

## Concurrency and Multi-threading - 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

Code:

```
package creatingmanagingthreads;
```

```
public class PrintingNumbersThreads implements Runnable {
```

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

```
        Runnable rTask = new PrintingNumbersThreads();
```

```
        Thread t1 = new Thread(rTask);
```

```
        Thread t2 = new Thread(rTask);
```

```
        t1.start();
```

```
        t2.start();
```

```
    }
```

```
    @Override
```

```
    public void run() {
```

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

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

```
            try {
```

```
                Thread.sleep(1000);
```

```
            } catch (InterruptedException e) {
```

```
                Thread.currentThread().interrupt();
```

```
                System.out.println(Thread.currentThread().getName() + " was interrupted.");
```

```
            } break;
```

```
        }
```

```
    }
```

```
}
```

```
}
```

## Output:

```
Console ×
<terminated> PrintingNumbers
Thread-0: 1
Thread-1: 1
Thread-0: 2
Thread-1: 2
Thread-0: 3
Thread-1: 3
Thread-0: 4
Thread-1: 4
Thread-0: 5
Thread-1: 5
Thread-0: 6
Thread-1: 6
Thread-0: 7
Thread-1: 7
Thread-0: 8
Thread-1: 8
Thread-0: 9
Thread-1: 9
Thread-0: 10
Thread-1: 10
```

## 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..

### Code:

```
package lifecyclestates;

public class LifeCycleStatesDemo {
    private static final Object monitor = new Object();

    public static void main(String[] args) {
        Thread thread = new Thread(new DemoRunnable());
```

```
System.out.println("State after creation: " + thread.getState());
```

```
thread.start();
```

```
System.out.println("State after start(): " + thread.getState());
```

```
sleep(100);
```

```
synchronized (monitor) {
```

```
System.out.println("Notifying the waiting thread.");
```

```
monitor.notify();
```

```
}
```

```
sleep(100);
```

```
try {
```

```
thread.join();
```

```
} catch (InterruptedException e) {
```

```
e.printStackTrace();
```

```
}
```

```
System.out.println("State after join(): " + thread.getState());
```

```
}
```

```
static class DemoRunnable implements Runnable {
```

```
@Override
```

```
public void run() {
```

```
try {
```

```
System.out.println("State in run(): " + Thread.currentThread().getState());
```

```
synchronized (monitor) {
```

```
System.out.println("Entering WAITING state.");
```

```
monitor.wait();
```

```
}
```

```
System.out.println("Transitioning to TIMED_WAITING state.");
```

```
Thread.sleep(1000);
```

```

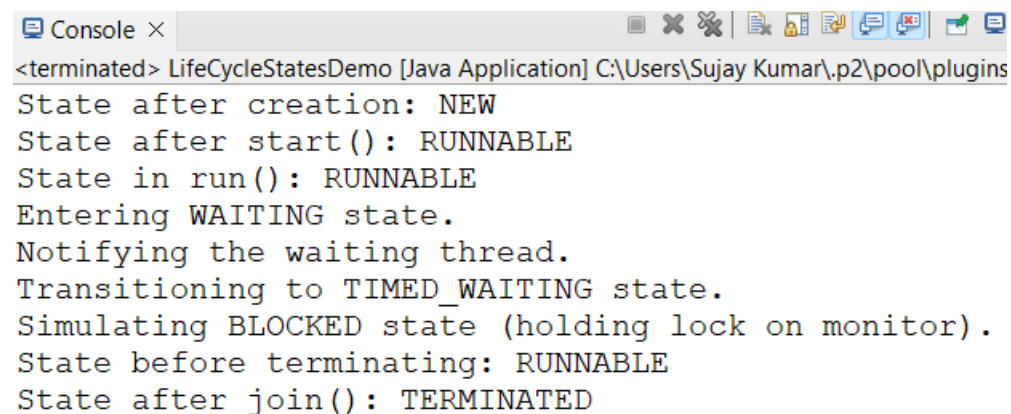
synchronized (monitor) {
    System.out.println("Simulating BLOCKED state (holding lock on monitor).");
}

System.out.println("State before terminating: " + Thread.currentThread().getState());
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}

private static void sleep(int m) {
    try {
        Thread.sleep(m);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
}

```

### Output:



The screenshot shows a console window titled "Console x" with the following output:

```

<terminated> LifeCycleStatesDemo [Java Application] C:\Users\Sujay Kumar\p2\pool\plugins
State after creation: NEW
State after start(): RUNNABLE
State in run(): RUNNABLE
Entering WAITING state.
Notifying the waiting thread.
Transitioning to TIMED_WAITING state.
Simulating BLOCKED state (holding lock on monitor).
State before terminating: RUNNABLE
State after join(): TERMINATED

```

### 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.

**Code:**

```
package producerconsumer;

import java.util.LinkedList;

public class ProducerConsumer {

    private LinkedList<Integer> buffer = new LinkedList<>();
    private final int CAPACITY = 5;

    public void produce() throws InterruptedException {
        int value = 0;
        while (true) {
            synchronized (this) {
                while (buffer.size() == CAPACITY) {
                    wait();
                }
                System.out.println("Producer produced: " + value);
                buffer.add(value++);
                notify();
                Thread.sleep(1000);
            }
        }
    }

    public void consume() throws InterruptedException {
        while (true) {
            synchronized (this) {
                while (buffer.isEmpty()) {
                    wait();
                }
                int consumedValue = buffer.removeFirst();
                System.out.println("Consumer consumed: " + consumedValue);
                notify();
                Thread.sleep(1000);
            }
        }
    }
}
```

```
}  
}
```

```
public static void main(String[] args) {  
    ProducerConsumer pc = new ProducerConsumer();
```

```
    Thread pThread = new Thread(() -> {  
        try {  
            pc.produce();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    });
```

```
    Thread cThread = new Thread(() -> {  
        try {  
            pc.consume();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    });
```

```
    pThread.start();  
    cThread.start();  
}  
}
```

**Output:**

```
Console x
<terminated> ProducerConsumer [Java A
Producer produced: 0
Producer produced: 1
Producer produced: 2
Producer produced: 3
Consumer consumed: 0
Consumer consumed: 1
Consumer consumed: 2
Consumer consumed: 3
Producer produced: 4
Consumer consumed: 4
Producer produced: 5
Producer produced: 6
Producer produced: 7
Producer produced: 8
Producer produced: 9
Consumer consumed: 5
Consumer consumed: 6
Producer produced: 10
Consumer consumed: 7
Consumer consumed: 8
Consumer consumed: 9
Consumer consumed: 10
Producer produced: 11 and output goes on.....
```

#### 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.

Code:

```
package synchronizedbacksystem;

public class BankAccountWithSynchronizing {
    private double balance;

    public BankAccountWithSynchronizing(double initialBalance) {
        this.balance = initialBalance;
    }
}
```

```
public synchronized void deposit(double amount) {  
    balance += amount;  
    System.out.println(Thread.currentThread().getName() + " Deposit: " + amount + ",  
    New Balance: " + balance);  
}
```

```
public synchronized void withdraw(double amount) {  
    if (balance >= amount) {  
        balance -= amount;  
        System.out  
        .println(Thread.currentThread().getName() + " Withdrawal: " + amount + ", New  
        Balance: " + balance);  
    } else {  
        System.out.println("Insufficient funds for withdrawal: " + amount);  
    }  
}
```

```
public static void main(String[] args) {  
    BankAccountWithSynchronizing account = new  
    BankAccountWithSynchronizing(1000);
```

```
    Thread t1 = new Thread(() -> {  
        for (int i = 0; i < 5; i++) {  
            account.deposit(200);  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    });
```

```
    Thread t2 = new Thread(() -> {  
        for (int i = 0; i < 3; i++) {
```



```
account.withdraw(300);  
try {  
    Thread.sleep(1000);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}  
}  
});
```

```
Thread t3 = new Thread() -> {  
    for (int i = 0; i < 4; i++) {  
        account.deposit(500);  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
});
```

```
Thread t4 = new Thread() -> {  
    for (int i = 0; i < 2; i++) {  
        account.withdraw(400);  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
});
```

```
t1.start();  
t2.start();  
t3.start();
```

```

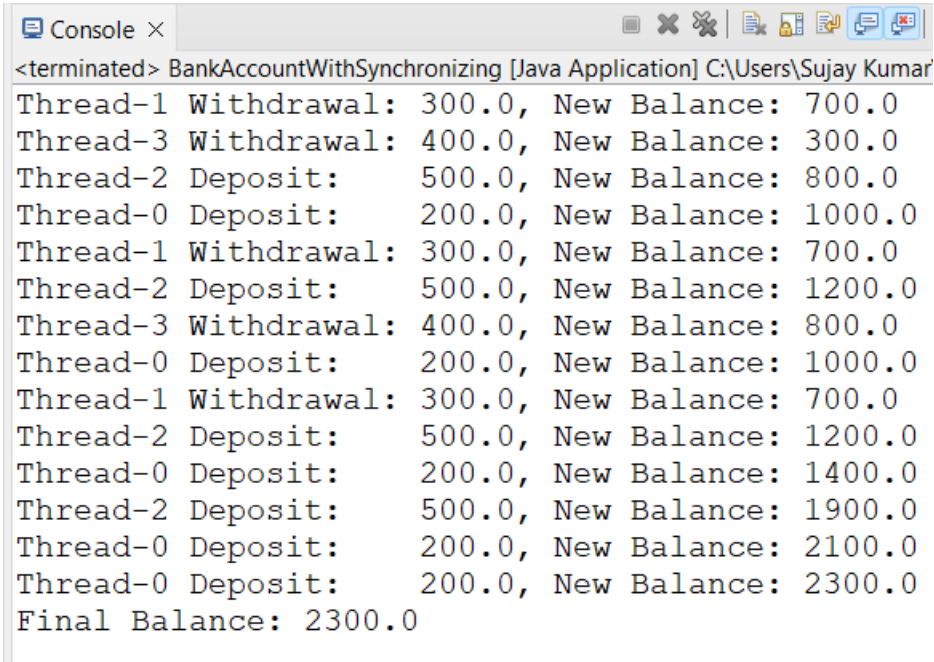
t4.start();

try {
t1.join();
t2.join();
t3.join();
t4.join();
} catch (InterruptedException e) {
e.printStackTrace();
}

System.out.println("Final Balance: " + account.balance);
}
}

```

### Output:



```

<terminated> BankAccountWithSynchronizing [Java Application] C:\Users\Sujay Kumar
Thread-1 Withdrawal: 300.0, New Balance: 700.0
Thread-3 Withdrawal: 400.0, New Balance: 300.0
Thread-2 Deposit: 500.0, New Balance: 800.0
Thread-0 Deposit: 200.0, New Balance: 1000.0
Thread-1 Withdrawal: 300.0, New Balance: 700.0
Thread-2 Deposit: 500.0, New Balance: 1200.0
Thread-3 Withdrawal: 400.0, New Balance: 800.0
Thread-0 Deposit: 200.0, New Balance: 1000.0
Thread-1 Withdrawal: 300.0, New Balance: 700.0
Thread-2 Deposit: 500.0, New Balance: 1200.0
Thread-0 Deposit: 200.0, New Balance: 1400.0
Thread-2 Deposit: 500.0, New Balance: 1900.0
Thread-0 Deposit: 200.0, New Balance: 2100.0
Thread-0 Deposit: 200.0, New Balance: 2300.0
Final Balance: 2300.0

```

### 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.

**Code:**

```
package threadpool;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class ComplexTask implements Runnable {
    private int taskId;

    public ComplexTask(int taskId) {
        this.taskId = taskId;
    }

    @Override
    public void run() {
        System.out.println("Task " + taskId + " started.");
        for (int i = 1; i < 4; i++) {
            System.out.println("Task " + taskId + " processing: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Task " + taskId + " completed.");
    }
}

public class MultipleTasksWithFixedThreadPool {
    public static void main(String[] args) {

        ExecutorService executor = Executors.newFixedThreadPool(3);

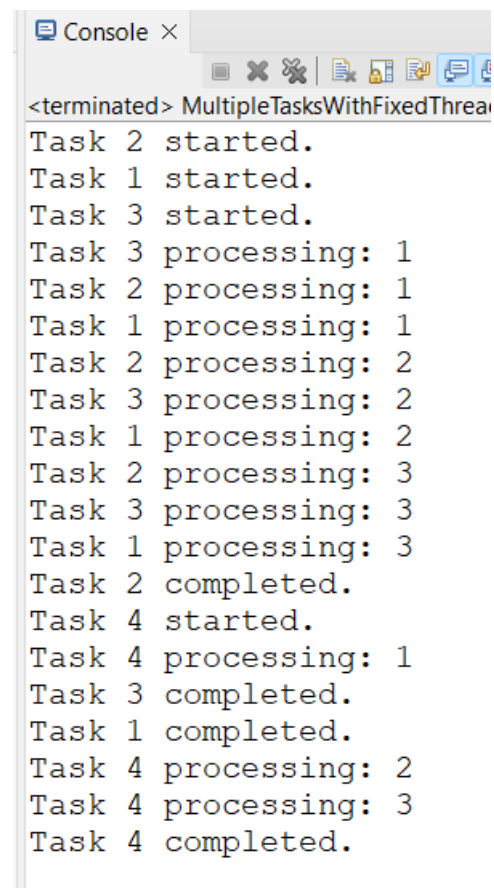
        for (int i = 0; i < 4; i++) {
            executor.submit(new ComplexTask(i + 1));
        }
    }
}
```

```

    }
    executor.shutdown();
}
}

```

### Output:



```

Console X
<terminated> MultipleTasksWithFixedThrea
Task 2 started.
Task 1 started.
Task 3 started.
Task 3 processing: 1
Task 2 processing: 1
Task 1 processing: 1
Task 2 processing: 2
Task 3 processing: 2
Task 1 processing: 2
Task 2 processing: 3
Task 3 processing: 3
Task 1 processing: 3
Task 2 completed.
Task 4 started.
Task 4 processing: 1
Task 3 completed.
Task 1 completed.
Task 4 processing: 2
Task 4 processing: 3
Task 4 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.

### Code:

```

package concurrencyutilities;
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;
```

```
class PrimeCalculator {
    public List<Integer> calculatePrimes(int n) {
        List<Integer> primes = new ArrayList<>();
        for (int i = 2; i <= n; i++) {
            if (isPrime(i)) {
                primes.add(i);
            }
        }
        return primes;
    }
}
```

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

```
public class ExecutorServiceAndCompletableFuture {
    public static void main(String[] args) throws IOException {
```

```

int n = 100;

String outputFile = "primes.txt";

ExecutorService executor = Executors.newFixedThreadPool(4);

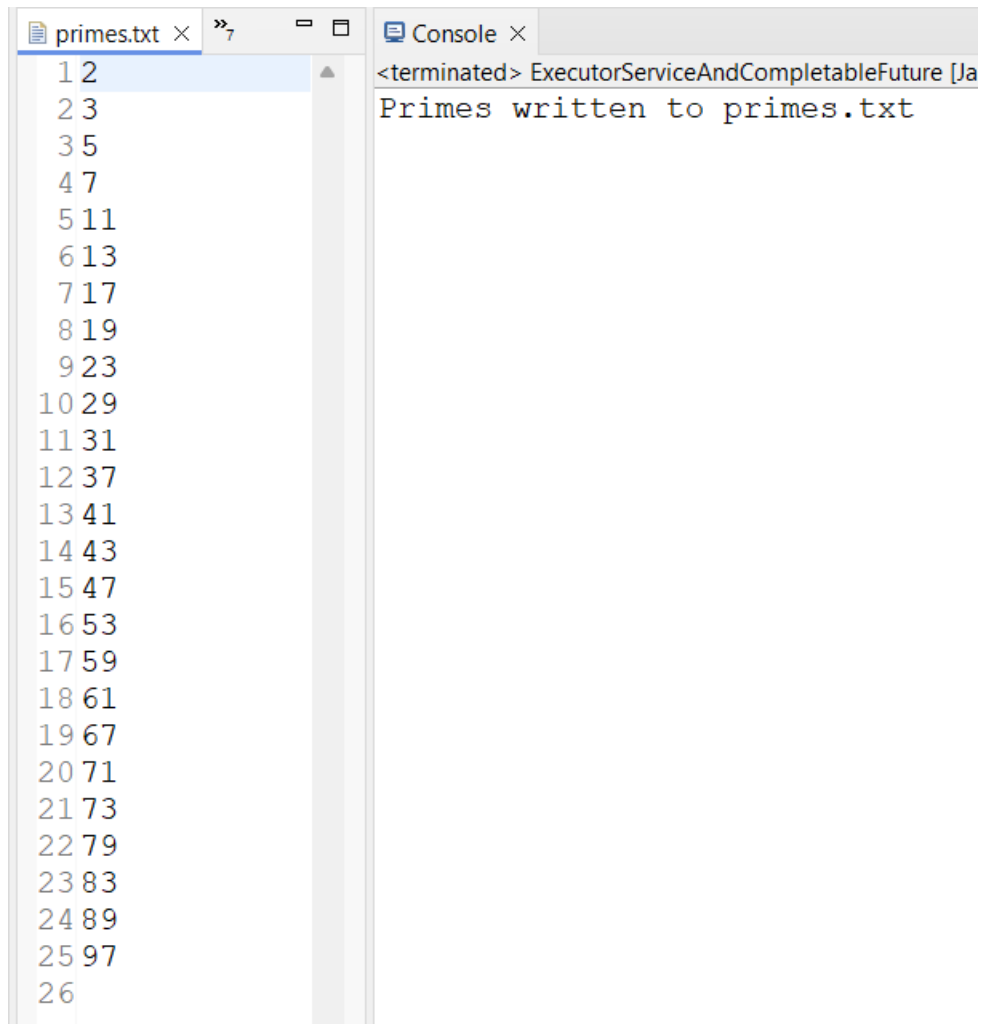
CompletableFuture<List<Integer>> future = CompletableFuture.supplyAsync(() -> {
    PrimeCalculator calculator = new PrimeCalculator();
    return calculator.calculatePrimes(n);
}, executor);

CompletableFuture<Void> writeFuture = future.thenAcceptAsync(primes -> {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
        for (int prime : primes) {
            writer.write(String.valueOf(prime));
            writer.newLine();
        }
        System.out.println("Primes written to " + outputFile);
    } catch (IOException e) {
        e.printStackTrace();
    }
}, executor);

executor.shutdown();
writeFuture.join();
}
}

```

**Output:**



```
primes.txt x 7
1 2
2 3
3 5
4 7
5 11
6 13
7 17
8 19
9 23
10 29
11 31
12 37
13 41
14 43
15 47
16 53
17 59
18 61
19 67
20 71
21 73
22 79
23 83
24 89
25 97
26

Console x
<terminated> ExecutorServiceAndCompletableFuture [Ja
Primes 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.

Counter.java Code:

```
package threadsafe;
```

```
public 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;  
}
```

```
public static void main(String[] args) {  
    Counter counter = new Counter();  
    Thread[] threads = new Thread[5];  
    for (int i = 0; i < threads.length; i++) {  
        threads[i] = new Thread(() -> {  
            for (int j = 0; j < 1000; j++) {  
                counter.increment();  
                counter.decrement();  
            }  
        });  
        threads[i].start();  
    }  
  
    for (Thread thread : threads) {  
        try {  
            thread.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
  
    System.out.println("Final count: " + counter.getCount());  
}
```



```
}
```

### CounterSnapshot.java Code:

```
package threadsafe;
```

```
class CounterSnapshot {
```

```
    private final int count;
```

```
    public CounterSnapshot(int count) {
```

```
        this.count = count;
```

```
    }
```

```
    public int getCount() {
```

```
        return count;
```

```
    }
```

```
}
```

### Output:

