

Assignment – 19

Prakash Manikanta Irrinki
Prakashnaidu9494@gmail.com

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.

Program:

```
class MyThread extends Thread{

    @Override
    public void run() {
        for(int i=1;i<=10;i++) {
            System.out.print(i+" ");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println();
    }

}

public class CreatingAndManagingThreads {

    public static void main(String[] args) {
        MyThread thread1=new MyThread();
        MyThread thread2=new MyThread();

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Both threads have finished!");
    }

}
```

Output:

1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10

Both threads have 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.

Program:

```
public class StatesAndTransitionsInThreads {
    public static void main(String[] args) {
        MyThreadClass thread = new MyThreadClass();
        thread.start();
        try {
            Thread.sleep(1000);
            synchronized (thread) {
                thread.notifyAll();
            }
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Main thread is eisting!");
    }
}

class MyThreadClass extends Thread {
    public void run() {
        try {
            System.out.println("Thread is in NEW state");
            System.out.println("Thread is in RUNNABLE state");
            Thread.sleep(3000);
            System.out.println("Thread is in TIMED_WAITING state");
            Thread.sleep(2000);
            synchronized (this) {
                System.out.println("Thread is in WAITING state");
                wait(2000);
            }
            System.out.println("Thread is in BLOCKED state");
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread is in TERMINATED state");
    }
}
```

Output:

```
Thread is in NEW state
Thread is in RUNNABLE state
Thread is in TIMED_WAITING state
Thread is in WAITING state
Thread is in BLOCKED state
Thread is in TERMINATED state
Main thread is eisting!
```

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.

Program:

```
import java.util.LinkedList;
public class Producer extends Thread{
    private final LinkedList<Integer> buffer;
    private final int capacity;
    private int value=0;
    public Producer(LinkedList<Integer> buffer, int capacity) {
        this.buffer = buffer;
        this.capacity = capacity;
    }
    public void run() {
        while(true) {
            try {
                synchronized (buffer) {
                    while(buffer.size() == capacity) {
                        System.out.println("Buffer is full.
                        Producer is waiting...");
                        buffer.wait();
                    }
                    System.out.println("Producer produced:
                        "+value);
                    buffer.add(value++);
                    buffer.notify();
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Consumer extends Thread {

    private final LinkedList<Integer> buffer;

    public Consumer(LinkedList<Integer> buffer) {
        this.buffer = buffer;
    }
    public void run() {
        while (true) {
            try {
                synchronized (buffer) {
                    while (buffer.isEmpty()) {
                        System.out.println("Buffer is
                        empty. Consumer is waiting...");
                        buffer.wait();
                    }
                }
                int value = buffer.removeFirst();
                System.out.println("Consumer consumed: "
                    + value);
            }
        }
    }
}
```


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.

Program:

```
public class SynchronizedBlocksAndMethods {
    private double balance;
    public SynchronizedBlocksAndMethods(double initialBalance) {
        this.balance = initialBalance;
    }
    public synchronized void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited: " + amount + ",
                           New Balance: " + balance);
    }
    public synchronized void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount + ",
                               New Balance: " + balance);
        } else {
            System.out.println("Insufficient funds for
                               withdrawal: " + amount);
        }
    }
    public static void main(String[] args) {
        SynchronizedBlocksAndMethods account = new
            SynchronizedBlocksAndMethods(1000);
        Thread depositThread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                account.deposit(100);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        Thread withdrawThread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                account.withdraw(200);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        depositThread.start();
        withdrawThread.start();
        try {
            depositThread.join();
            withdrawThread.join();
        }
    }
}
```

```

        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Available Balance: "
                           + account.balance);
    }
}

```

Output:

```

Deposited: 100.0, New Balance: 1100.0
Withdrawn: 200.0, New Balance: 900.0
Withdrawn: 200.0, New Balance: 700.0
Deposited: 100.0, New Balance: 800.0
Withdrawn: 200.0, New Balance: 600.0
Deposited: 100.0, New Balance: 700.0
Withdrawn: 200.0, New Balance: 500.0
Deposited: 100.0, New Balance: 600.0
Withdrawn: 200.0, New Balance: 400.0
Deposited: 100.0, New Balance: 500.0
Available Balance: 500.0

```

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.

Program:

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolsAndConcurrencyUtilities {
    public static void main(String[] args) {
        ExecutorService executor =
            Executors.newFixedThreadPool(3);
        for (int i = 0; i < 10; i++) {
            final int taskNumber = i;
            executor.submit(() -> {
                System.out.println("Task " + taskNumber +
                    " started by Thread: " + Thread.currentThread().getName());
                performTask(taskNumber);
                System.out.println("Task " + taskNumber + "
                    completed by Thread: " + Thread.currentThread().getName());
            });
        }
        executor.shutdown();
    }
    private static void performTask(int taskNumber) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Output:

```
Task 0 started by Thread: pool-1-thread-1
Task 1 started by Thread: pool-1-thread-2
Task 2 started by Thread: pool-1-thread-3
Task 0 completed by Thread: pool-1-thread-1
Task 1 completed by Thread: pool-1-thread-2
Task 2 completed by Thread: pool-1-thread-3
Task 3 started by Thread: pool-1-thread-3
Task 4 started by Thread: pool-1-thread-2
Task 5 started by Thread: pool-1-thread-1
Task 3 completed by Thread: pool-1-thread-3
Task 6 started by Thread: pool-1-thread-3
Task 4 completed by Thread: pool-1-thread-2
Task 7 started by Thread: pool-1-thread-2
Task 5 completed by Thread: pool-1-thread-1
Task 8 started by Thread: pool-1-thread-1
Task 6 completed by Thread: pool-1-thread-3
Task 9 started by Thread: pool-1-thread-3
Task 7 completed by Thread: pool-1-thread-2
Task 8 completed by Thread: pool-1-thread-1
Task 9 completed by Thread: pool-1-thread-3
```

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.

Program:

```
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;

public class CalculatesPrimeNumbers {
    public static void main(String[] args) {
        int primecount = 100;
        ExecutorService executor =
            Executors.newFixedThreadPool
                (Runtime.getRuntime().availableProcessors());
        CompletableFuture<List<Integer>> primeNumbersFuture =
            CompletableFuture.supplyAsync(() ->
                calculatePrimes(primecount, executor), executor);
        primeNumbersFuture.thenAcceptAsync(primeNumbers ->
            {
                writeToFile(primeNumbers);
            }).thenRun(executor::shutdown);
    }

    public static List<Integer> calculatePrimes(int primecount,
        ExecutorService executor) {
        List<Integer> primes = new ArrayList<>();
        List<CompletableFuture<Void>> futures = new ArrayList<>();
        for (int i = 2; i <= primecount; i++) {
            int n = i;
```

```

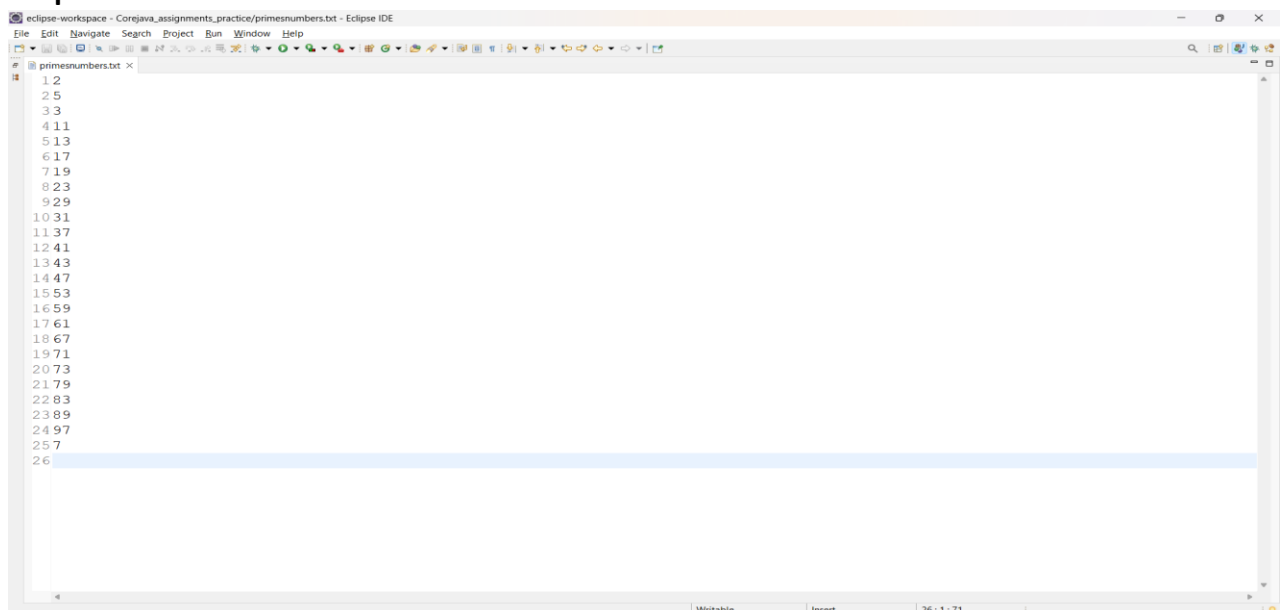
        CompletableFuture<Void> future =
            CompletableFuture.supplyAsync(() -> {
                if (isPrime(n)) {
                    synchronized (primes) {
                        primes.add(n);
                    }
                }
                return null;
            }, executor);
        futures.add(future);
    }
    CompletableFuture<Void> allFutures =
        CompletableFuture.allOf(futures.toArray(
            new CompletableFuture[0]));
    allFutures.join();
    return primes;
}

public static boolean isPrime(int num) {
    if (num <= 1) {
        return false;
    }
    for (int i = 2; i <= Math.sqrt(num); i++) {
        if (num % i == 0) {
            return false;
        }
    }
    return true;
}

public static void writeToFile(List<Integer> primes) {
    try (BufferedWriter writer = new BufferedWriter(new
FileWriter("primesnumbers.txt"))) {
        for (int prime : primes) {
            writer.write(prime + "\n");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

Output:



```

eclipse-workspace - Corejava_assignments_practice/primesnumbers.txt - Eclipse IDE
File Edit Navigate Search Project Run Window Help
# primesnumbers.txt x
12
25
33
411
513
617
719
823
929
1031
1137
1241
1343
1447
1553
1659
1761
1867
1971
2073
2179
2283
2389
2497
257
26

```


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.

Program:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class Counter {

    private int count;
    private final Lock lock;

    public Counter() {
        count = 0;
        lock = new ReentrantLock();
    }

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }

    public void decrement() {
        lock.lock();
        try {
            count--;
        } finally {
            lock.unlock();
        }
    }

    public int getCount() {
        lock.lock();
        try {
            return count;
        } finally {
            lock.unlock();
        }
    }
}

final class Data {
    private final String value;

    public Data(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}
```

