Day 10 - 18th June 2025

Advance concepts

Collections Framework intro, Streams, File I/O, Multithreading overview

Trobuleshooting

Debugging Tools, Error Messages and Stack Traces, Breakpoints and Code Stepping, Logging for Debugging, Common Bug Patterns, Debugging Strategies, Hands-on Debugging Practice

Multi Threading:

Task 1: What is a thread and a Process?

Solution:

Processes: A process is an instance of a computer program being executed. It has its own isolated address space, memory, and resources allocated by the operating system.

Threads: A thread is a unit of execution within a process. Threads are lightweight compared to processes and share the same memory and resources of the process they belong to.

Process-Thread Relationship: Each process can have one or more threads. Threads within a process can run concurrently, allowing the application to take advantage of multiple CPU cores and perform tasks in parallel.

Benefits of Threads: Threads enable improved performance and responsiveness by allowing applications to leverage parallel processing capabilities of modern hardware.

Challenges of Threads: Managing shared resources and ensuring proper synchronization between threads is critical to avoid issues like race conditions and deadlocks. The increased complexity can also make debugging multithreaded applications more challenging.

Task 2: Understand the below code and run it to see the output..

class RunnableDemo implements Runnable {

private Thread t;

private String threadName;

RunnableDemo( String name){

threadName = name;

System.out.println("Creating " + threadName );

}

public void run() {

System.out.println("Running " + threadName );

try {

for(int i = 4; i > 0; i--) {

System.out.println("Thread: " + threadName + ", " + i);

// Let the thread sleep for a while.

Thread.sleep(50);

}

} catch (InterruptedException e) {

System.out.println("Thread " + threadName + " interrupted.");

}

System.out.println("Thread " + threadName + " exiting.");

}

public void start ()

{

System.out.println("Starting " + threadName );

if (t == null)

{

t = new Thread (this, threadName);

t.start ();

}

}

}

public class TestThread {

public static void main(String args[]) {

RunnableDemo R1 = new RunnableDemo( "Thread-1");

R1.start();

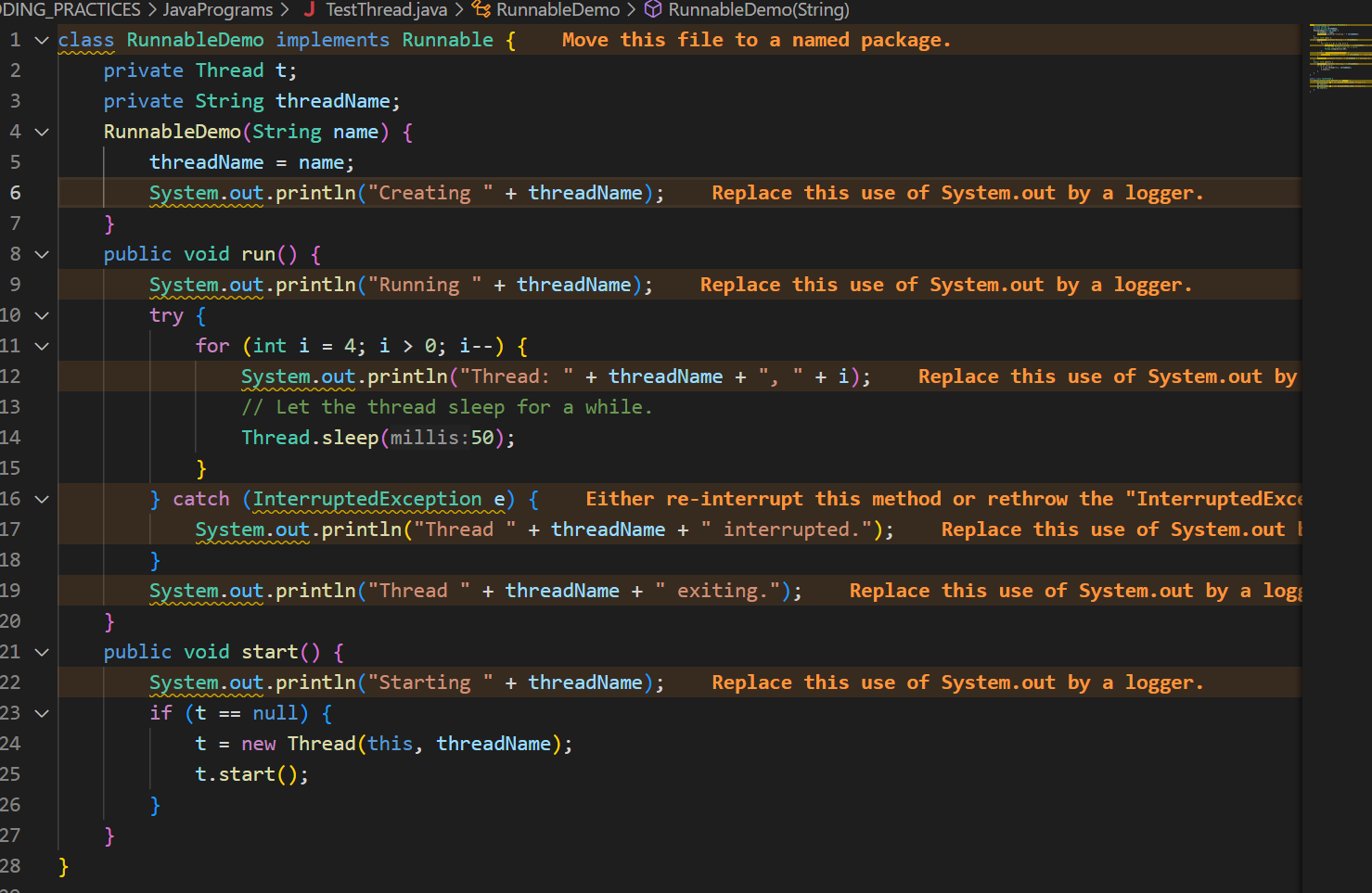
RunnableDemo R2 = new RunnableDemo( "Thread-2");

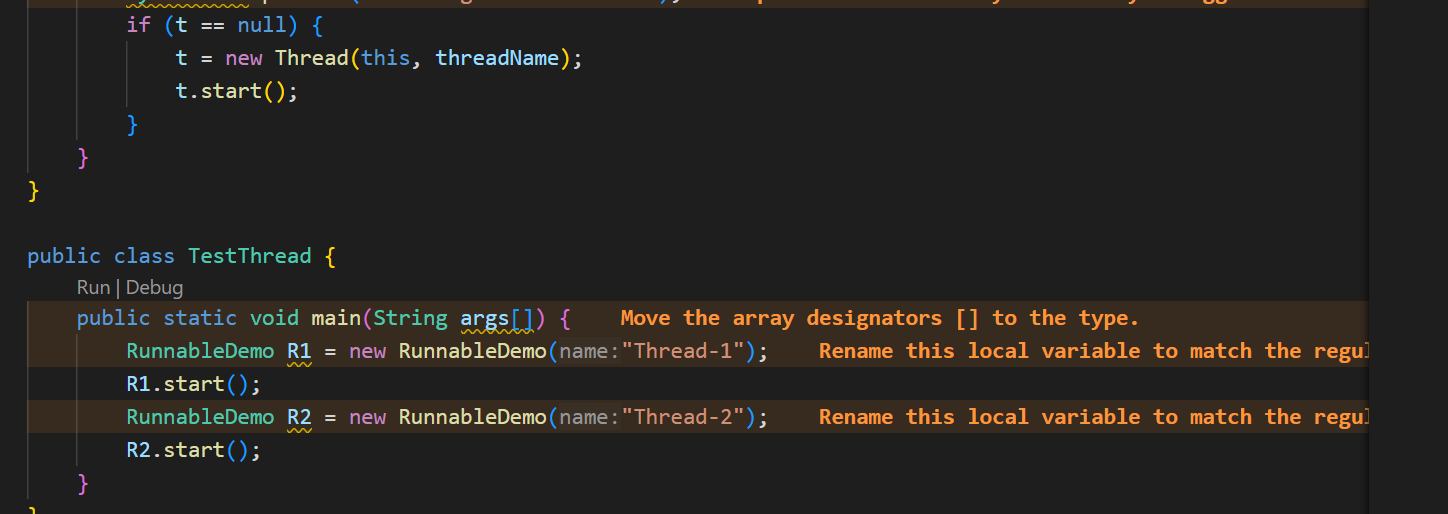
R2.start();

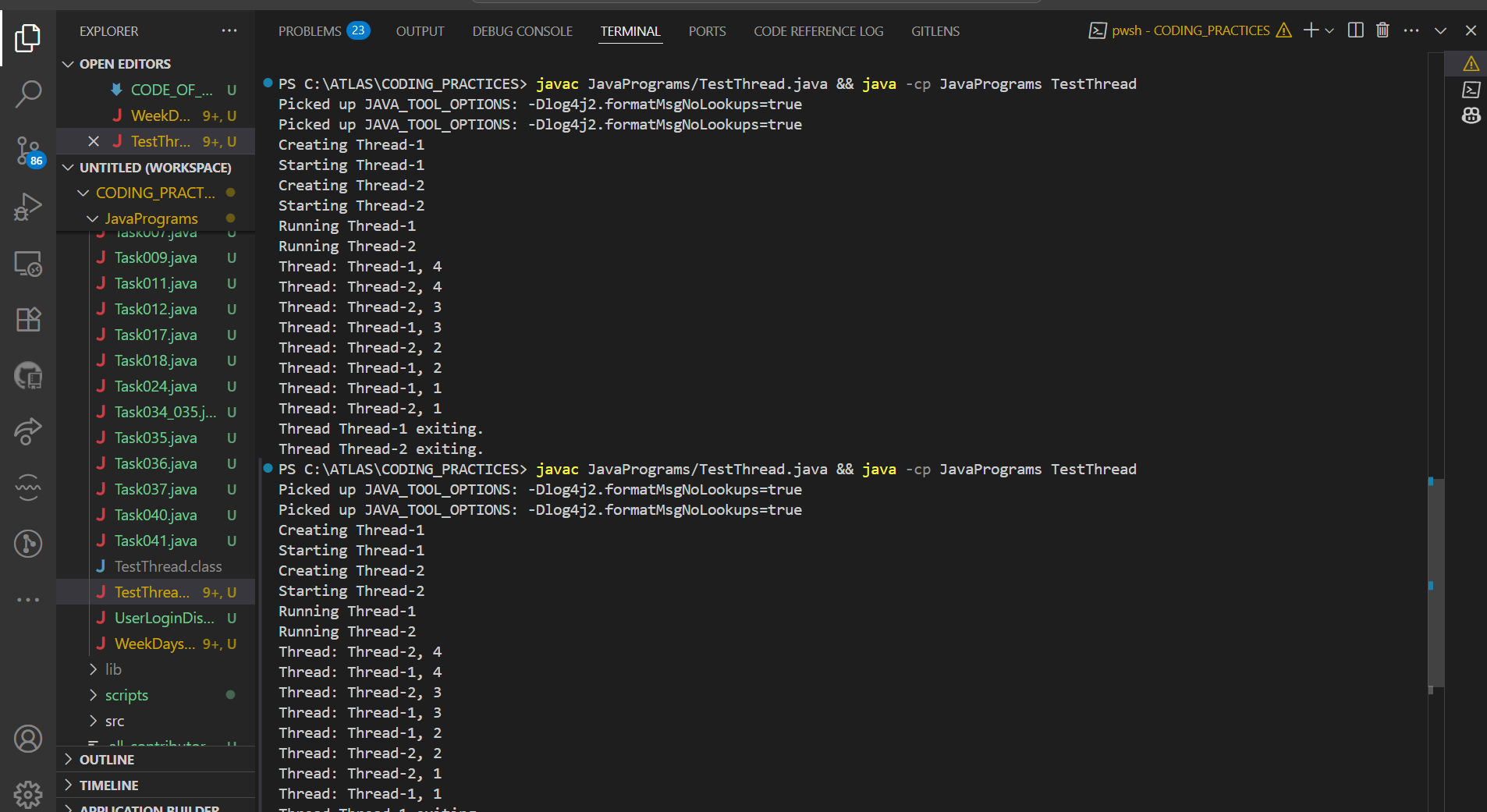
}

}

Solution :







EXPLAINATION :

1.Class Definition

RunnableDemo implements the Runnable interface, which means it must define a run() method.

2. Constructor

When you create a RunnableDemo object, it takes a thread name as a parameter and prints Creating <threadName>.

3. start() Method

The start() method checks if the thread (t) is null.

If it is, it creates a new Thread object, passing this (the current RunnableDemo object) and the thread name.

It then starts the thread, which calls the run() method.

4. run() Method

When the thread starts, it prints Running <threadName>.

It enters a loop from 4 down to 1, printing the thread name and the current number each time.

After each print, it sleeps for 50 milliseconds.

If interrupted, it prints an interruption message.

After the loop, it prints that the thread is exiting.

5. main() Method

In main, two RunnableDemo objects are created: one named "Thread-1" and one named "Thread-2".

start() is called on both, so both threads run at the same time.

6. Output

The output from both threads will be mixed together, showing that they are running concurrently.

Summary : Two threads are created and started.

Each thread counts down from 4 to 1, printing its name and number.

The output from both threads appears interleaved, showing parallel execution

Task 3 : In the above code … try extending Thread class… and observe the output..

Understand:

ex:

Public Class1 extends Thread{ }

Or

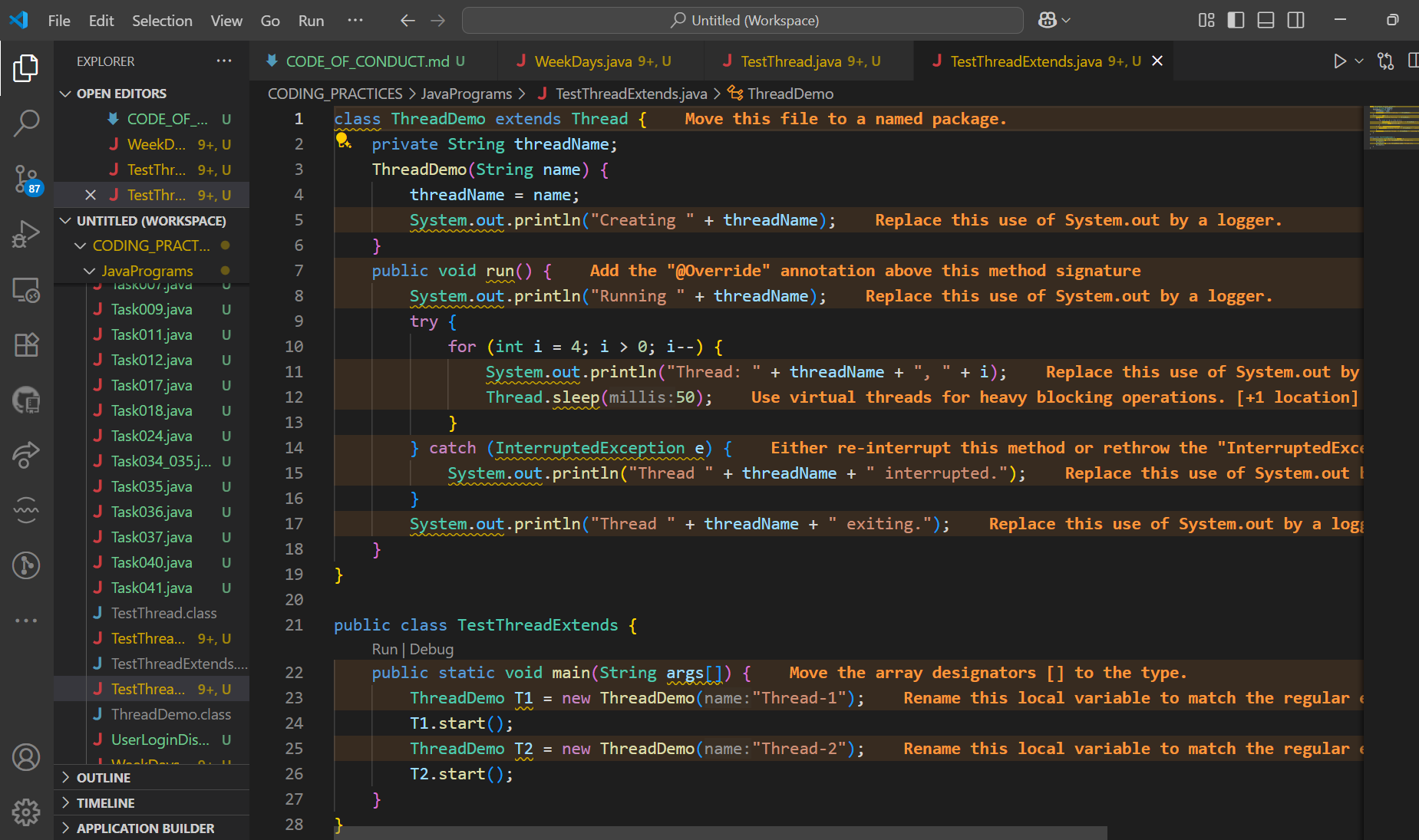
4public Class1 extends Class2 implements Runnable ( )

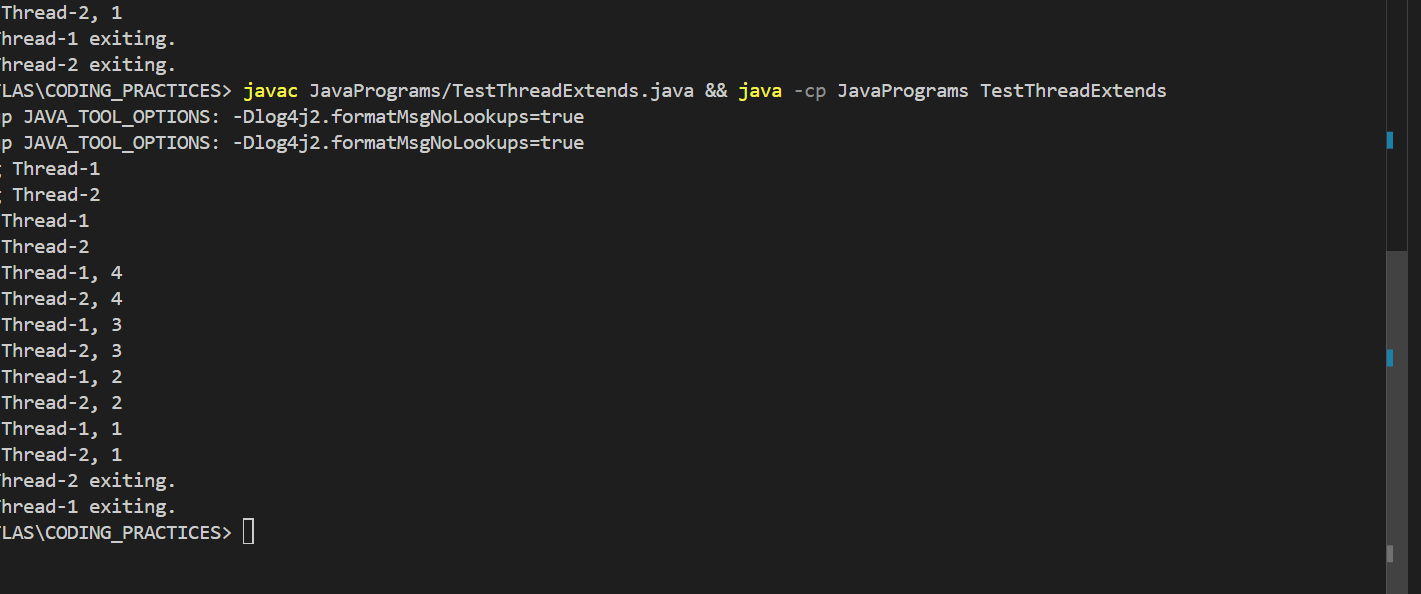
Solution : Both approaches (implements Runnable and extends Thread) allow concurrent execution.

The main difference is in how you structure your class:

extends Thread: Your class is a thread.

implements Runnable: Your class defines the work, and a separate Thread object runs it.





Task 4: class Counter {

private int count = 0;

public void increment() {

count++;

}

public int getCount() {

return count;

}

}

class ThreadDemo extends Thread {

Counter counter;

ThreadDemo(Counter counter) {

this.counter = counter;

}

public void run() {

for (int i = 0; i < 10; i++) {

counter.increment();

}

}

}

public class Main {

public static void main(String[] args) {

Counter counter = new Counter();

ThreadDemo t1 = new ThreadDemo(counter);

ThreadDemo t2 = new ThreadDemo(counter);

t1.start();

t2.start();

try {

t1.join();

t2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Final count: " + counter.getCount());

}

}

Solution :

Step-by-Step Explanation

Counter Class

Has a private integer count.

increment() increases count by 1.

getCount() returns the current value of count.

ThreadDemo Class

Extends Thread.

Takes a Counter object in its constructor and stores it.

In the run() method, it calls counter.increment() 10 times.

Main Class

Creates a single Counter object.

Creates two ThreadDemo threads, both sharing the same Counter.

Starts both threads.

Waits for both threads to finish using join().

Prints the final value of count.

What Should Happen

Each thread increments the counter 10 times.

Ideally, the final count should be 20.

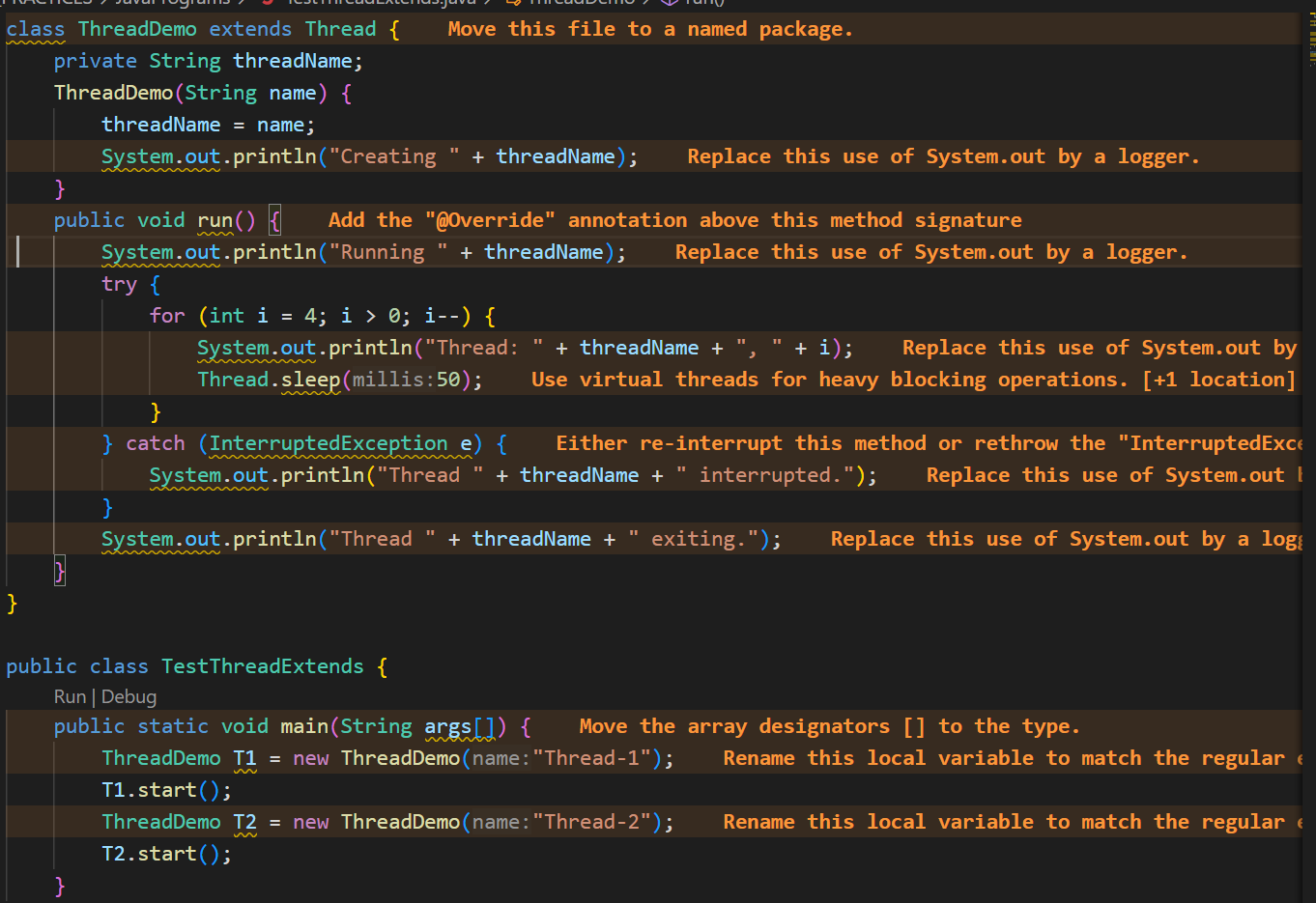
Important Note

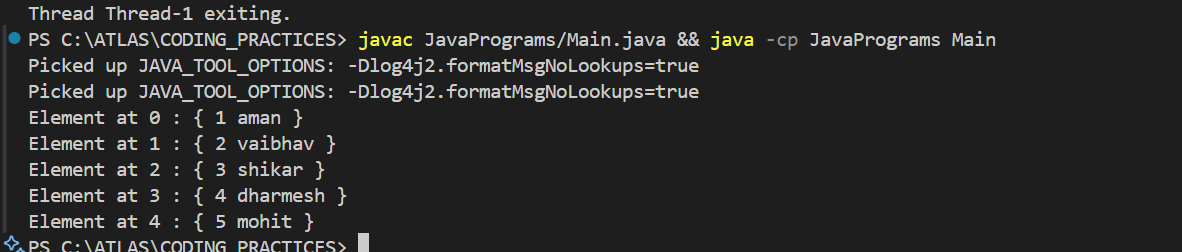
Because increment() is not synchronized, both threads may sometimes update count at the same time, causing a race condition.

This means the final count may sometimes be less than 20.

Summary:

This program demonstrates how multiple threads can share and update a single object, and highlights the need for synchronization when updating shared data in a multithreaded environment.





Task 5: Use synchronized method:

Hint: 1. Synchronized Method: Synchronize the entire method to ensure only one thread can execute it at a time.

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

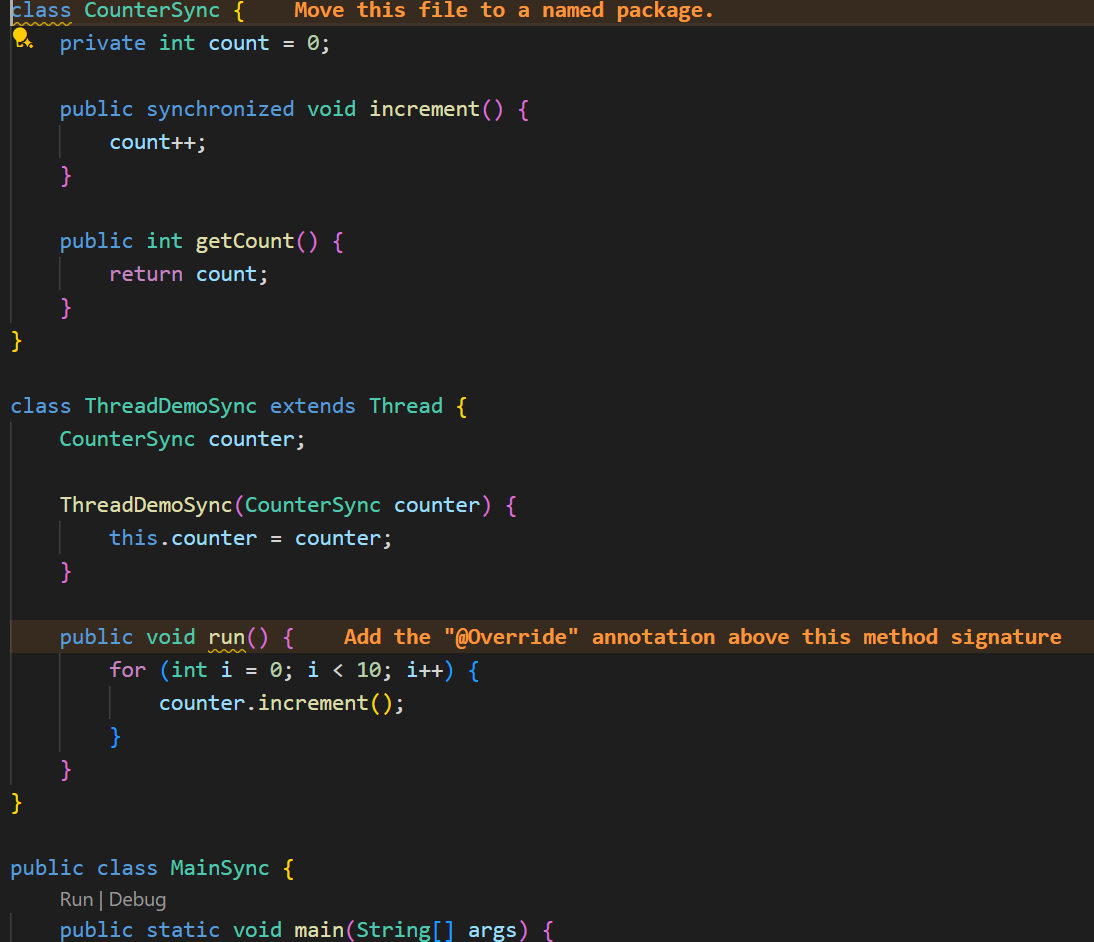
public int getCount() {

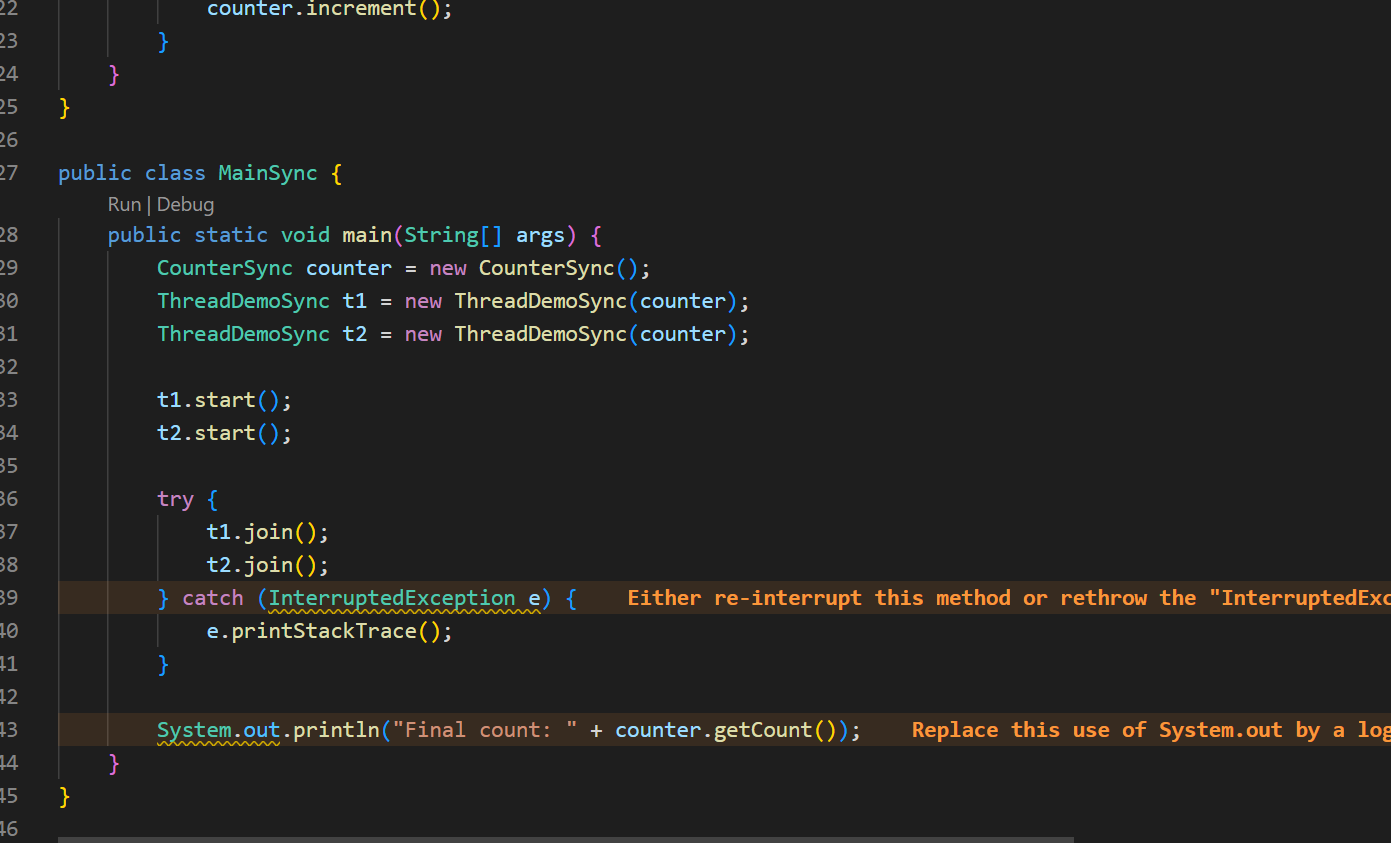
return count;

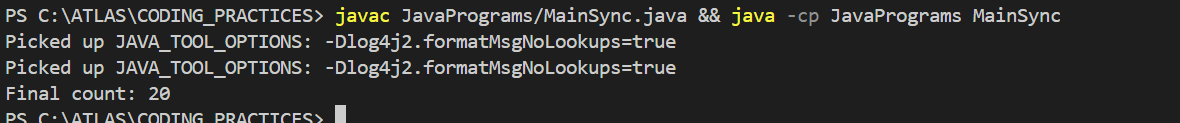
}

}

Solution :







Task 6: Using Sync Block

Hint: 2. Synchronized Block: Synchronize a block of code instead of the entire method, providing more control and efficiency.

class Counter {

private int count = 0;

public void increment() {

synchronized (this) {

count++;

}

}

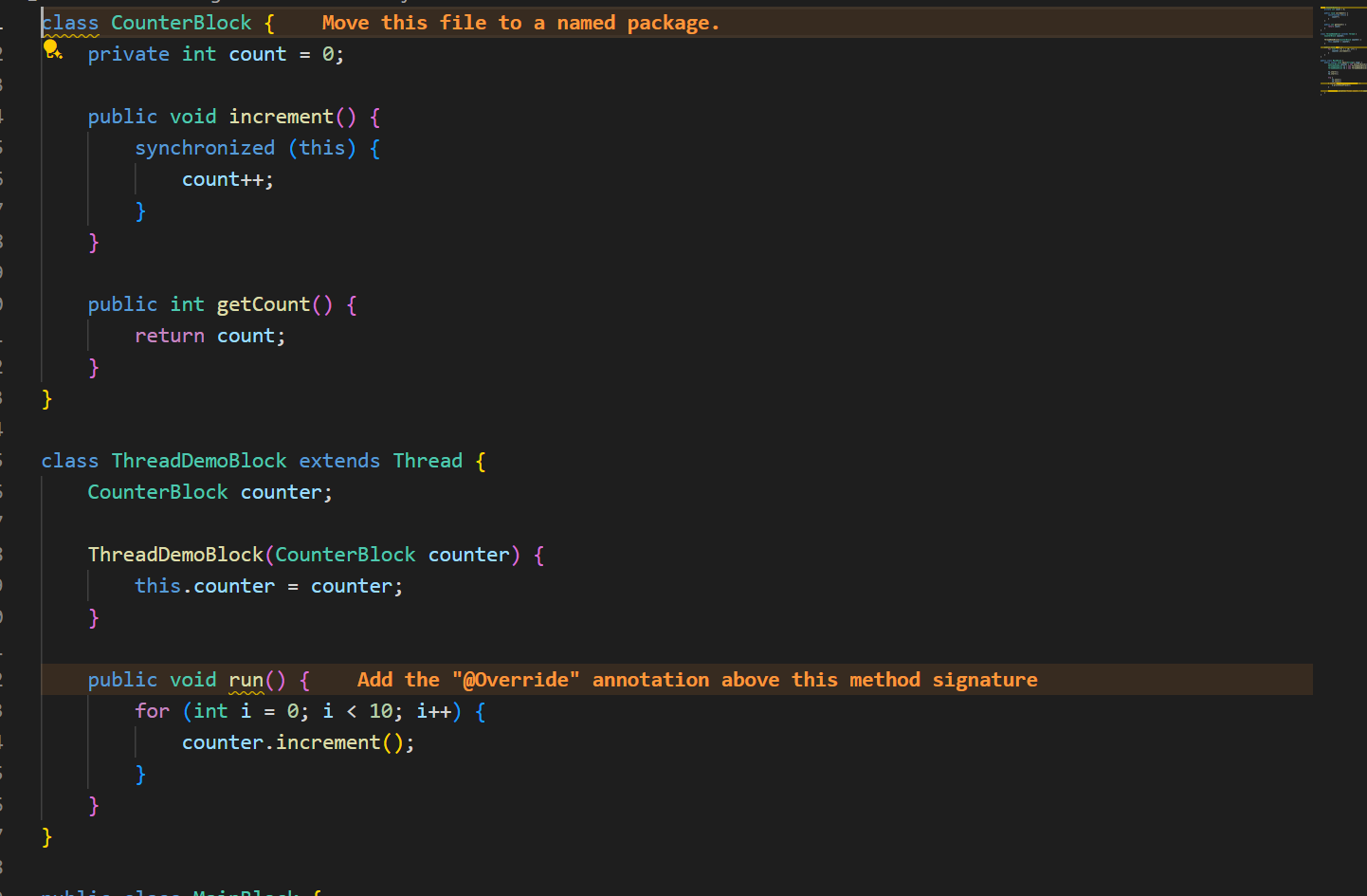
public int getCount() {

return count;

}

}

Solution :





Task 7:Using Static Sync

Hint:3. Static Synchronization: Synchronize static methods to ensure only one thread can execute them for the class, not the instance.

class Counter {

private static int count = 0;

public static synchronized void increment() {

count++;

}

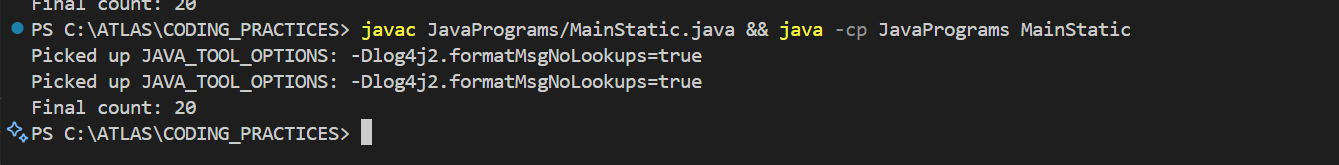
public static int getCount() {

return count;

}

}

Solution : 



Task 8:Using Locks

Hint: 4. Locks:

Use `java.util.concurrent.locks.Lock` for more sophisticated thread synchronization.

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

class Counter {

private int count = 0;

private final Lock lock = new ReentrantLock();

public void increment() {

lock.lock();

try {

count++;

} finally {

lock.unlock();

}

}

public int getCount() {

return count;

}

}

Solution :

Use a special lock to control access to a number

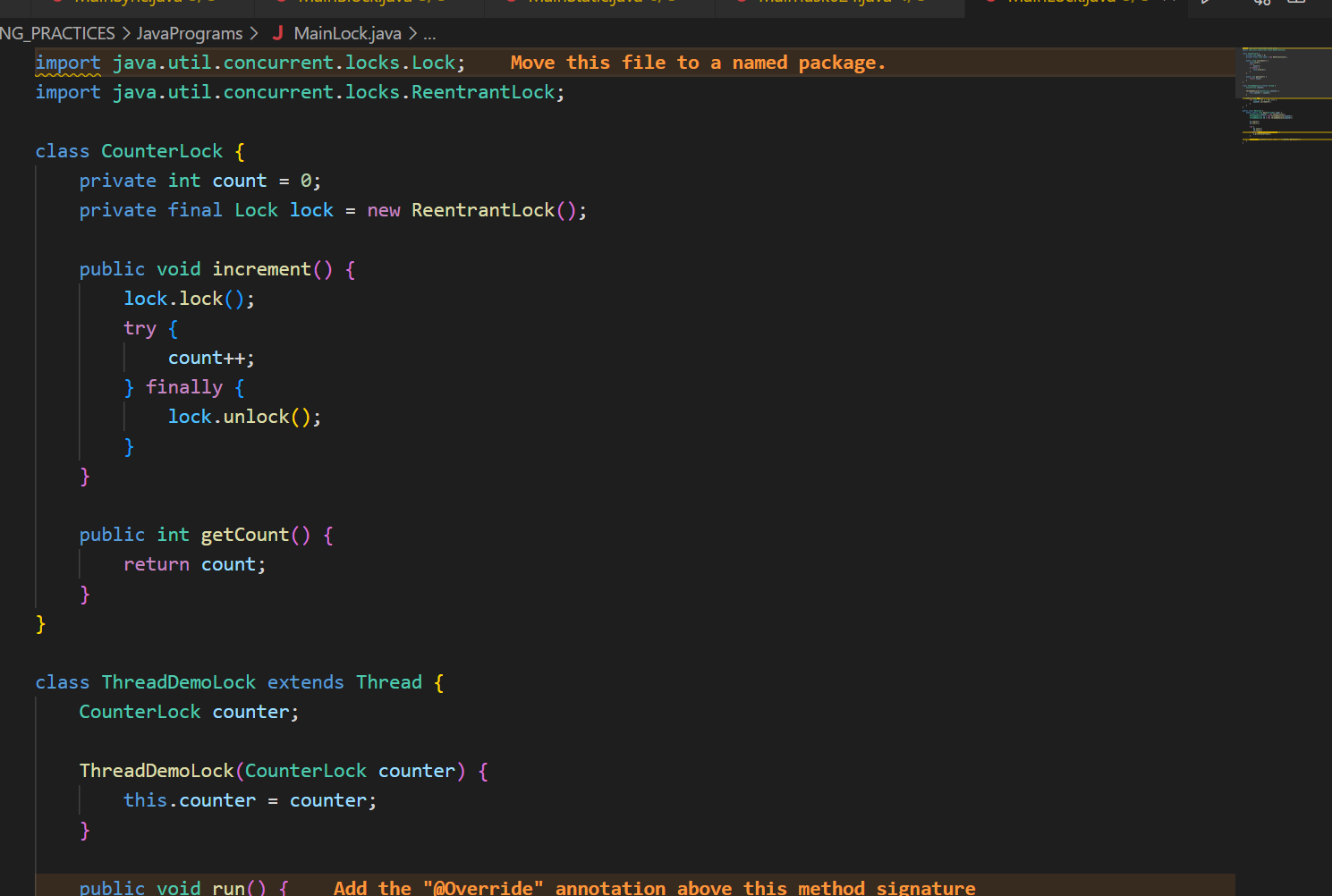
Step-by-step :

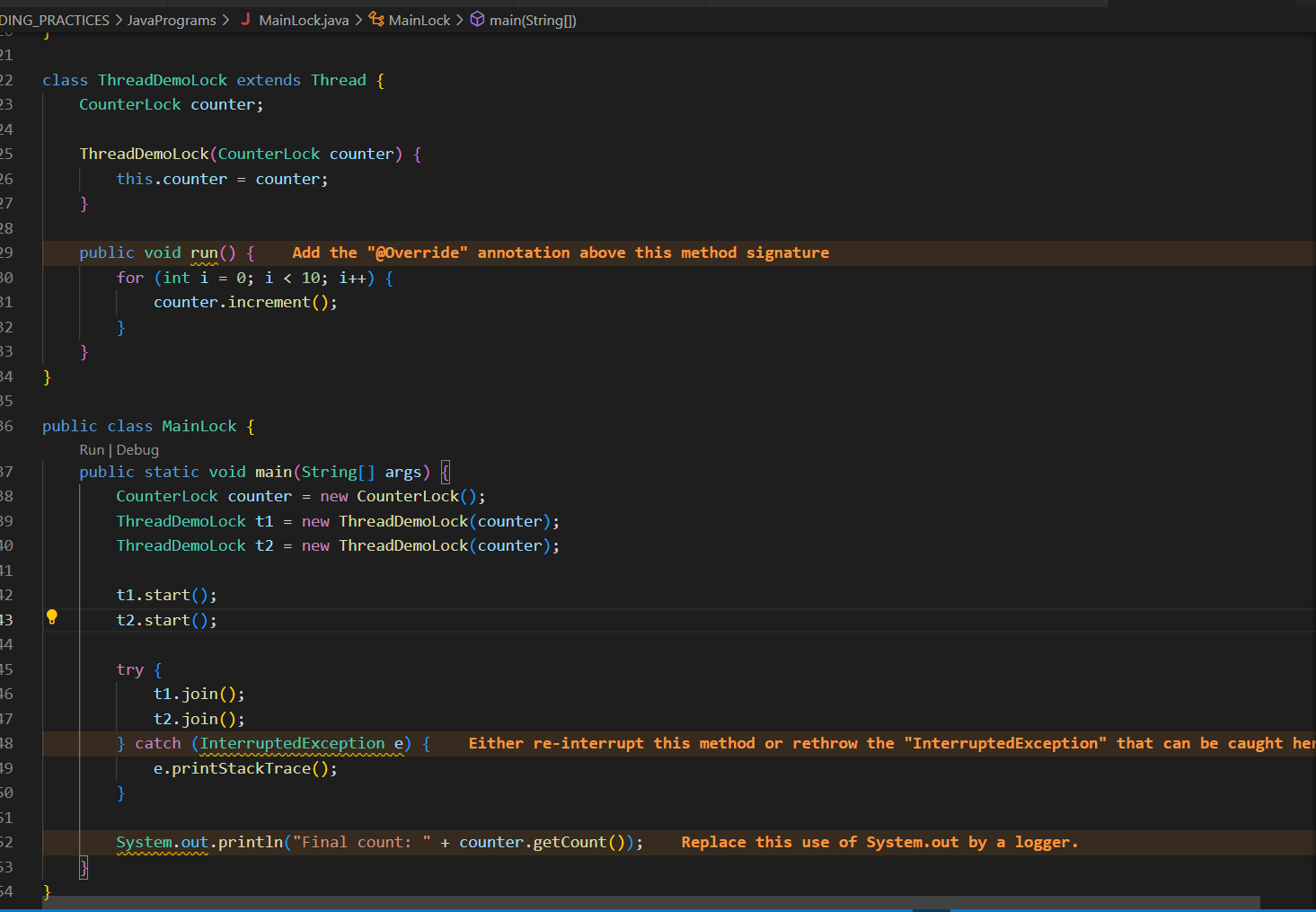
You make a CounterLock class with a number and a Lock object.

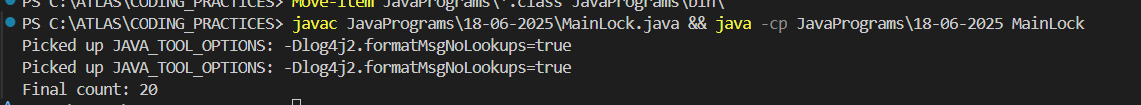
The increment() method locks before changing the number and unlocks after.

Two threads each increase the number 10 times.

You wait for both threads to finish. The final count is always 20, because the lock makes sure only one thread can change the number at a time.







Task 10:

Dead Lock 👍

Example of Deadlock

class Resource {

synchronized void method1(Resource r) {

System.out.println(Thread.currentThread().getName() + " is executing method1");

try { Thread.sleep(100); } catch (InterruptedException e) {}

r.method2(this);

}

synchronized void method2(Resource r) {

System.out.println(Thread.currentThread().getName() + " is executing method2");

try { Thread.sleep(100); } catch (InterruptedException e) {}

r.method1(this);

}

}

public class DeadlockExample {

public static void main(String[] args) {

final Resource r1 = new Resource();

final Resource r2 = new Resource();

Thread t1 = new Thread(() -> r1.method1(r2), "Thread-1");

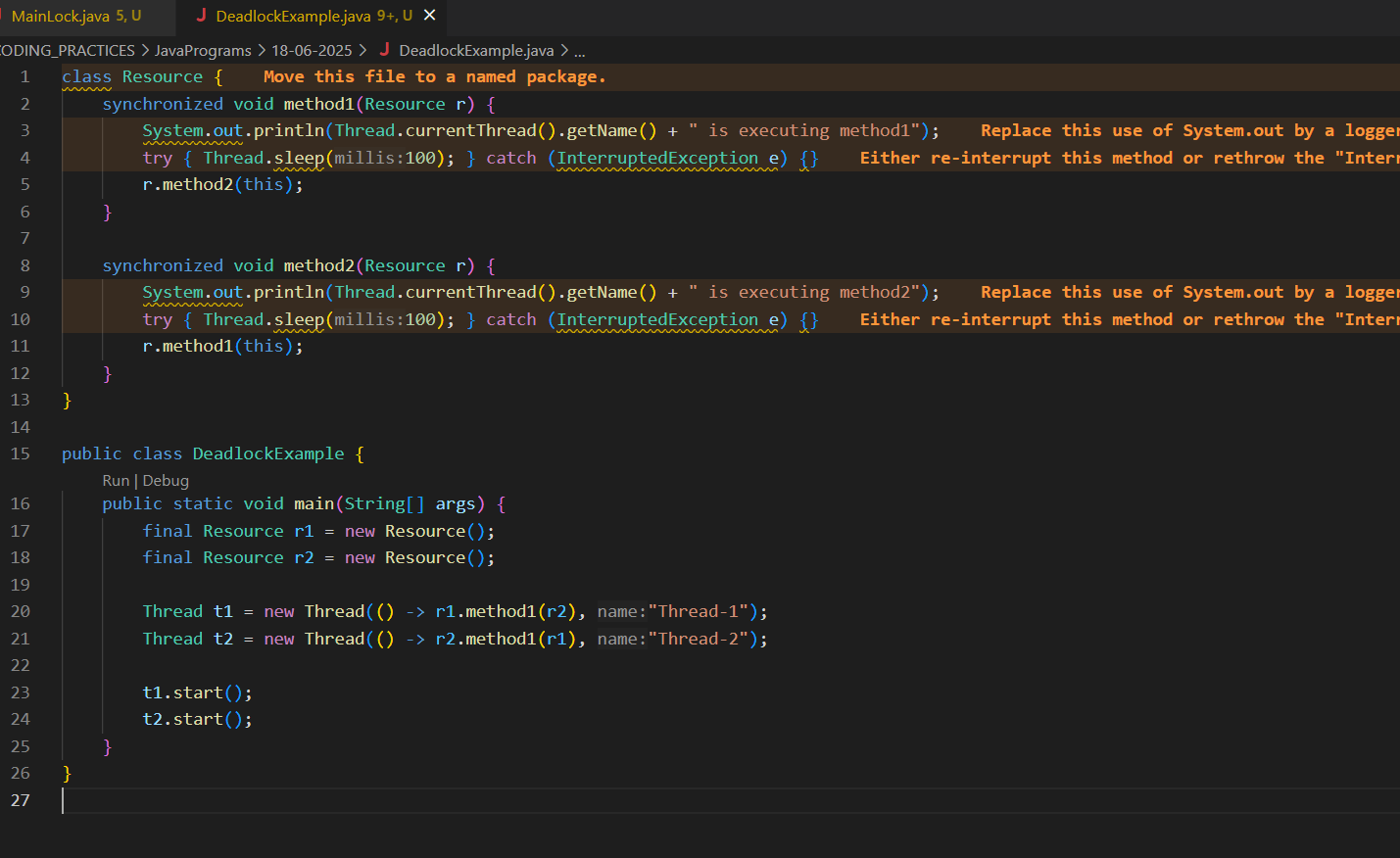
Thread t2 = new Thread(() -> r2.method1(r1), "Thread-2");

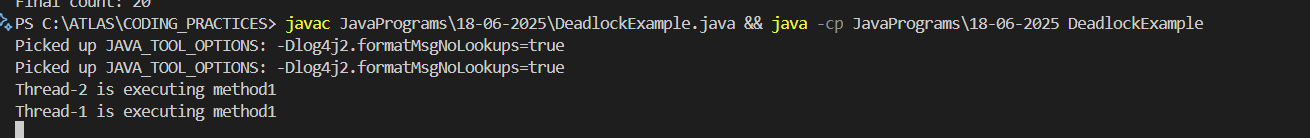
t1.start();

t2.start();

}

}

Solution : 



Task 11:

Inter- thread communication…

Example of Inter-thread Communication

class SharedResource {

private boolean ready = false;

synchronized void produce() {

try {

while (ready) {

wait();

}

System.out.println("Producing...");

ready = true;

notify();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

synchronized void consume() {

try {

while (!ready) {

wait();

}

System.out.println("Consuming...");

ready = false;

notify();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class InterThreadCommunicationExample {

public static void main(String[] args) {

SharedResource resource = new SharedResource();

Thread producer = new Thread(resource::produce);

Thread consumer = new Thread(resource::consume);

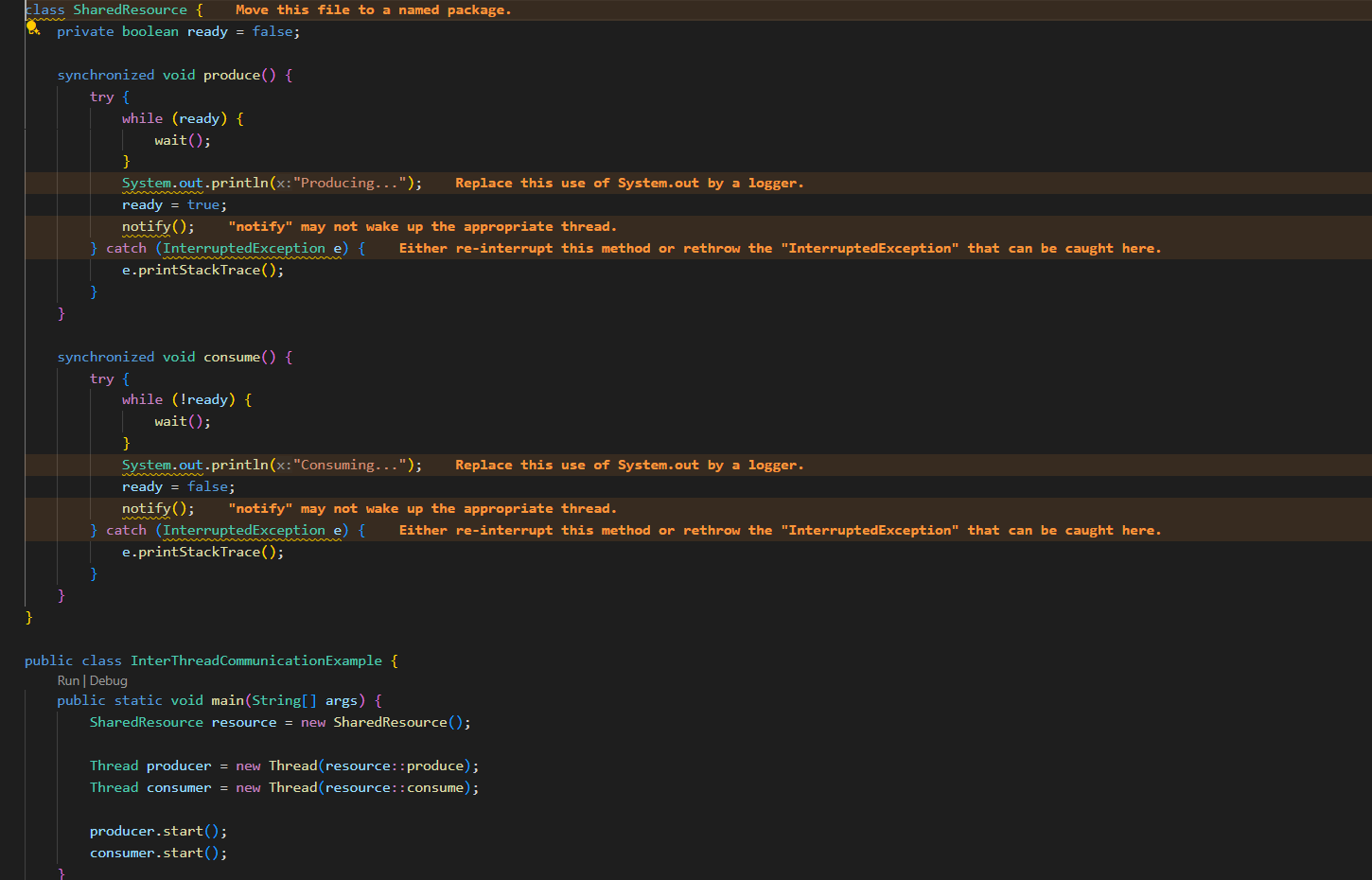
producer.start();

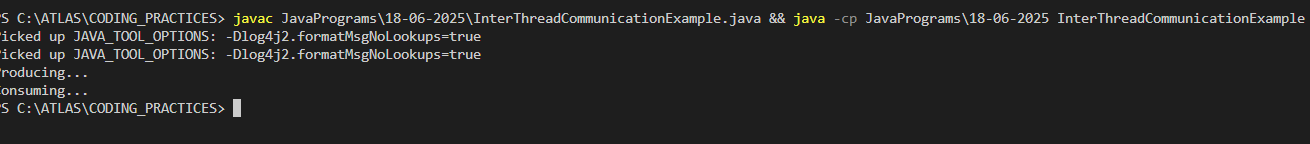
consumer.start();

}

}

Solution :





Task 12:

Interrupting a thread

Example of Interrupting a Thread

class InterruptibleThread extends Thread {

public void run() {

try {

while (!Thread.currentThread().isInterrupted()) {

System.out.println("Thread is running");

Thread.sleep(100);

}

} catch (InterruptedException e) {

System.out.println("Thread was interrupted");

}

}

}

public class InterruptExample {

public static void main(String[] args) {

InterruptibleThread thread = new InterruptibleThread();

thread.start();

try {

Thread.sleep(500);

thread.interrupt();

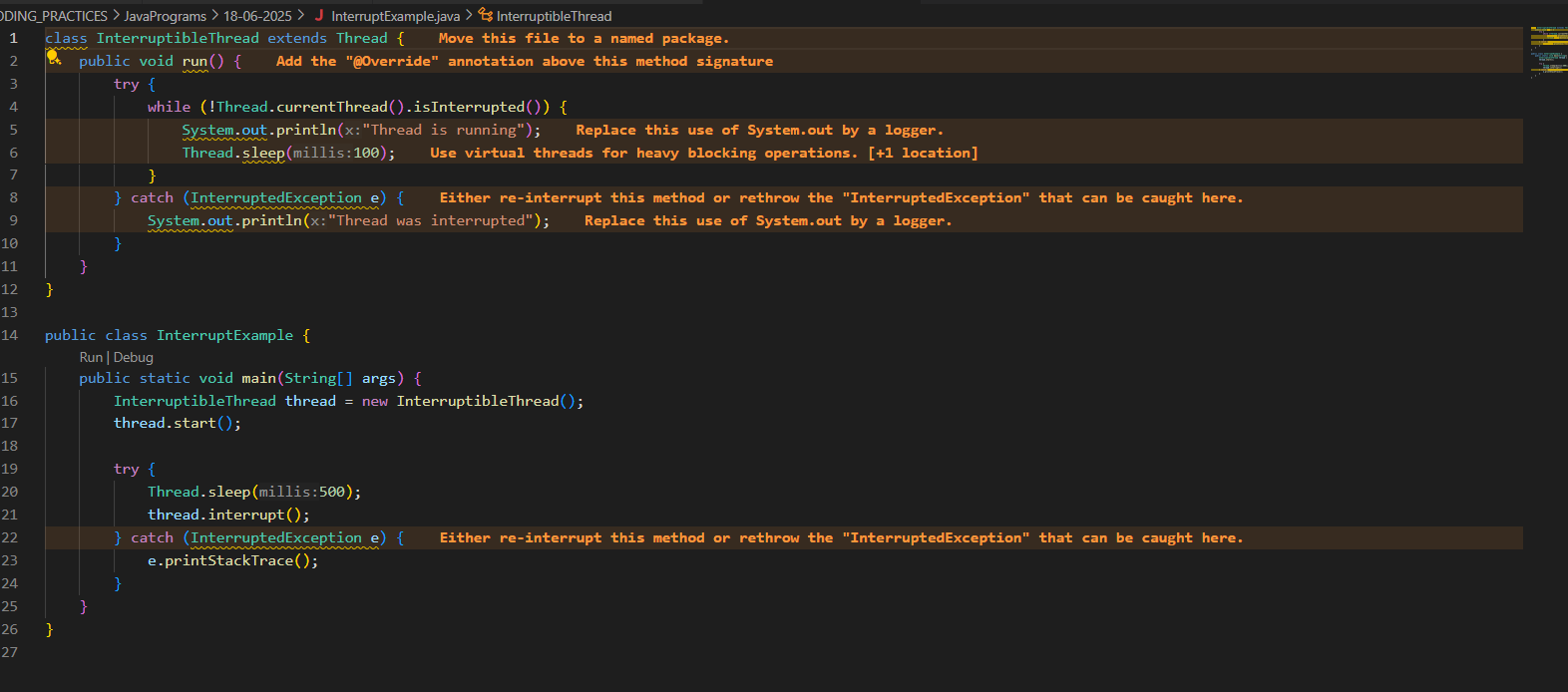
} catch (InterruptedException e) {

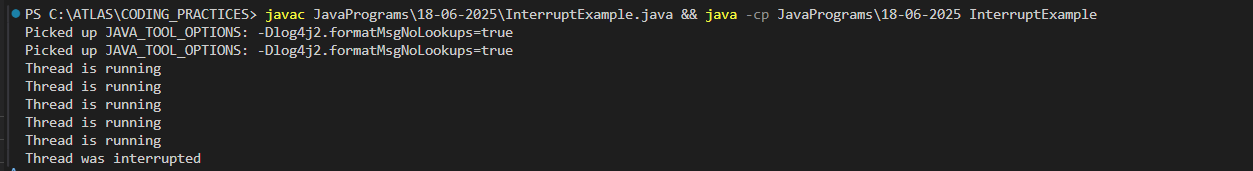
e.printStackTrace();

}

}

}

Solution : 



Task 14::What is Thread pool?

Solution : A Thread Pool is a collection of pre-initialized and reusable threads that execute tasks submitted to it, avoiding the overhead of creating and destroying threads for every task. They are widely used in Java applications to manage and efficiently utilize system resources, especially in scenarios where there are a large number of short-lived tasks that need to be executed.

========================================================================

File handling:

=======================================================================

Task 15:

Run the below code and see the file with the given name created or not..

Run it again with I like India instead of I love India.. And see the file …

public class WriteByte

{

public static void main(String args[])

{

File f1=new File(“FileName01.txt”); \\ to create new file FileOutputStream outfile = null;

byte Text[] = {'I',’ ‘,’'L','O','V','E',’ ‘,'I','N','D','I’,’A'};

try

{

outfile = new FileOutputStream(f1);

outfile.write(Text);

}

catch(IOException e)

{

System.out.println(e);

System.exit(-1);

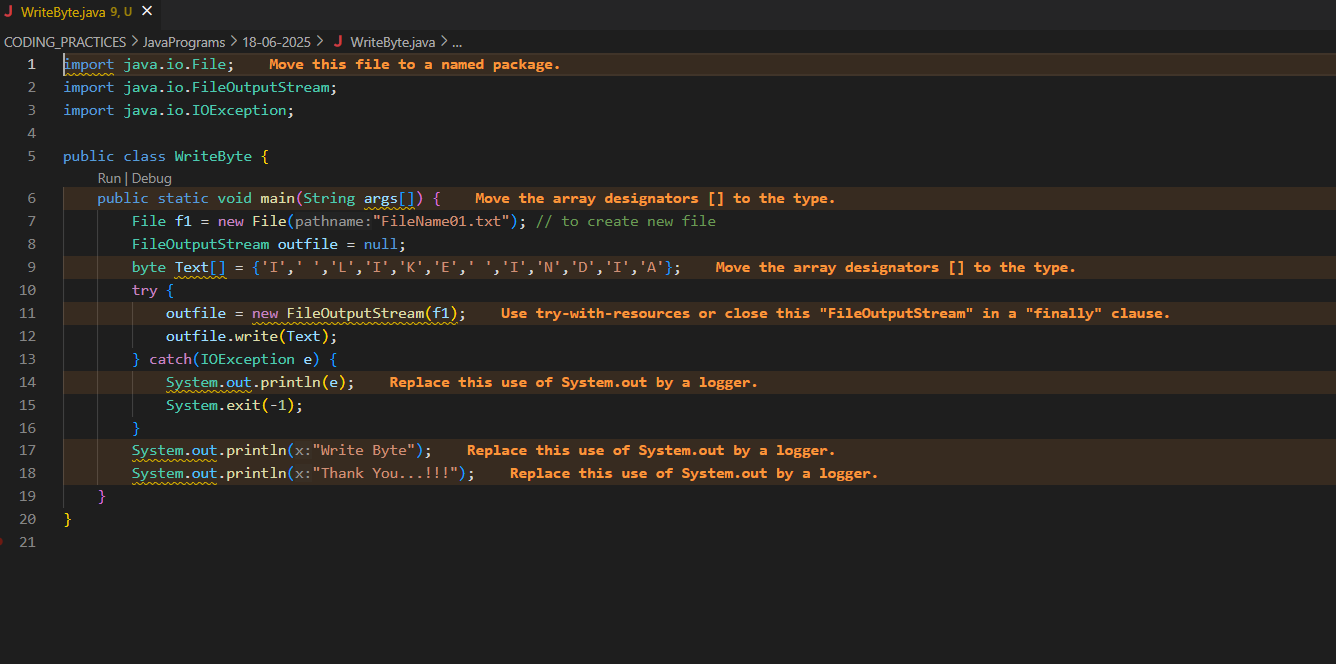
}

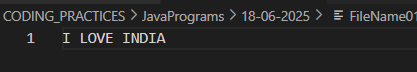
System.out.println("Write Byte");

System.out.println("Thank You...!!!");

}

}

Solution : 



Task 16: Try this code to see the output …

Write a program which reads byte from file.

import java.io.\*;

public class ReadingByte

{

public static void main(String args[])

{

FileInputStream infile = null;

int b;

try

{

infile = new FileInputStream("FileName01.txt");

while((b = infile.read()) != -1)

{

System.out.println((char)b);

}

infile.close();

}

catch(IOException e)

{

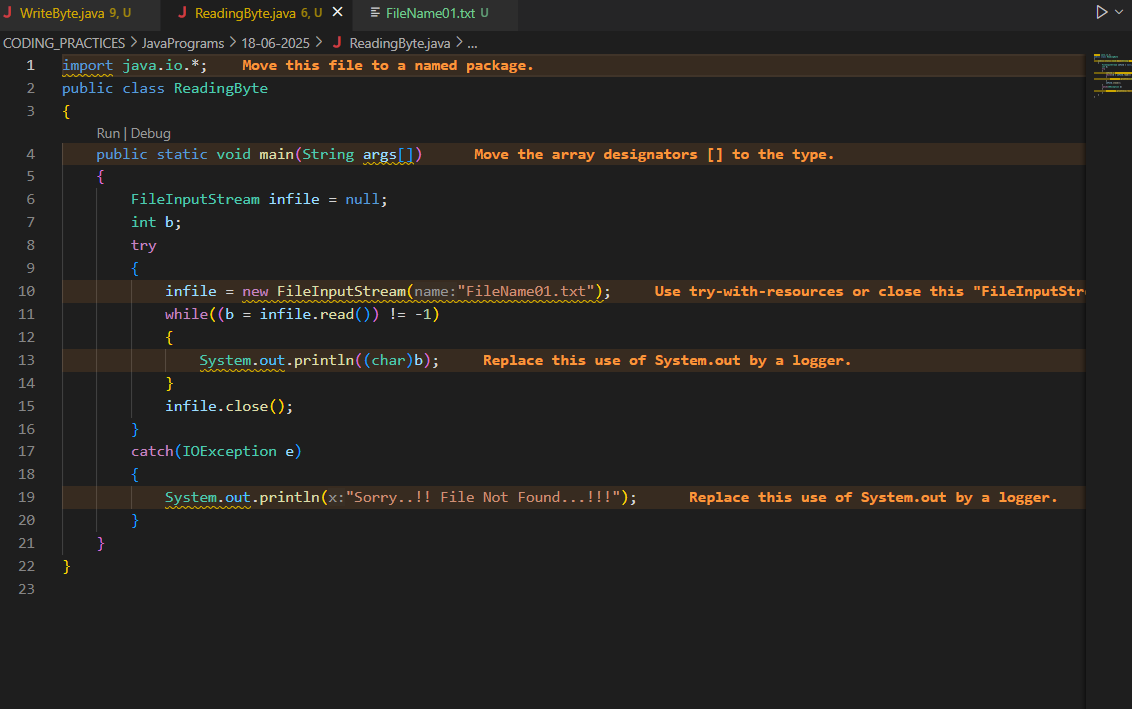
System.out.println("Sorry..!! File Not Found...!!!");

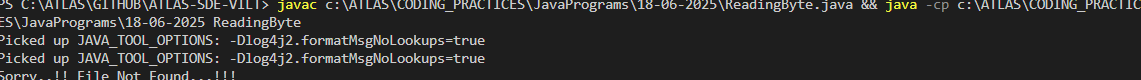
}

}

}

Solution :





Task 17:

import java.io.\*;

import java.util.\*;

public class WriteByte\_1

{

public static void main(String args[]) {

FileOutputStream outfile = null;

//String s=args[0]; // to input string from command line Scanner sc=new Scanner(System.in);

String s=sc.nextLine();

byte b1[] = s.getBytes();

try

{

outfile = new FileOutputStream("in.txt");

outfile.write(b1);

}

catch(IOException e)

{

System.out.println(e);

System.exit(-1);

}

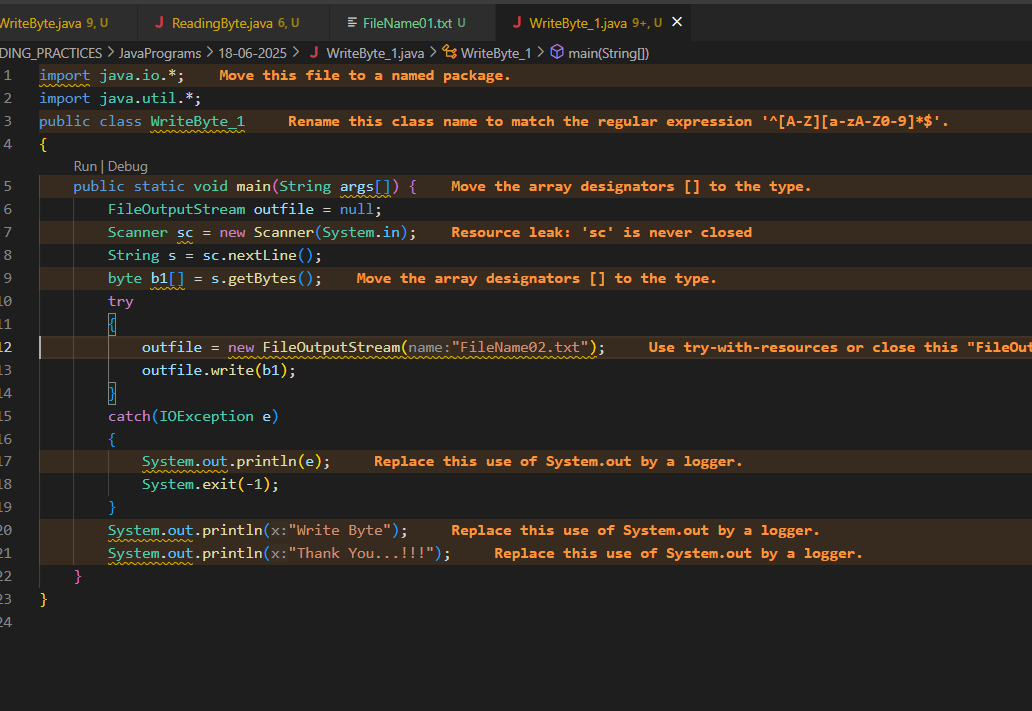
System.out.println("Write Byte");

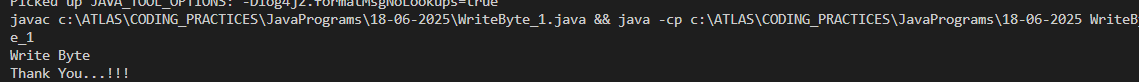
System.out.println("Thank You...!!!");

}

}

Solution :





FileReader and FileWriter

Task 18:

Write a program which creates file and writes character into that file.

import java.io.\*;

Class CharacterWrite {

public static void main(String args[]) {

File f1=new File("FileName03.txt");

FileWriterfw = null;

try {

fw=new FileWriter(f1);

fw.write("ahmedabad \n");

fw.write(" baroda \n");

fw.close();

}

catch(FileNotFoundException e)

{

System.out.println("Sorry..!! File Not Found...!!!");

}

catch(IOException e)

{

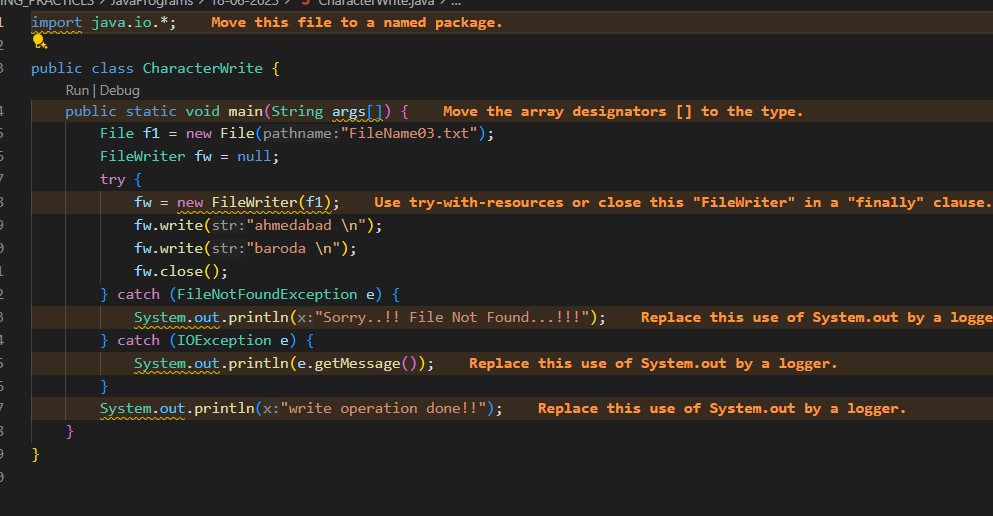
System.out.println(e.getMessage());

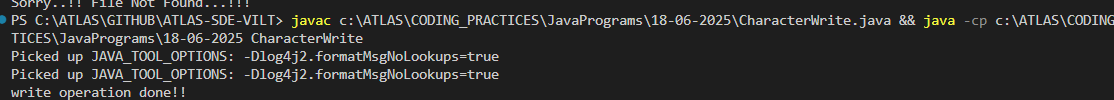
}

System.out.println(“ write operation done!!”);

}

}





Task 19

Write a program which reads character from file.

import java.io.\*;

Class Readchar

{

public static void main(String args[])

{

FileReader fr =null;

try

{

fr = new FileReader("FileName03.txt");

int ch;

while((ch = fr.read()) != -1)

{

System.out.print((char)ch);

}

System.out.println("Reading complete");

fr.close();

}

catch(FileNotFoundException e)

{

System.out.println("Sorry..!! File Not Found...!!!");

}

catch(IOException e)

{

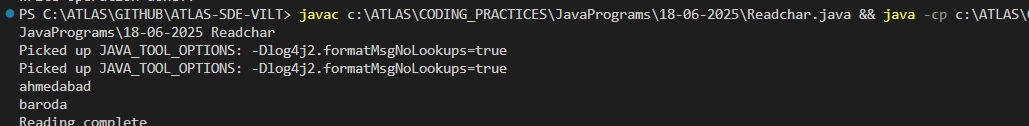
System.out.println(e.getMessage());

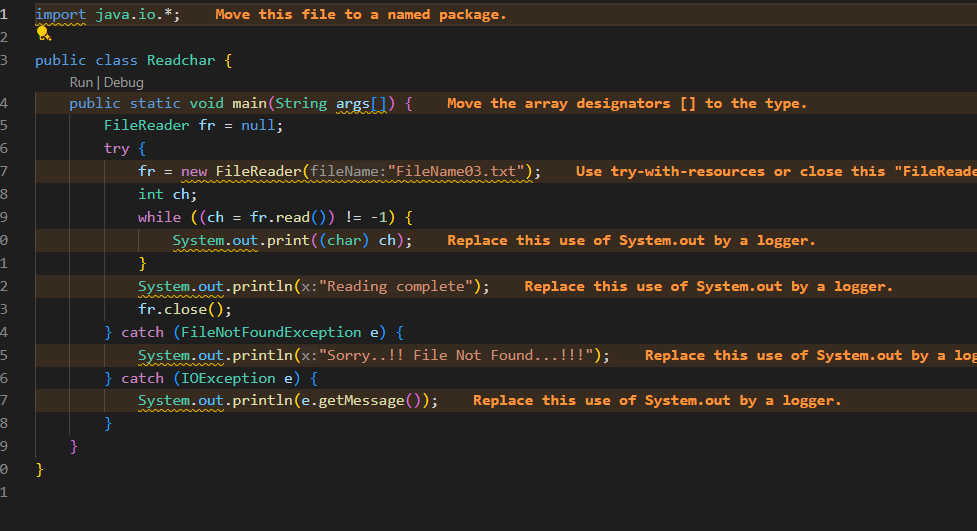
}

}

}

Solution :





Task 20

Write a program to read one byte at a time from a file and copy it into another file immediately.

import java.io.\*;

Class CopyByte

{

public static void main(String args[])

{

try

{

byte b=0;

FileInputStream infile = new FileInputStream("NewFile01.txt");

FileOutputStreamoutfile = new FileOutputStream("NewFile05.txt");

Initialize byteread here….

while(byteread != -1)

{

b = (byte)infile.read();

outfile.write(b);

}

System.out.println("Byte Copied From in.txt to out.txt FIle ");

}

catch(FileNotFoundException e)

{

System.out.println("Sorry..!! File Not Found...!!!");

}

catch(IOException e)

{

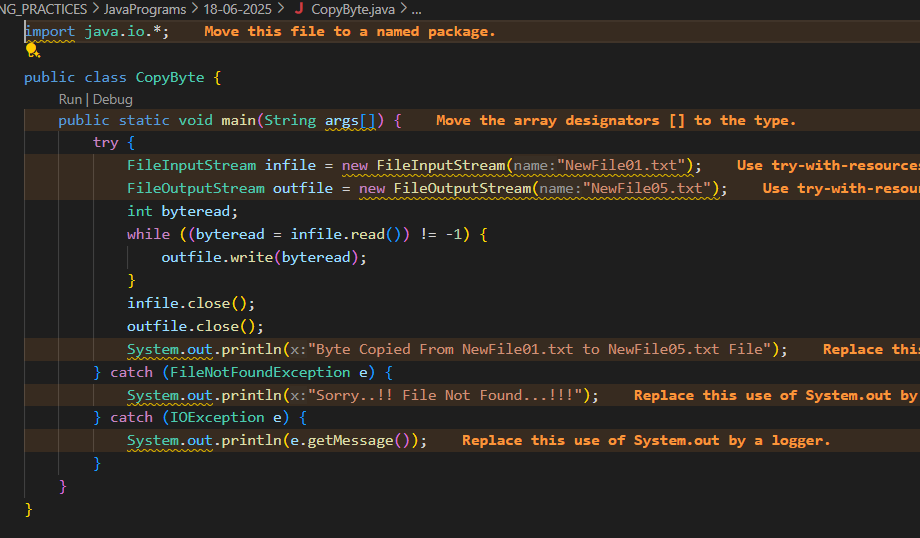
System.out.println(e.getMessage());

}

}

}

Solution :



Task 21:

Merging two files to 3rd file..

Write a program to merge two files in third file.

import java.io.\*;

classFileMergeDemo

{

public static void main(String args[])

{

try

{

FileInputStream file1 = new FileInputStream("File1.txt"); FileInputStream file2 = new FileInputStream("File2.txt"); SequenceInputStream file3 = new SequenceInputStream(file1, file2); BufferedInputStream br1 = new BufferedInputStream(file3); BufferedOutputStream br2 = new BufferedOutputStream(System.out); intch;

while((ch = br1.read())!=-1)

{

br2.write((char)ch);

}

br1.close();

br2.close();

file1.close();

file2.close();

System.out.println("Merge Two File Sucessfully ");

}

catch(IOException e)

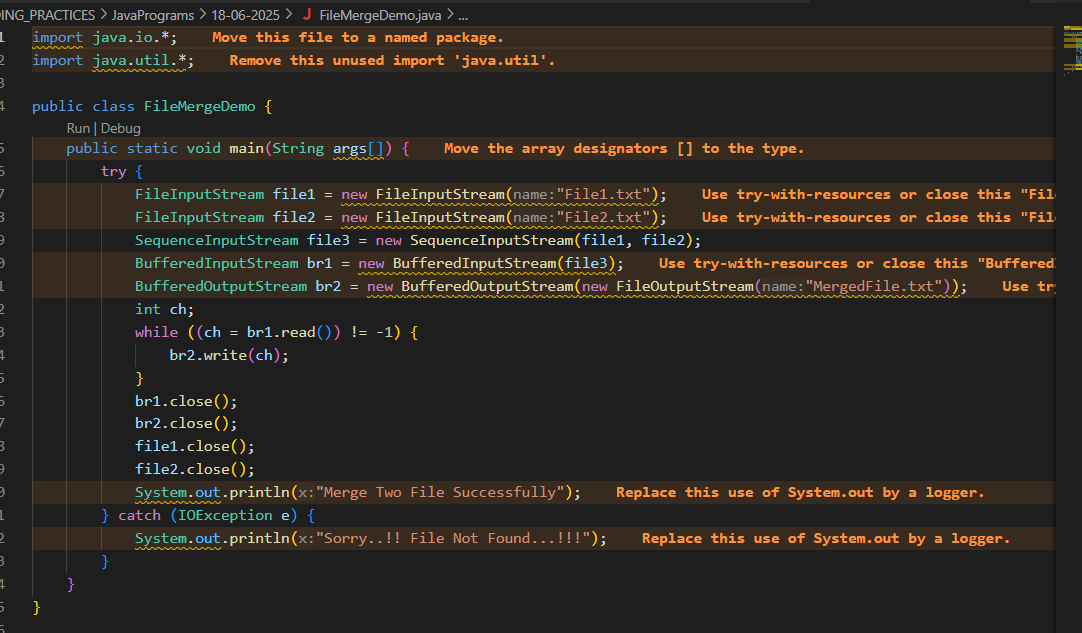
{

System.out.println("Sorry..!! File Not Found...!!!");

}

}

}

Solution : 

Task 22:

Write an application to rename a file. Use the renameTo() method of File to accomplish

/\*this task. The first command line argument is the old filename and the second is the newfilename.

\*/

import java.io.\*;

classFileRenameDemo

{

public static void main(String args[])

{

File f1 = new File(args[0]);

File f2 = new File(args[1]);

f1.renameTo(f2);

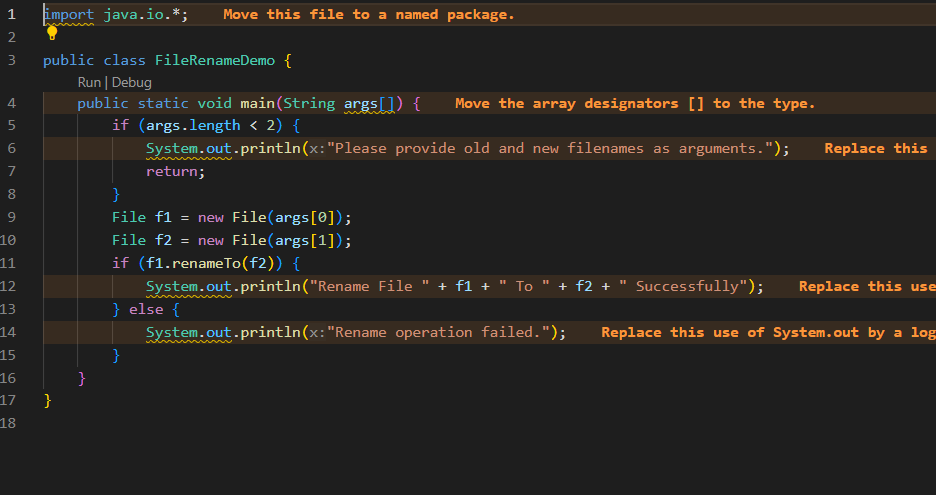
System.out.println("Rename File " +f1+" To "+f2+" Sucessfully "); }

}

Output :

javacFileRenameDemo.java

javaFileRenameDemo input1.txt abc.txt

Solution : 

Task 23 👍

==================================================

Buffered reader and writer — for large files to be read.

==================================================

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class ReadFileExample {

public static void main(String[] args) {

try (BufferedReader br = new BufferedReader(new FileReader("largefile.txt"))) {

String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

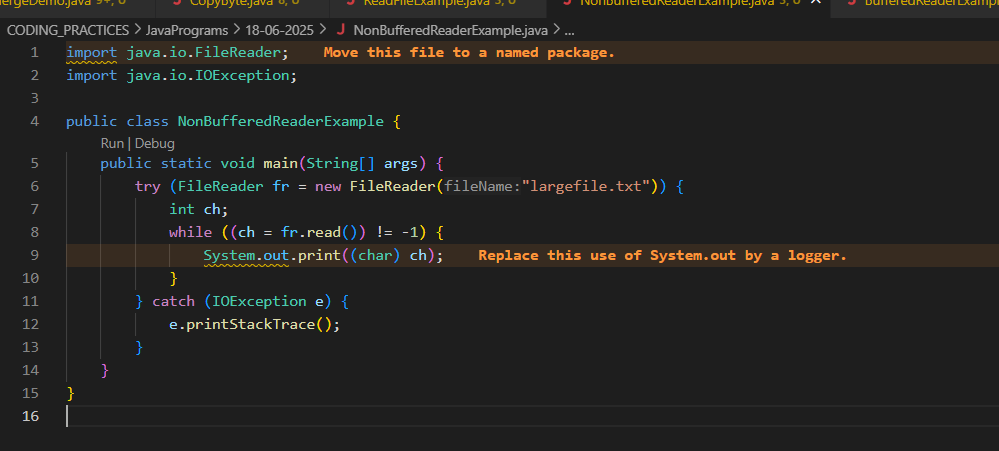
} catch (IOException e) {

e.printStackTrace();

}

}

}

Solution : 

Task 24 : non buffered

import java.io.FileReader;

import java.io.IOException;

public class NonBufferedReaderExample {

public static void main(String[] args) {

try (FileReader fr = new FileReader("largefile.txt")) {

int ch;

while ((ch = fr.read()) != -1) {

System.out.print((char) ch);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

Buffered

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class BufferedReaderExample {

public static void main(String[] args) {

try (BufferedReader br = new BufferedReader(new FileReader("largefile.txt"))) {

String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

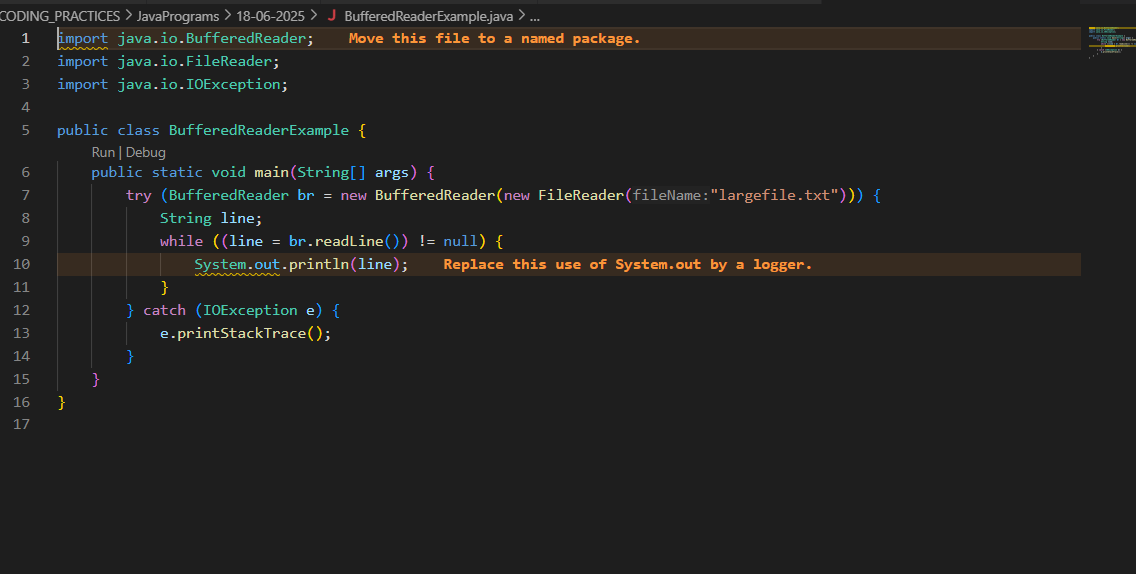
} catch (IOException e) {

e.printStackTrace();

}

}

}

Solution : 

Task 25:

Buffered Writer

import java.io. BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

public class BufferedWriterDemo {

public static void main(String[] args) {

String filePath = "example.txt";

String content = "Hello, World!\nThis is a BufferedWriter example.";

// Initialize BufferedWriter with a FileWriter

try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {

// Write content to the file

writer.write(content);

System.out.println("Content written to file.");

} catch (IOException e) {

System.err.println("An error occurred: " + e.getMessage());

}

}

}

Solution :

