Digvijay Thakare
digvijaythakare2017@gmail.com
Day 18:
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 WiproEP;
class NumberPrinter implements Runnable {
    private String threadName;
    public NumberPrinter(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); // Sleep for 1 second
            } catch (InterruptedException e) {
                System.out.println(threadName + " interrupted.");
            }
        }
        System.out.println(threadName + " finished.");
    }
}
public class ThreadExample {
    public static void main(String[] args) {
        Runnable numberPrinter1 = new NumberPrinter("Thread 1");
        Runnable numberPrinter2 = new NumberPrinter("Thread 2");
        Thread thread1 = new Thread(numberPrinter1);
        Thread thread2 = new Thread(numberPrinter2);
        thread1.start();
        thread2.start();
    }
}
```

Output

```
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 2: 5
Thread 1: 5
Thread 2: 6
Thread 1: 6
Thread 2: 7
Thread 1: 7
Thread 2: 8
Thread 1: 8
Thread 2: 9
Thread 1: 9
Thread 2: 10
Thread 1: 10
Thread 2 finished.
Thread 1 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.

Code

```java
package WiproEP;
public class ThreadLifeCycleDemo extends Thread {

   private final Object lock = new Object();
   @Override
   public void run() {
       try {
           // RUNNABLE state
```

```java
            System.out.println(getState() + ": Thread is 
running");
            // Demonstrate TIMED_WAITING state using sleep
            System.out.println(getState() + ": Thread is going to 
sleep");
            Thread.sleep(2000);
            // Demonstrate WAITING state using wait
            synchronized (lock) {
                System.out.println(getState() + ": Thread is 
waiting for lock");
            }
            // Demonstrate BLOCKED state by trying to enter a 
synchronized block
            Thread blockerThread = new Thread(() -> {
                synchronized (lock) {
                    
System.out.println(Thread.currentThread().getState() + ": 
Holding the lock");
                    try {
                        Thread.sleep(3000); // Hold the lock for 
3 seconds to simulate BLOCKED state
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
            blockerThread.start();
            // Allow some time for the other thread to start and 
acquire the lock
            Thread.sleep(100);
            synchronized (lock) {
                System.out.println(getState() + ": Acquired the 
lock again");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // TERMINATED state after run method completes
        System.out.println(getState() + ": Thread is 
terminating");
    }
```

```
    public static void main(String[] args) {
        try {
            ThreadLifeCycleDemo thread = new
ThreadLifeCycleDemo();
            // NEW state
            System.out.println(thread.getState() + ": Thread is
in NEW state");
            // Start the thread to move to RUNNABLE state
            thread.start();
            System.out.println(thread.getState() + ": Thread is
in RUNNABLE state");
            // Wait a little to ensure the thread has started
            Thread.sleep(100);
            // Wake up the waiting thread
            synchronized (thread.lock) {
                thread.lock.notify();
            }
            // Join the thread to ensure it finishes
            thread.join();
            System.out.println(thread.getState() + ": Thread has
TERMINATED");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Output

```
NEW: Thread is in NEW state
RUNNABLE: Thread is in RUNNABLE state
RUNNABLE: Thread is running
RUNNABLE: Thread is going to sleep
RUNNABLE: Thread is waiting for lock
RUNNABLE: Holding the lock
RUNNABLE: Acquired the lock again
RUNNABLE: Thread is terminating
TERMINATED: Thread has 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 WiproEP;
import java.util.LinkedList;
import java.util.Queue;
class ProducerConsumer {
    private final Queue<Integer> buffer = new LinkedList<>();
    private final int capacity;
    private boolean stopRequested = false;
    public ProducerConsumer(int capacity) {
        this.capacity = capacity;
    }
    public void produce() throws InterruptedException {
        int value = 0;
        while (true) {
            synchronized (this) {
                while (buffer.size() == capacity) {
                    wait();  // Wait until the buffer has space
                }
                if (stopRequested) {
                    break;
                }
                System.out.println("Producer produced: " +
value);
                buffer.add(value++);
                notify();  // Notify the consumer that buffer is
not empty
                Thread.sleep(1000);  // Simulate time taken to
produce an item
            }
        }
        System.out.println("Producer thread stopped.");
    }
    public void consume() throws InterruptedException {
```

```java
        while (true) {
            synchronized (this) {
                while (buffer.isEmpty()) {
                    wait();  // Wait until the buffer has at
least one item
                }
                if (stopRequested && buffer.isEmpty()) {
                    break;
                }
                int value = buffer.poll();
                System.out.println("Consumer consumed: " +
value);
                notify();  // Notify the producer that buffer is
not full
                Thread.sleep(1000);  // Simulate time taken to
consume an item
            }
        }
        System.out.println("Consumer thread stopped.");
    }
    public synchronized void stop() {
        stopRequested = true;
        notifyAll();  // Notify all waiting threads to wake up
and check the stop condition
    }
    public static void main(String[] args) {
        ProducerConsumer pc = new ProducerConsumer(5);
        Thread producerThread = new Thread(() -> {
            try {
                pc.produce();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        Thread consumerThread = new Thread(() -> {
            try {
                pc.consume();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
```

```java
        producerThread.start();
        consumerThread.start();
        try {
            Thread.sleep(10000);  // Let the producer and
consumer run for a while
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        pc.stop();  // Request stop of producer and consumer
threads
        try {
            producerThread.join();
            consumerThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Producer and Consumer have been
stopped.");
    }
}
```

Output

```
Producer produced: 0
Producer produced: 1
Producer produced: 2
Producer produced: 3
Producer produced: 4
Consumer consumed: 0
Consumer consumed: 1
Consumer consumed: 2
Consumer consumed: 3
Consumer consumed: 4
Producer produced: 5
Producer produced: 6
Producer produced: 7
Consumer consumed: 5
Consumer consumed: 6
Consumer consumed: 7
Producer produced: 8
Producer produced: 9
```

```
Producer produced: 10
Producer produced: 11
Producer produced: 12
Consumer consumed: 8
Consumer consumed: 9
Consumer consumed: 10
Consumer consumed: 11
Producer thread stopped.
Consumer consumed: 12
```

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 WiproEP;
class BankAccount {
    private int balance = 0;
    // Synchronized method to deposit money
    public synchronized void deposit(int amount) {
        balance += amount;
        System.out.println(Thread.currentThread().getName() + "
deposited " + amount + ", new balance: " + balance);
    }
    // Synchronized method to withdraw money
    public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println(Thread.currentThread().getName() +
" withdrew " + amount + ", new balance: " + balance);
        } else {
            System.out.println(Thread.currentThread().getName() +
" attempted to withdraw " + amount + ", insufficient balance: "
+ balance);
        }
    }
    // Synchronized method to get the current balance
    public synchronized int getBalance() {
        return balance;
    }
```

```
}
public class BankSimulation {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        // Creating and starting threads for deposits and
withdrawals
        Thread t1 = new Thread(() -> account.deposit(100),
"Thread-1");
        Thread t2 = new Thread(() -> account.withdraw(50),
"Thread-2");
        Thread t3 = new Thread(() -> account.deposit(200),
"Thread-3");
        Thread t4 = new Thread(() -> account.withdraw(150),
"Thread-4");
        t1.start();
        t2.start();
        t3.start();
        t4.start();
        // Joining all threads to ensure they complete before the
program ends
        try {
            t1.join();
            t2.join();
            t3.join();
            t4.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // Print final balance
        System.out.println("Final balance: " +
account.getBalance());
    }
}
```

Output

```
Thread-2 attempted to withdraw 50, insufficient balance: 0
Thread-4 attempted to withdraw 150, insufficient balance: 0
Thread-3 deposited 200, new balance: 200
Thread-1 deposited 100, new balance: 300
Final balance: 300
```

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 WiproEP;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
public class ThreadPoolExample {
   public static void main(String[] args) {
       // Create a fixed-size thread pool with 4 threads
       ExecutorService executor =
Executors.newFixedThreadPool(4);
       // Submit multiple tasks to the thread pool
       for (int i = 0; i < 10; i++) {
           int taskId = i;
           executor.submit(() -> performTask(taskId));
       }
       // Shut down the executor and wait for tasks to complete
       executor.shutdown();
       try {
           if (!executor.awaitTermination(60, TimeUnit.SECONDS))
{
               executor.shutdownNow();
           }
       } catch (InterruptedException e) {
           executor.shutdownNow();
       }
   }
   // Task performing a complex calculation (e.g., Fibonacci)
   private static void performTask(int taskId) {
       System.out.println("Task " + taskId + " started by " +
Thread.currentThread().getName());
```

```
        long result = fibonacci(30); // Example complex
calculation
        System.out.println("Task " + taskId + " completed by " +
Thread.currentThread().getName() + " with result: " + result);
    }
    // Example of a complex calculation: Fibonacci sequence
    private static long fibonacci(int n) {
        if (n <= 1) return n;
        else return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Output

```
Task 0 started by pool-1-thread-1
Task 3 started by pool-1-thread-4
Task 2 started by pool-1-thread-3
Task 1 started by pool-1-thread-2
Task 2 completed by pool-1-thread-3 with result: 832040
Task 1 completed by pool-1-thread-2 with result: 832040
Task 4 started by pool-1-thread-3
Task 0 completed by pool-1-thread-1 with result: 832040
Task 5 started by pool-1-thread-1
Task 3 completed by pool-1-thread-4 with result: 832040
Task 6 started by pool-1-thread-2
Task 7 started by pool-1-thread-4
Task 5 completed by pool-1-thread-1 with result: 832040
Task 4 completed by pool-1-thread-3 with result: 832040
Task 8 started by pool-1-thread-1
Task 7 completed by pool-1-thread-4 with result: 832040
Task 6 completed by pool-1-thread-2 with result: 832040
Task 9 started by pool-1-thread-3
Task 8 completed by pool-1-thread-1 with result: 832040
Task 9 completed by pool-1-thread-3 with result: 832040
```

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 WiproEP;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
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 void main(String[] args) {
       int limit = 100; // Specify the upper limit for prime
numbers
       // Step 1: Calculate prime numbers using ExecutorService
       List<Integer> primeNumbers =
calculatePrimeNumbers(limit);
       // Step 2: Write prime numbers to a file asynchronously
       writePrimeNumbersToFileAsync(primeNumbers);
   }
   private static List<Integer> calculatePrimeNumbers(int limit)
{
       ExecutorService executor =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProce
ssors());
       List<Callable<Integer>> tasks = new ArrayList<>();
       for (int i = 2; i <= limit; i++) {
           final int num = i;
```

```java
            tasks.add(() -> isPrime(num) ? num : null);
        }
        try {
            List<Future<Integer>> results =
executor.invokeAll(tasks);
            List<Integer> primeNumbers = new ArrayList<>();
            for (Future<Integer> result : results) {
                if (result.get() != null) {
                    primeNumbers.add(result.get());
                }
            }
            return primeNumbers;
        } catch (Exception e) {
            e.printStackTrace();
            return new ArrayList<>();
        } finally {
            executor.shutdown();
        }
    }
    private static boolean isPrime(int num) {
        if (num <= 1) {
            return false;
        }
        for (int i = 2; i * i <= num; i++) {
            if (num % i == 0) {
                return false;
            }
        }
        return true;
    }
    private static void
writePrimeNumbersToFileAsync(List<Integer> primeNumbers) {
        CompletableFuture<Void> writeToFileFuture =
CompletableFuture.runAsync(() -> {
            try (BufferedWriter writer = new BufferedWriter(new
FileWriter("prime_numbers.txt"))) {
                for (Integer prime : primeNumbers) {
                    writer.write(prime.toString());
                    writer.newLine();
                }
            } catch (IOException e) {
```

```
                e.printStackTrace();
            }
        });
        // Wait for the asynchronous write operation to complete
        writeToFileFuture.join();
    }
}
```

Output

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

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.

Code

```
package WiproEP;
public class ThreadSafeDemo {
   public static void main(String[] args) {
       Counter counter = new Counter();
       ImmutableData sharedData = new ImmutableData(100); //
Example shared data
       Runnable incrementTask = () -> {
           for (int i = 0; i < 1000; i++) {
               counter.increment();
           }
           System.out.println(Thread.currentThread().getName() +
" finished incrementing. Counter: " + counter.getCount());
       };
       Runnable decrementTask = () -> {
           for (int i = 0; i < 1000; i++) {
               counter.decrement();
           }
           System.out.println(Thread.currentThread().getName() +
" finished decrementing. Counter: " + counter.getCount());
       };
       Thread thread1 = new Thread(incrementTask);
       Thread thread2 = new Thread(decrementTask);
       Thread thread3 = new Thread(incrementTask);
       Thread thread4 = new Thread(decrementTask);
       thread1.start();
       thread2.start();
       thread3.start();
       thread4.start();
       try {
           thread1.join();
           thread2.join();
           thread3.join();
           thread4.join();
       } catch (InterruptedException e) {
           e.printStackTrace();
```

```java
        }
        System.out.println("Final Counter value: " +
counter.getCount());
        System.out.println("Shared ImmutableData value: " +
sharedData.getValue());
    }
}
// Thread-Safe Counter Class
class Counter {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
    public synchronized void decrement() {
        count--;
    }
    public synchronized int getCount() {
        return count;
    }
}
// Immutable Data Class
final class ImmutableData {
    private final int value;
    public ImmutableData(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
}
```

Output

```
Thread-2 finished incrementing. Counter: 0
Thread-1 finished decrementing. Counter: 0
Thread-3 finished decrementing. Counter: 0
Thread-0 finished incrementing. Counter: 0
Final Counter value: 0
Shared ImmutableData value: 100
```