep(100); // Simulate some work
} catch (InterruptedException e) {
```

```
e.printStackTrace();
}
}
}
}
class CounterDecrementer implements Runnable {
private Counter counter;
public CounterDecrementer(Counter counter) {
this.counter = counter;
}
@Override
public void run() {
for (int i = 0; i < 10; i++) {
counter.decrement();
try {
Thread.sleep(100);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
}
class DataPrinter implements Runnable {
private ImmutableData data;
```

```
public DataPrinter(ImmutableData data) {

this.data = data;

}

@Override

public void run() {

System.out.println("ImmutableData value: " + data.getValue());

}

}
```

OUTPUT:

```
<terminated> ThreadSafeDemo [Java Applicat
Final count: 10
ImmutableData value: 42
ImmutableData value: 42
```