# Day 18

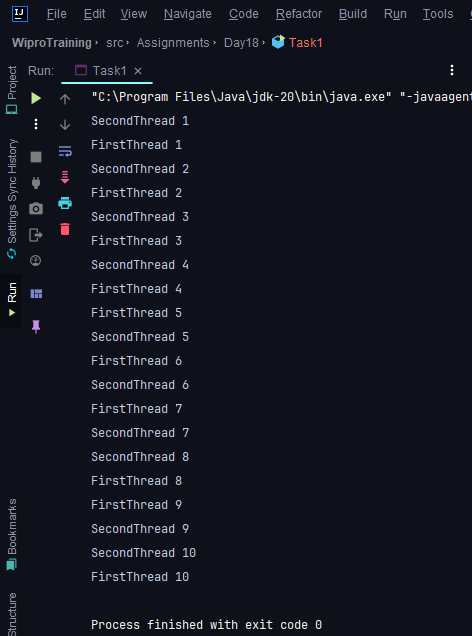
**Task 1: Creating and Managing Threads**

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

**Program:**

package Assignments.Day18;  
  
public class Task1 implements Runnable {  
 public static void main(String[] args) {  
 Runnable r = new Task1();  
 Thread t1 = new Thread(r);  
 t1.setName("FirstThread");  
 t1.setPriority(Thread.MIN\_PRIORITY);  
 Thread t2 = new Thread(r);  
 t2.setName("SecondThread");  
 t1.start();  
 t2.start();  
 }  
  
 @Override  
 public void run() {  
 for (int i = 1; i < 11; i++) {  
 System.out.println(Thread.currentThread().getName() +" "+i);  
  
 try {  
 Thread.sleep(1000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
}

**Output:**



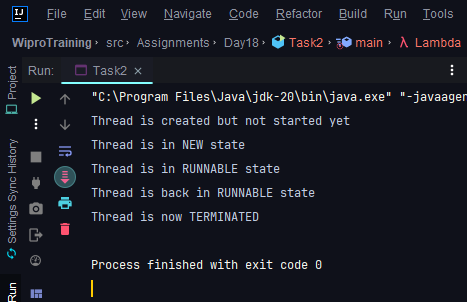
**Task 2: States and Transitions**

Create a Java class that simulates a thread going through different life-cycle states: NEW, RUNNABLE, WAITING, TIMED\_WAITING, BLOCKED, and TERMINATED. Use methods like sleep(), wait(), notify(), and join() to demonstrate these states..

**Program:**

package Assignments.Day18;  
  
public class Task2 {  
  
 public static void main(String[] args) {  
 Thread thread = new Thread(() -> {  
 try {  
 System.out.println("Thread is in NEW state");  
 Thread.sleep(1000); // Sleep for 1 second (TIMED\_WAITING)  
 System.out.println("Thread is in RUNNABLE state");  
 synchronized (Task2.class) {  
 Task2.class.wait(2000); // Wait (wait for 2 seconds)  
 }  
 System.out.println("Thread is back in RUNNABLE state");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
  
 System.out.println("Thread is created but not started yet");  
 thread.start(); // Start the thread (RUNNABLE)  
  
 try {  
 Thread.sleep(200);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 synchronized (Task2.class) {  
 Task2.class.notify(); // Notify the waiting thread  
 }  
  
 try {  
 thread.join(); // Wait for the thread to finish (TERMINATED)  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.out.println("Thread is now TERMINATED");  
 }  
}

**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.

**Program:**

package Assignments.Day18;

public class Task3 {

private static final int BUFFER\_SIZE = 5;

private static final Object lock = new Object();

private static int[] buffer = new int[BUFFER\_SIZE];

private static int itemCount = 0;

public static void main(String[] args) {

Thread producerThread = new Thread(() -> {

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

produce(i);

}

});

Thread consumerThread = new Thread(() -> {

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

consume();

}

});

producerThread.start();

consumerThread.start();

try {

producerThread.join();

consumerThread.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

private static void produce(int item) {

synchronized (lock) {

while (itemCount == BUFFER\_SIZE) {

try {

lock.wait(); // Buffer is full, wait for consumer

} catch (InterruptedException e) {

e.printStackTrace();

}

}

buffer[itemCount] = item;

itemCount++;

System.out.println("Produced: " + item);

lock.notify(); // Notify consumer

}

}

private static void consume() {

synchronized (lock) {

while (itemCount == 0) {

try {

lock.wait(); // Buffer is empty, wait for producer

} catch (InterruptedException e) {

e.printStackTrace();

}

}

int item = buffer[itemCount - 1];

itemCount--;

System.out.println("Consumed: " + item);

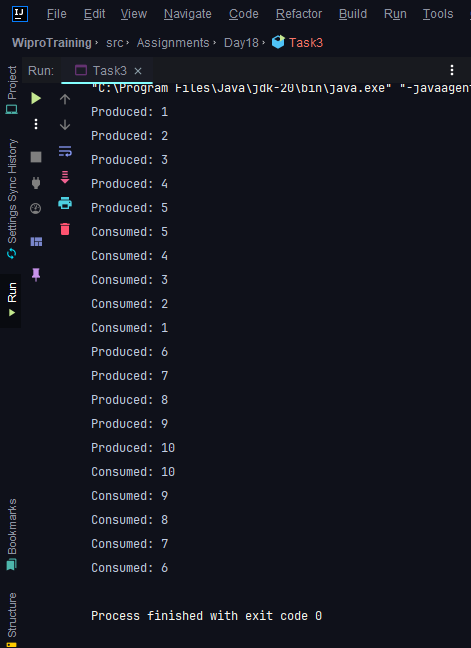
lock.notify(); // Notify producer

}

}

}

**Output:**



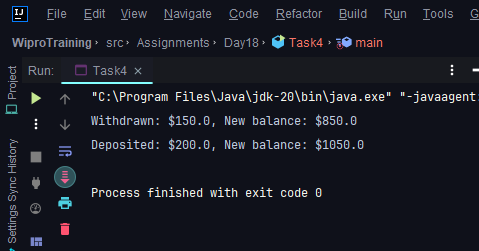
**Task 4: Synchronized Blocks and Methods**

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.

**Program:**

package Assignments.Day18;  
  
public class Task4 {  
 private double balance;  
  
 public Task4(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.");  
 }  
 }  
  
 public static void main(String[] args) {  
 Task4 account = new Task4(1000.0);  
  
 // Simulate multiple threads accessing the account  
 Thread thread1 = new Thread(() -> account.deposit(200.0));  
 Thread thread2 = new Thread(() -> account.withdraw(150.0));  
  
 thread1.start();  
 thread2.start();  
  
 try {  
 thread1.join();  
 thread2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

**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.

**Program:**

package Assignments.Day18;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Task5 {

public static void main(String[] args) {

int numThreads = 3;

ExecutorService executor = Executors.newFixedThreadPool(numThreads);

for (int i = 0; i < 5; i++) {

int taskId = i;

executor.submit(() -> {

System.out.println("Task " + taskId + " executed by thread " + Thread.currentThread().getName());

});

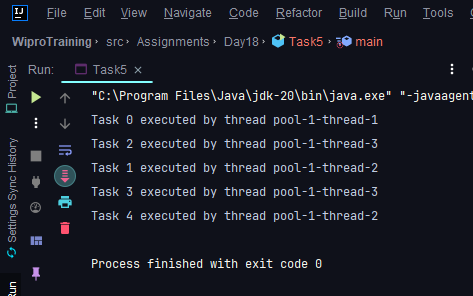
}

executor.shutdown();

}

}

**Output:**



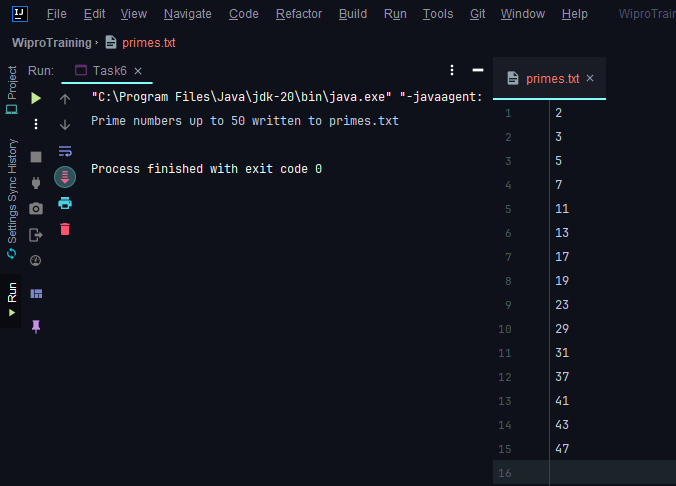
**Task 6: Executors, Concurrent Collections, CompletableFuture**

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:**

package Assignments.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;  
  
public class Task6 {  
  
 public 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;  
 }  
  
 public static List<Integer> calculatePrimes(int maxNumber) {  
 List<Integer> primes = new ArrayList<>();  
 for (int i = 2; i <= maxNumber; i++) {  
 if (isPrime(i)) {  
 primes.add(i);  
 }  
 }  
 return primes;  
 }  
  
 public static void main(String[] args) throws IOException {  
 int maxNumber = 100;  
 List<Integer> primes = calculatePrimes(maxNumber);  
  
 // Write primes to a file asynchronously  
 CompletableFuture<Void> writeToFileFuture = CompletableFuture.runAsync(() -> {  
 try (BufferedWriter writer = new BufferedWriter(new FileWriter("primes.txt"))) {  
 for (int prime : primes) {  
 writer.write(prime + "\n");  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 });  
  
 // Wait for the write operation to complete  
 writeToFileFuture.join();  
  
 System.out.println("Prime numbers up to " + maxNumber + " written to primes.txt");  
 }  
}

**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.

**Program:**

package Assignments.Day18;  
  
*// Thread-safe Counter class*  
class Counter {  
 private int value = 0;  
  
 public synchronized void increment() {  
 value++;  
 }  
  
 public synchronized void decrement() {  
 value--;  
 }  
  
 public synchronized int getValue() {  
 return value;  
 }  
}  
  
*// Immutable class for shared data*  
final class SharedData {  
 private final String data;  
  
 public SharedData(String data) {  
 this.data = data;  
 }  
  
 public String getData() {  
 return data;  
 }  
}  
  
public class Task7 {  
 public static void main(String[] args) {  
 Counter counter = new Counter();  
  
  *//multiple threads to demonstrate counter usage*  
   
 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();  
  
 try {  
 incrementThread.join();  
 decrementThread.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.out.println("Final counter value: " + counter.getValue());  
  
 SharedData sharedData = new SharedData("Hello, world!");  
  
 Thread readThread1 = new Thread(() -> {  
 System.out.println("Thread 1: " + sharedData.getData());  
 });  
  
 Thread readThread2 = new Thread(() -> {  
 System.out.println("Thread 2: " + sharedData.getData());  
 });  
  
 readThread1.start();  
 readThread2.start();  
 }  
}

**Output:**

