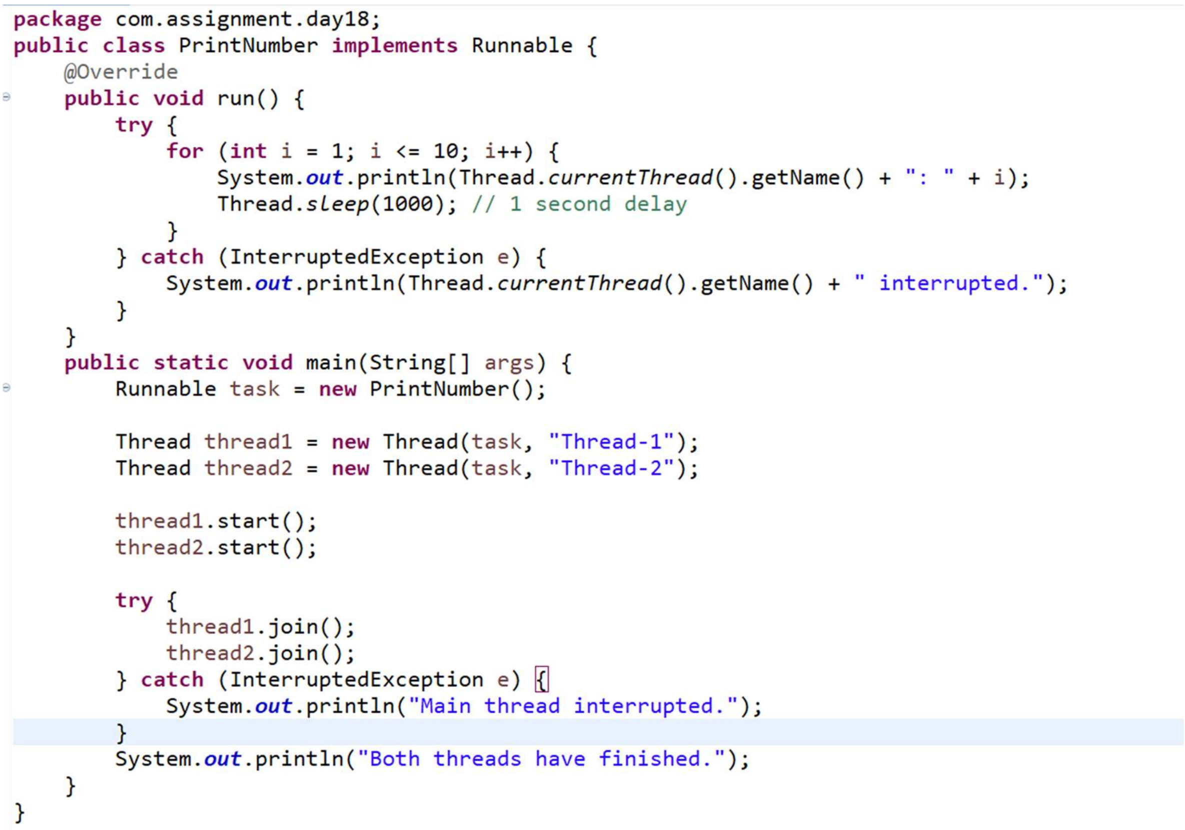
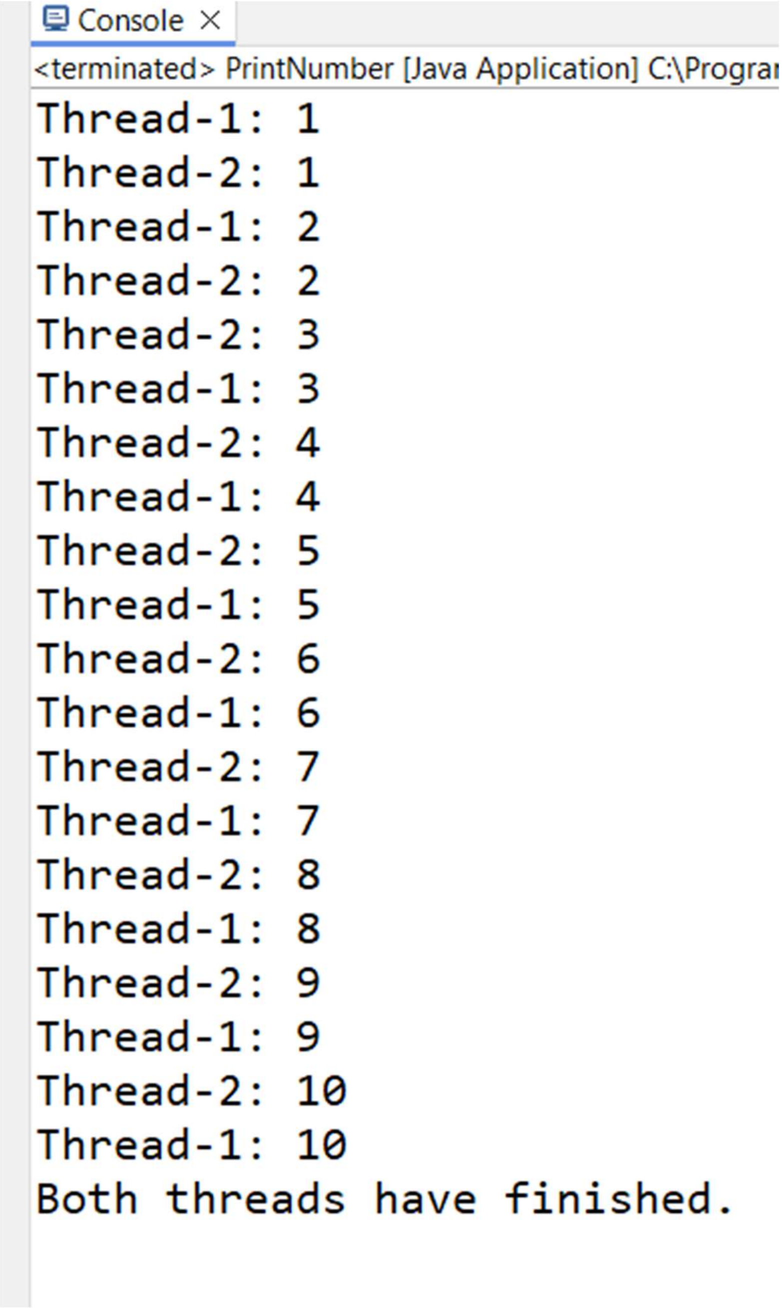
Day 18:
Task 1: Creating and Managing Threads
Write a program that starts two threads, where each thread prints numbers from 1 to 10 with a 1-second delay between each number



```java
package com.assignment.day18;
public class PrintNumber implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 1; i <= 10; i++) {
                System.out.println(Thread.currentThread().getName() + ": " + i);
                Thread.sleep(1000); // 1 second delay
            }
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " interrupted.");
        }
    }
    public static void main(String[] args) {
        Runnable task = new PrintNumber();

        Thread thread1 = new Thread(task, "Thread-1");
        Thread thread2 = new Thread(task, "Thread-2");

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

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Both threads have finished.");
    }
}
```

Output:

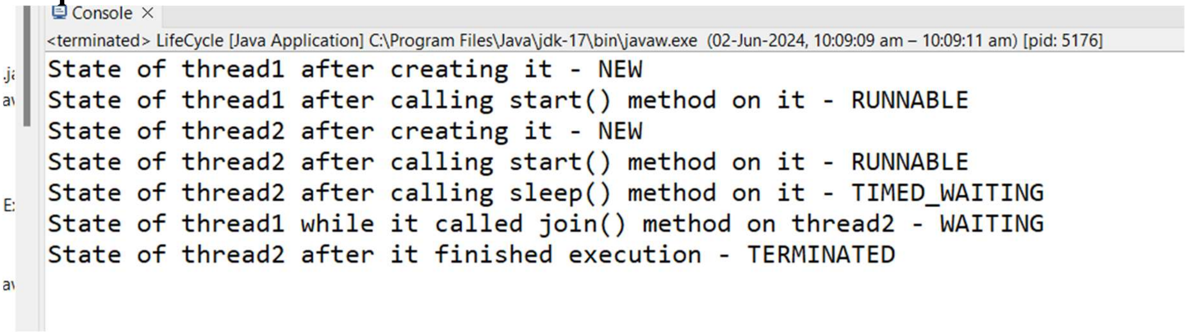


```
<terminated> PrintNumber [Java Application] C:\Prograı
Thread-1: 1
Thread-2: 1
Thread-1: 2
Thread-2: 2
Thread-2: 3
Thread-1: 3
Thread-2: 4
Thread-1: 4
Thread-2: 5
Thread-1: 5
Thread-2: 6
Thread-1: 6
Thread-2: 7
Thread-1: 7
Thread-2: 8
Thread-1: 8
Thread-2: 9
Thread-1: 9
Thread-2: 10
Thread-1: 10
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.

```java
package com.assignment.day18;

class ThreadExample implements Runnable {
    @Override
    public void run() {
        try {
            Thread.sleep(1500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(
            "State of thread1 while it called join()
method on thread2 - " + LifeCycle.thread1.getState()
        );

        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public class LifeCycle implements Runnable {
    public static Thread thread1;
    public static LifeCycle obj;
    public static void main(String[] args) {
        obj = new LifeCycle();
        thread1 = new Thread(obj);
        System.out.println("State of thread1 after
creating it - " + thread1.getState());
        thread1.start();
        System.out.println("State of thread1 after
calling start() method on it - " + thread1.getState());
    }
    @Override
    public void run() {
        ThreadExample myThread = new ThreadExample();
        Thread thread2 = new Thread(myThread);
        System.out.println("State of thread2 after
creating it - " + thread2.getState());
        thread2.start();
        System.out.println("State of thread2 after
calling start() method on it - " + thread2.getState());
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("State of thread2 after 
calling sleep() method on it - " + thread2.getState());
        try {
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("State of thread2 after it 
finished execution - " + thread2.getState());
    }
}
```

Output:



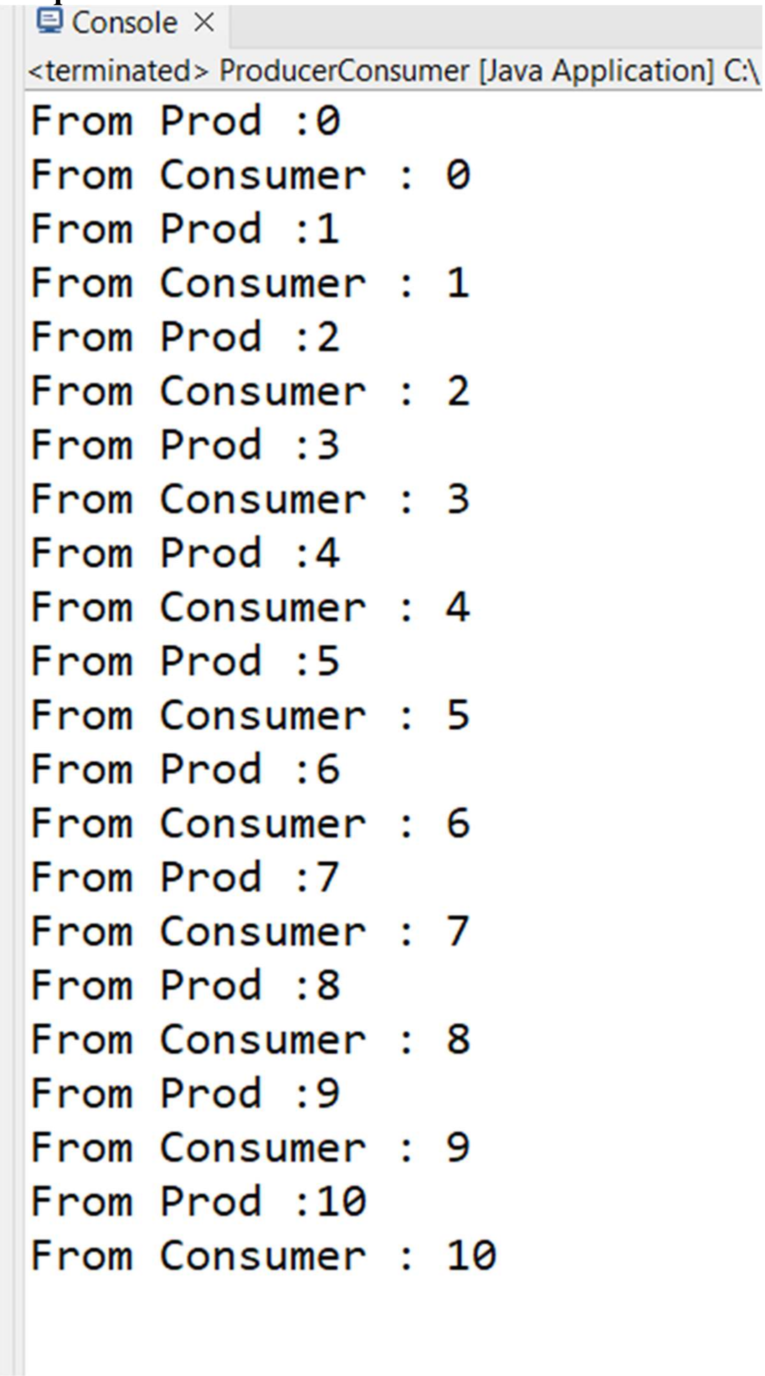
```
<terminated> LifeCycle [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe  (02-Jun-2024, 10:09:09 am – 10:09:11 am) [pid: 5176]
State of thread1 after creating it - NEW
State of thread1 after calling start() method on it - RUNNABLE
State of thread2 after creating it - NEW
State of thread2 after calling start() method on it - RUNNABLE
State of thread2 after calling sleep() method on it - TIMED_WAITING
State of thread1 while it called join() method on thread2 - WAITING
State of thread2 after it finished execution - 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.
Solution:

```
package com.assignment.day18;

class Common {
    int num;
    boolean available = false;

    public synchronized int put(int num) {
        synchronized (this) {
            if (available)
                try {
                    wait();
                } catch (InterruptedException e) {
                    // TODO: handle exception
                    e.printStackTrace();
                }
            this.num = num;
```

```java
            System.out.println("From Prod :" +
this.num);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO: handle exception
                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) {
            // TODO Auto-generated catch block
            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 ProducerConsumer {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Common c = new Common();
        new Producer(c);
        new Consumer(c);

    }
}
```

Output:



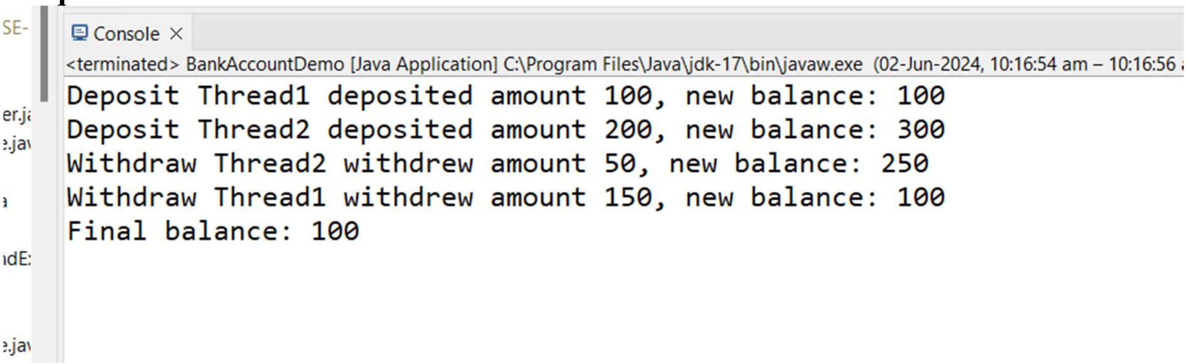
```
<terminated> ProducerConsumer [Java Application] C:\
From Prod :0
From Consumer : 0
From Prod :1
From Consumer : 1
From Prod :2
From Consumer : 2
From Prod :3
From Consumer : 3
From Prod :4
From Consumer : 4
From Prod :5
From Consumer : 5
From Prod :6
From Consumer : 6
From Prod :7
From Consumer : 7
From Prod :8
From Consumer : 8
From Prod :9
From Consumer : 9
From Prod :10
From Consumer : 10
```

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.

Solution:

```
package com.assignment.day18;

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

        Thread depositThread1 = new Thread(new
DepositTask(account, 100), "Deposit Thread1");
        Thread depositThread2 = new Thread(new
DepositTask(account, 200), "Deposit Thread2");
        Thread withdrawThread1 = new Thread(new
WithdrawTask(account, 150), "Withdraw Thread1");
        Thread withdrawThread2 = new Thread(new
WithdrawTask(account, 50), "Withdraw Thread2");

        depositThread1.start();
        depositThread2.start();
        withdrawThread1.start();
        withdrawThread2.start();

        try {
            depositThread1.join();
            depositThread2.join();
            withdrawThread1.join();
            withdrawThread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

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

class BankAccount {
    private int balance = 0;

    public synchronized void deposit(int amount) {
        balance += amount;
        System.out.println(
                Thread.currentThread().getName() + "
deposited amount " + amount + ", new balance: " +
balance);
    }

    public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println(
                    Thread.currentThread().getName() +
" withdrew amount " + amount + ", new balance: " +
balance);
        } else {

    System.out.println(Thread.currentThread().getName()
+ " attempted to withdraw " + amount
                    + ", but insufficient funds.
Balance: " + balance);
        }
    }

    public int getBalance() {
        return balance;
    }
}

class DepositTask implements Runnable {
    private final BankAccount account;
    private final int amount;

    public DepositTask(BankAccount account, int amount)
{
        this.account = account;
        this.amount = amount;
    }

    @Override
    public void run() {
        account.deposit(amount);
    }
}

class WithdrawTask implements Runnable {
    private final BankAccount account;
    private final int amount;

    public WithdrawTask(BankAccount account, int amount)
{
        this.account = account;
        this.amount = amount;
    }

    @Override
    public void run() {
        account.withdraw(amount);
    }
}
```

Output:



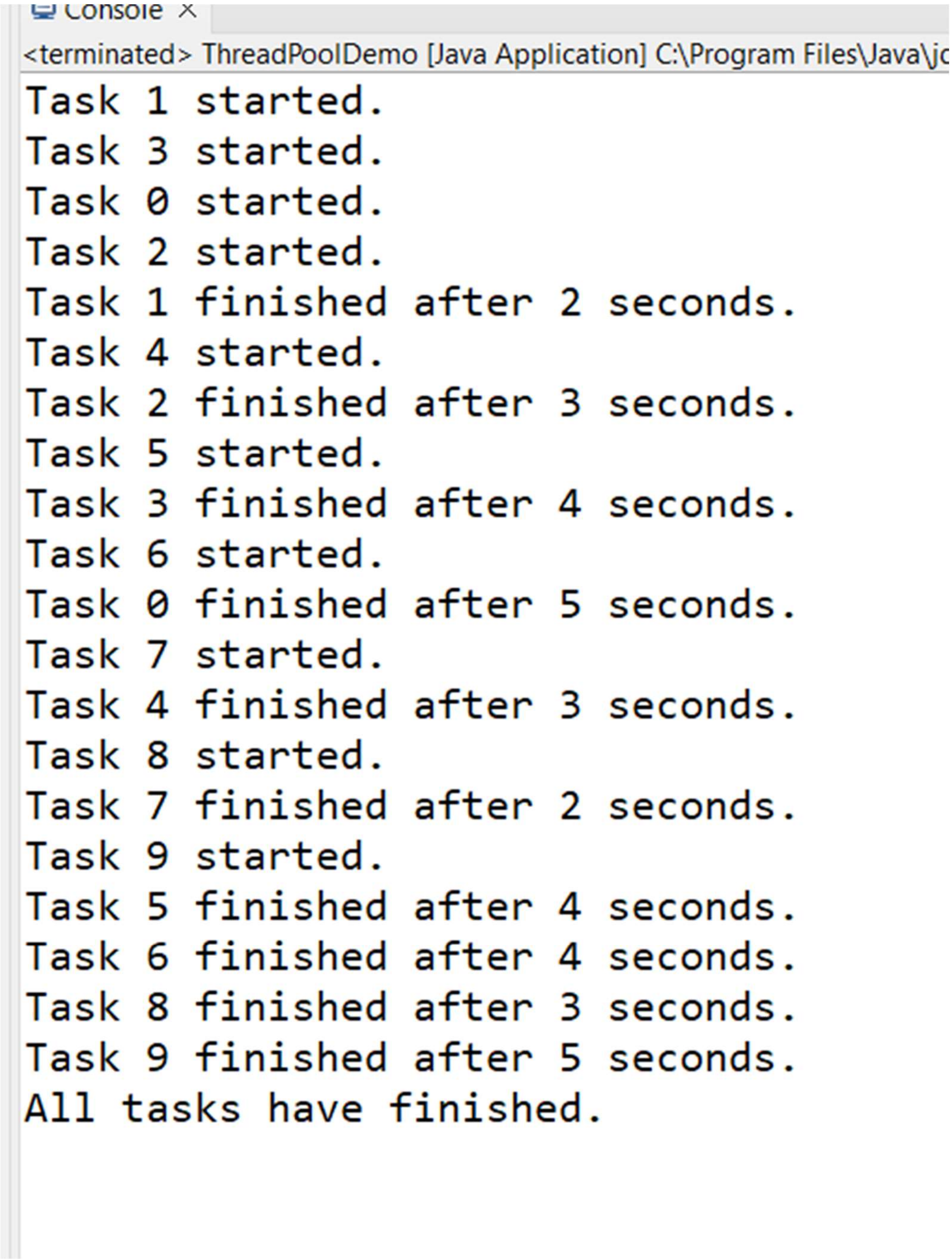
```
<terminated> BankAccountDemo [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (02-Jun-2024, 10:16:54 am – 10:16:56
Deposit Thread1 deposited amount 100, new balance: 100
Deposit Thread2 deposited amount 200, new balance: 300
Withdraw Thread2 withdrew amount 50, new balance: 250
Withdraw Thread1 withdrew amount 150, new balance: 100
Final balance: 100
```

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.
Solution:

```
package com.assignment.day18;

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

public class ThreadPoolDemo {

    public static void main(String[] args) {

        ExecutorService executor = Executors.newFixedThreadPool(4);

        for (int i = 0; i < 10; i++) {
            executor.submit(new CalculationTask(i));
        }

        executor.shutdown();

        try {

            if (!executor.awaitTermination(1, TimeUnit.HOURS)) {
```

```
                                        executor.shutdownNow();
                                }
                        } catch (InterruptedException e) {
                                executor.shutdownNow();
                        }

                        System.out.println("All tasks have finished.");
                }
}

class CalculationTask implements Runnable {
        private final int taskId;
        private final Random random = new Random();

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

        @Override
        public void run() {
                System.out.println("Task " + taskId + " started.");

                long duration = random.nextInt(5) + 1;
                try {
                        TimeUnit.SECONDS.sleep(duration);
                } catch (InterruptedException e) {
                        System.out.println("Task " + taskId + " was interrupted.");
                }

                System.out.println("Task " + taskId + " finished after " + duration + "
seconds.");
        }
}
```

Output:

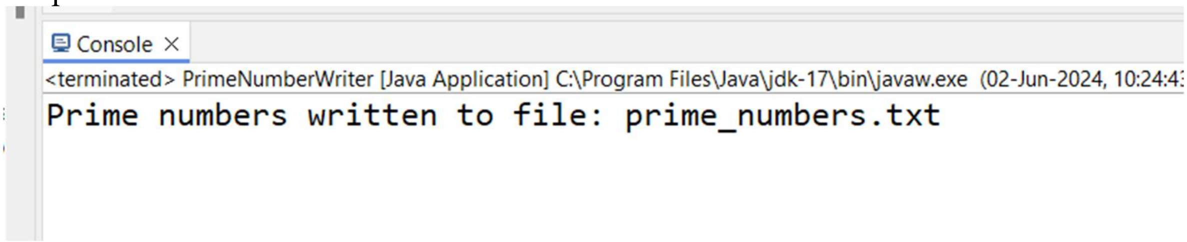


```
<terminated> ThreadPoolDemo [Java Application] C:\Program Files\Java\jc
Task 1 started.
Task 3 started.
Task 0 started.
Task 2 started.
Task 1 finished after 2 seconds.
Task 4 started.
Task 2 finished after 3 seconds.
Task 5 started.
Task 3 finished after 4 seconds.
Task 6 started.
Task 0 finished after 5 seconds.
Task 7 started.
Task 4 finished after 3 seconds.
Task 8 started.
Task 7 finished after 2 seconds.
Task 9 started.
Task 5 finished after 4 seconds.
Task 6 finished after 4 seconds.
Task 8 finished after 3 seconds.
Task 9 finished after 5 seconds.
All tasks have finished.
```

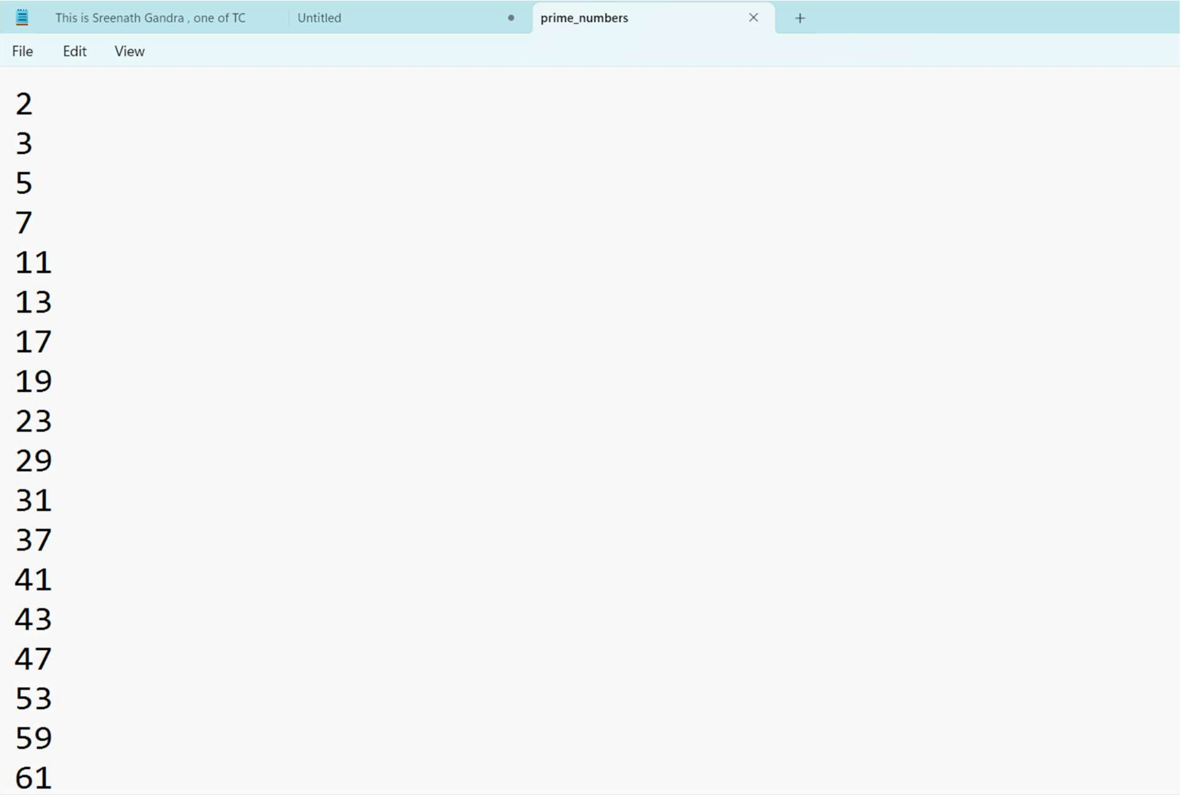
Task 6: Executors, Concurrent Collections, CompletableFuture
Use an ExecutorService to parallelize a task that calculates prime numbers up to a given number and then use CompletableFuture to write the results to a file asynchronously.

```
package com.assignment.day18;

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;
import java.util.concurrent.TimeUnit;

public class PrimeNumberWriter {

    private static final int NUM_THREADS = 4;
    private static final String FILE_NAME = "prime_numbers.txt";

    public static void main(String[] args) throws Exception {
        int upperLimit = 1000;

        List<Future<List<Integer>>> primeNumberFutures = calculatePrimes(upperLimit);
        List<Integer> allPrimes = new ArrayList<>();

        for (Future<List<Integer>> future : primeNumberFutures) {
            allPrimes.addAll(future.get());
        }

        writePrimesToFileAsync(allPrimes);

        System.out.println("Prime numbers written to file: " + FILE_NAME);
    }

    private static List<Future<List<Integer>>> calculatePrimes(int upperLimit) throws
Exception {
        ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);
        List<Future<List<Integer>>> futures = new ArrayList<>();
        int chunkSize = upperLimit / NUM_THREADS;

        for (int i = 0; i < upperLimit; i += chunkSize) {
            int start = i + 1;
            int end = Math.min(i + chunkSize, upperLimit);
            futures.add(executor.submit(() -> findPrimesInRange(start, end)));
        }

        executor.shutdown();
        executor.awaitTermination(10, TimeUnit.SECONDS);

        return futures;
    }

    private static List<Integer> findPrimesInRange(int start, int end) {
        List<Integer> primes = new ArrayList<>();
        for (int num = start; num <= end; num++) {
            if (isPrime(num)) {
                primes.add(num);
            }
        }
        return primes;
    }
    private static boolean isPrime(int num) {
        if (num < 2) {
            return false;
        }
        for (int i = 2; i <= Math.sqrt(num); i++) {
            if (num % i == 0) {
                return false;
            }
        }
        return true;
    }

    private static void writePrimesToFileAsync(List<Integer> primes) throws Exception {
        CompletableFuture<Void> writeFuture = CompletableFuture.runAsync(() -> {
            try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE_NAME))) {
                for (int prime : primes) {
                    writer.write(prime + "\n");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        });

        writeFuture.get();
    }
}
```

Output:



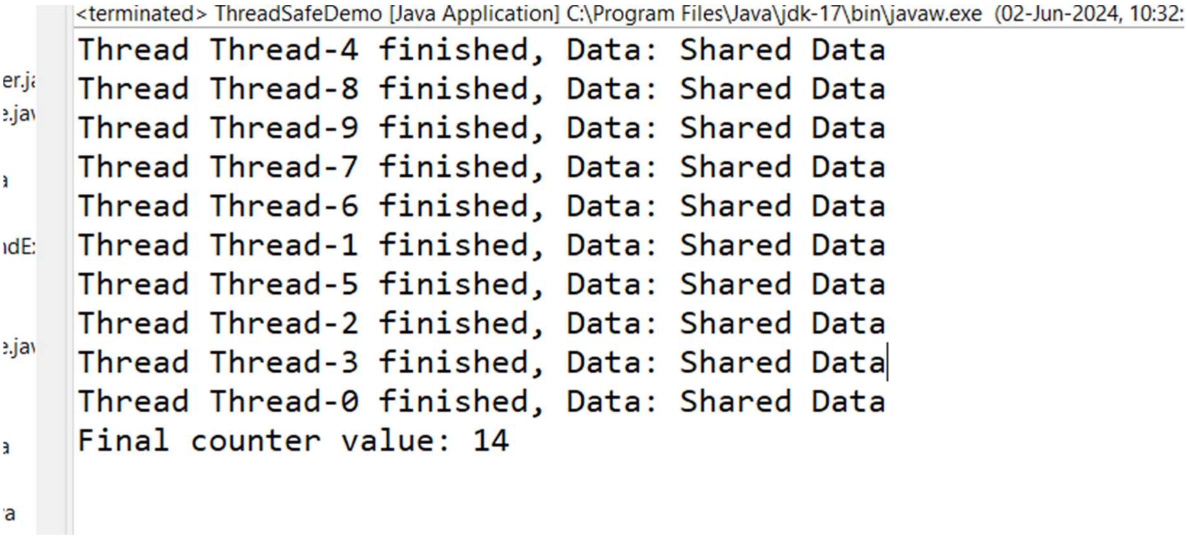
```
<terminated> PrimeNumberWriter [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe  (02-Jun-2024, 10:24:4
Prime numbers written to file: prime_numbers.txt
```



```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
```

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.assignment.day18;

import java.util.concurrent.atomic.AtomicInteger;

class ThreadSafeCounter {
    private final AtomicInteger count;

    public ThreadSafeCounter() {
        this.count = new AtomicInteger(0);
    }

    public void increment() {
        count.incrementAndGet();
    }

    public void decrement() {
        count.decrementAndGet();
    }
    public int get() {
        return count.get();
    }
}

class ImmutableData {
    private final String data;

    public ImmutableData(String data) {
        this.data = data;
    }

    public String getData() {
        return data;
    }
}

public class ThreadSafeDemo {
    public static void main(String[] args) {
        ThreadSafeCounter counter = new
ThreadSafeCounter();
        ImmutableData data = new ImmutableData("Shared
Data");

        int numThreads = 10;

        for (int i = 0; i < numThreads; i++) {
            Thread thread = new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    if (Math.random() > 0.5) {
                        counter.increment();
                    } else {
                        counter.decrement();
                    }
                }
                System.out.println("Thread " +
Thread.currentThread().getName() + " finished, Data: " +
data.getData());
            });
            thread.start();
        }
        for (int i = 0; i < numThreads; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Final counter value: " +
counter.get());
    }
}
```



```
<terminated> ThreadSafeDemo [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe  (02-Jun-2024, 10:32:
Thread Thread-4 finished, Data: Shared Data
Thread Thread-8 finished, Data: Shared Data
Thread Thread-9 finished, Data: Shared Data
Thread Thread-7 finished, Data: Shared Data
Thread Thread-6 finished, Data: Shared Data
Thread Thread-1 finished, Data: Shared Data
Thread Thread-5 finished, Data: Shared Data
Thread Thread-2 finished, Data: Shared Data
Thread Thread-3 finished, Data: Shared Data
Thread Thread-0 finished, Data: Shared Data
Final counter value: 14
```