Day 18

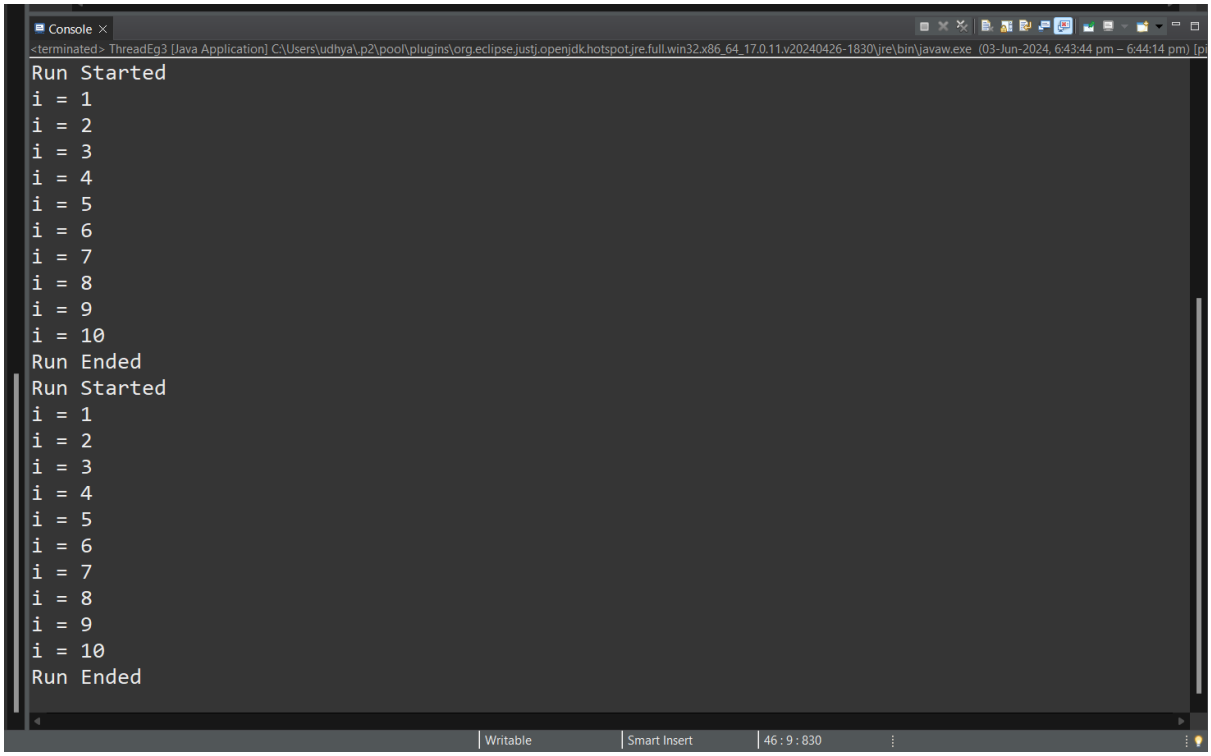
Task 1: Creating and Managing Threads

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

```
package com.wipro
class ThreadA {
	public synchronized void print1_10() {
		System.out.println("Run Started");
		try {
			for (int i = 1; i <= 10; i++) {
				Thread.sleep(1000);
				System.out.println("i = " + i);
			}
		} catch (InterruptedException e) {

			e.printStackTrace();
		}
		System.out.println("Run Ended");
	}
}
class ThreadB extends Thread {
	ThreadA ta;

	public ThreadB(ThreadA ta) {
		this.ta = ta;
	}
```

```
	public void run() {
		ta.print1_10();
	}
}

public class ThreadEg3 {
	public static void main(String[] args) {
		System.out.println("Main Started");
		ThreadA ta = new ThreadA();
		ThreadB t1 = new ThreadB(ta);
		t1.start();

		ThreadB t2 = new ThreadB(ta);
		t2.start();

		ThreadB t3 = new ThreadB(ta);
		t3.start();
		System.out.println("Main Ended");
	}

}
```

OUTPUT:



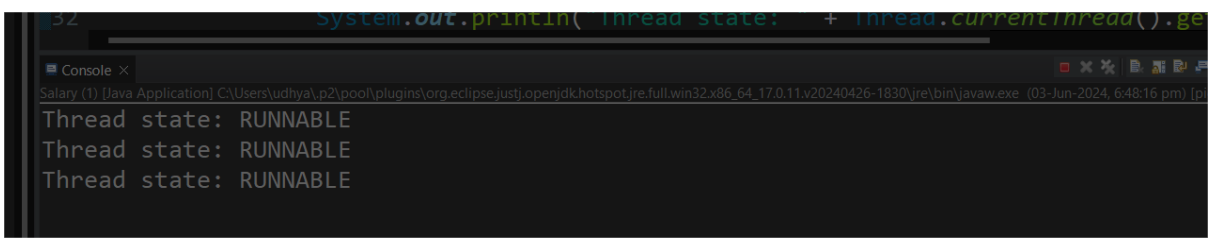
Task 2: States and Transitions

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..

```
public class ThreadLifecycleDemo {


    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(() -> {
            try {
                System.out.println("Thread state: " + Thread.currentThread().getState());
                Thread.sleep(1000);              System.out.println("Thread state: " +
Thread.currentThread().getState());


                synchronized(ThreadLifecycleDemo.class) {
                    ThreadLifecycleDemo.class.wait();
```

```
            }

            System.out.println("Thread state: " + Thread.currentThread().getState());

            Thread.sleep(2000);
            synchronized(ThreadLifecycleDemo.class) {
                System.out.println("Thread state: " + Thread.currentThread().getState());
            }

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


    thread.start();

    System.out.println("Thread state: " + thread.getState());
    thread.join();

    System.out.println("Thread state: " + thread.getState());
  }
}
```

OUTPUT:



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.

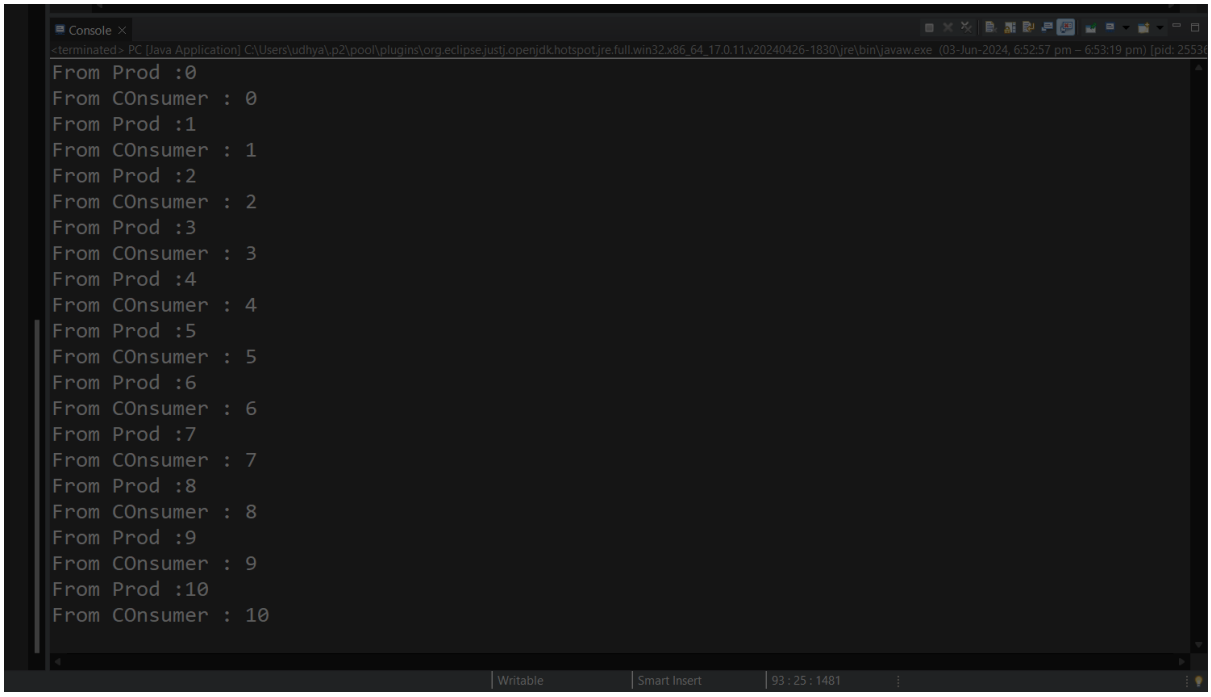
```
package com.wipro;


class Common {
	int num;
	boolean available = false;


	public synchronized int put(int num) {
		if (available)
			try {
				wait();
			} catch (InterruptedException e) {


				e.printStackTrace();
			}
		this.num = num;
		System.out.println("From Prod :" + this.num);


		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {


			e.printStackTrace();
		}
```

```
		available = true;
		notify();
		return num;
	}

	public synchronized int get() {
		if (!available)
			try {
				wait();
			} catch (InterruptedException e) {

				e.printStackTrace();
			}
		System.out.println("From COnsumer : " + this.num);

		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {

			e.printStackTrace();
		}
		available = false;
		notify();
		return num;

	}
}
```

```
class Producer extends Thread {
	Common c;

	public Producer(Common c) {
		this.c = c;
		new Thread(this, "Producer :").start();
	}

	public void run() {
		int x = 0, i = 0;
		while (x <= 10) {
			c.put(i++);
			x++;
		}
	}
}

class Consumer extends Thread {
	Common c;

	public Consumer(Common c) {
		this.c = c;
		new Thread(this, "Consumer :").start();
	}

	public void run() {
		int x = 0;
		while (x <= 10) {
			c.get();
```

```
                x++;
            }
        }
}

public class PC {

    public static void main(String[] args) {
        Common c = new Common();
        new Producer(c);
        new Consumer(c);
    }

}
```

OUTPUT:



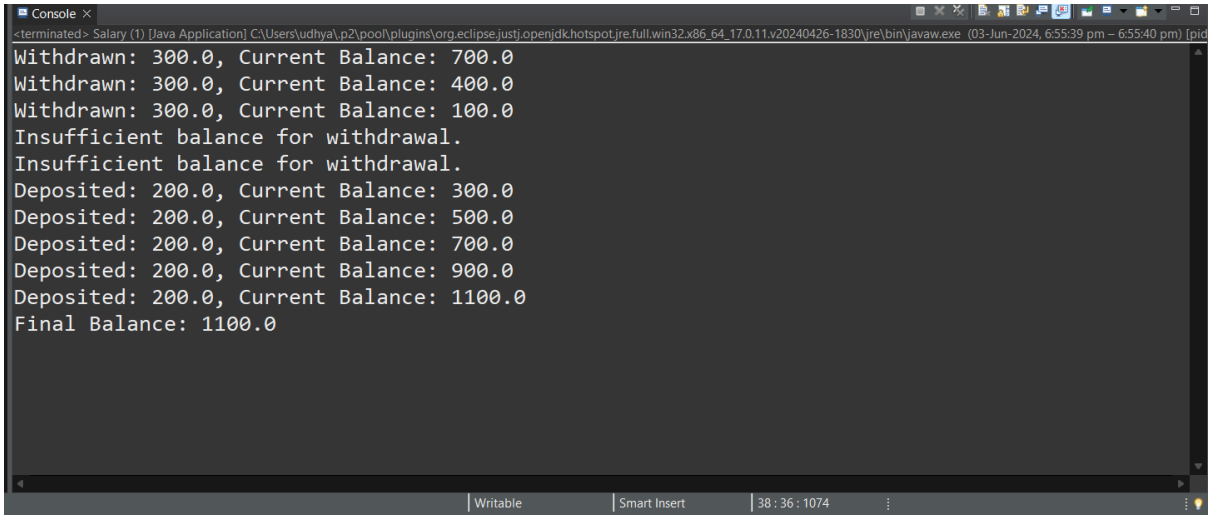
Task 4: Synchronized Blocks and Methods

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.

```
public class BankAccount {
    private double balance;

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

    public synchronized void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited: " + amount + ", Current Balance: " + balance);
    }
    public synchronized void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount + ", Current Balance: " + balance);
        } else {
            System.out.println("Insufficient balance for withdrawal.");
        }
    }
    public synchronized double getBalance() {
        return balance;
    }

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

```
        Thread depositThread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                account.deposit(200);
            }
        });

        Thread withdrawThread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                account.withdraw(300);
            }
        });
        depositThread.start();
        withdrawThread.start();
        try {
            depositThread.join();
            withdrawThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Final Balance: " + account.getBalance());
    }
}
```

OUTPUT:



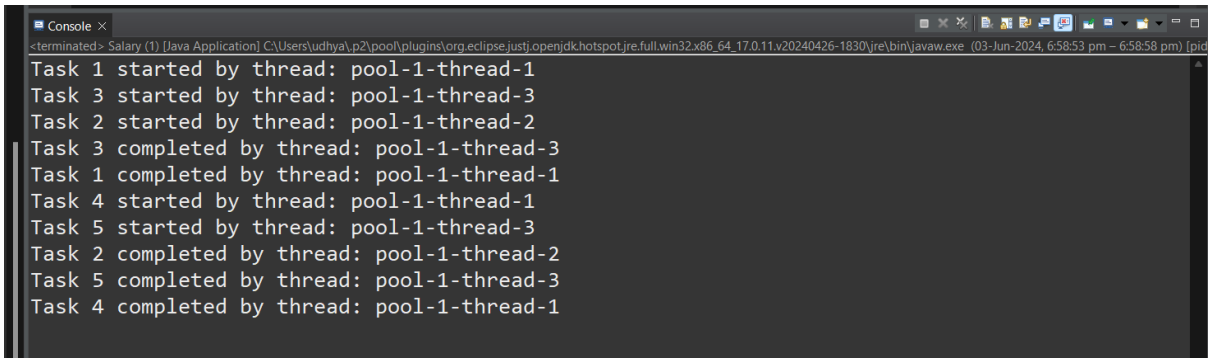
Task 5: Thread Pools and Concurrency Utilities

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

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

public class ThreadPoolDemo {

   public static void main(String[] args) {

      ExecutorService executor = Executors.newFixedThreadPool(3);

         for (int i = 1; i <= 5; i++) {
        int taskId = i;
        executor.submit(() -> {
          System.out.println("Task " + taskId + " started by thread: " +
Thread.currentThread().getName());
          try {
            Thread.sleep(2000); // Simulating a time-consuming task
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Task " + taskId + " completed by thread: " +
Thread.currentThread().getName());
        });
    }

            executor.shutdown();
  }
}
```

OUTPUT:



Task 6: Executors, Concurrent Collections, Completable Future

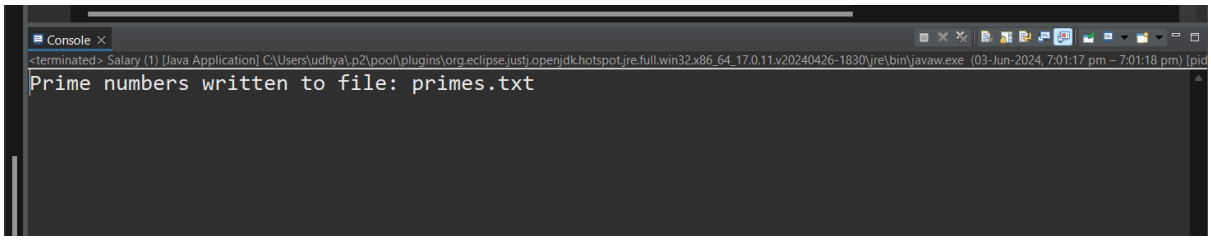
Use an ExecutorService to parallelize a task that calculates prime numbers up to a given number and then use CompletableFuture to write the results to a file asynchronously.

```
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 PrimeNumberCalculator {


    public static void main(String[] args) throws IOException {
        int n = 100;
        String filePath = "primes.txt";
        ExecutorService executor =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
        CompletableFuture<List<Integer>> primesFuture = CompletableFuture.supplyAsync(() -
> calculatePrimes(n), executor);
        primesFuture.thenAcceptAsync(primes -> {
            try (FileWriter writer = new FileWriter(filePath)) {
                for (Integer prime : primes) {
                    writer.write(prime + "\n");
                }
                System.out.println("Prime numbers written to file: " + filePath);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }, executor);
        executor.shutdown();
    }


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

OUTPUT:



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.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
class Counter {
   private int count;
   private final Lock 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 ImmutableData {
    private final int value;

    public ImmutableData(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

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

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

        incrementThread.start();
```

```
        decrementThread.start();

        incrementThread.join();
        decrementThread.join();
        System.out.println("Final Count: " + counter.getCount());
        ImmutableData immutableData = new ImmutableData(10);
        Thread accessThread1 = new Thread(() -> {
            System.out.println("Thread 1: Value = " + immutableData.getValue());
        });

        Thread accessThread2 = new Thread(() -> {
            System.out.println("Thread 2: Value = " + immutableData.getValue());
        });

        accessThread1.start();
        accessThread2.start();
    }
}
```

OUTPUT:

```
Final Count: 0
Thread 1: Value = 10
Thread 2: Value = 10
```