DAY 10

1. What do you understand by a process?

A process is just a program that is currently running.

A process is a independent program in execution. When you run a Java Programe(like Java, My App), the java Virtual Machine(JVM) is started aand it runs as a process in the operating system.

1. What is a thread?

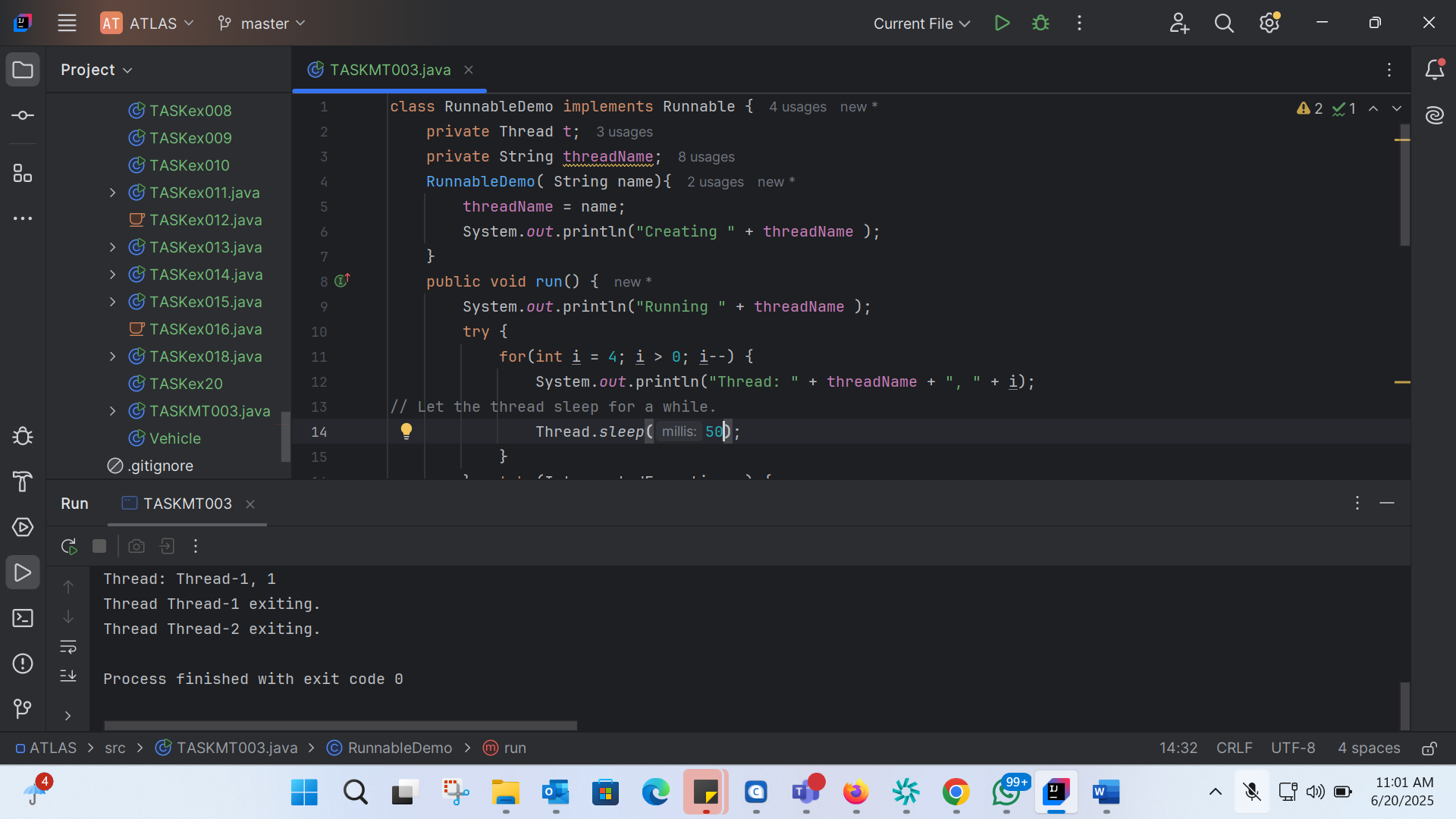
A thread in Java is a lightweight unit of execution within a process. It allows a program to do multiple tasks as the same time(concurrently.)

A process can have multiple threads running inside it.

All threads in the same process share memory and resources.

Threads allow for multitasking within a single programme.

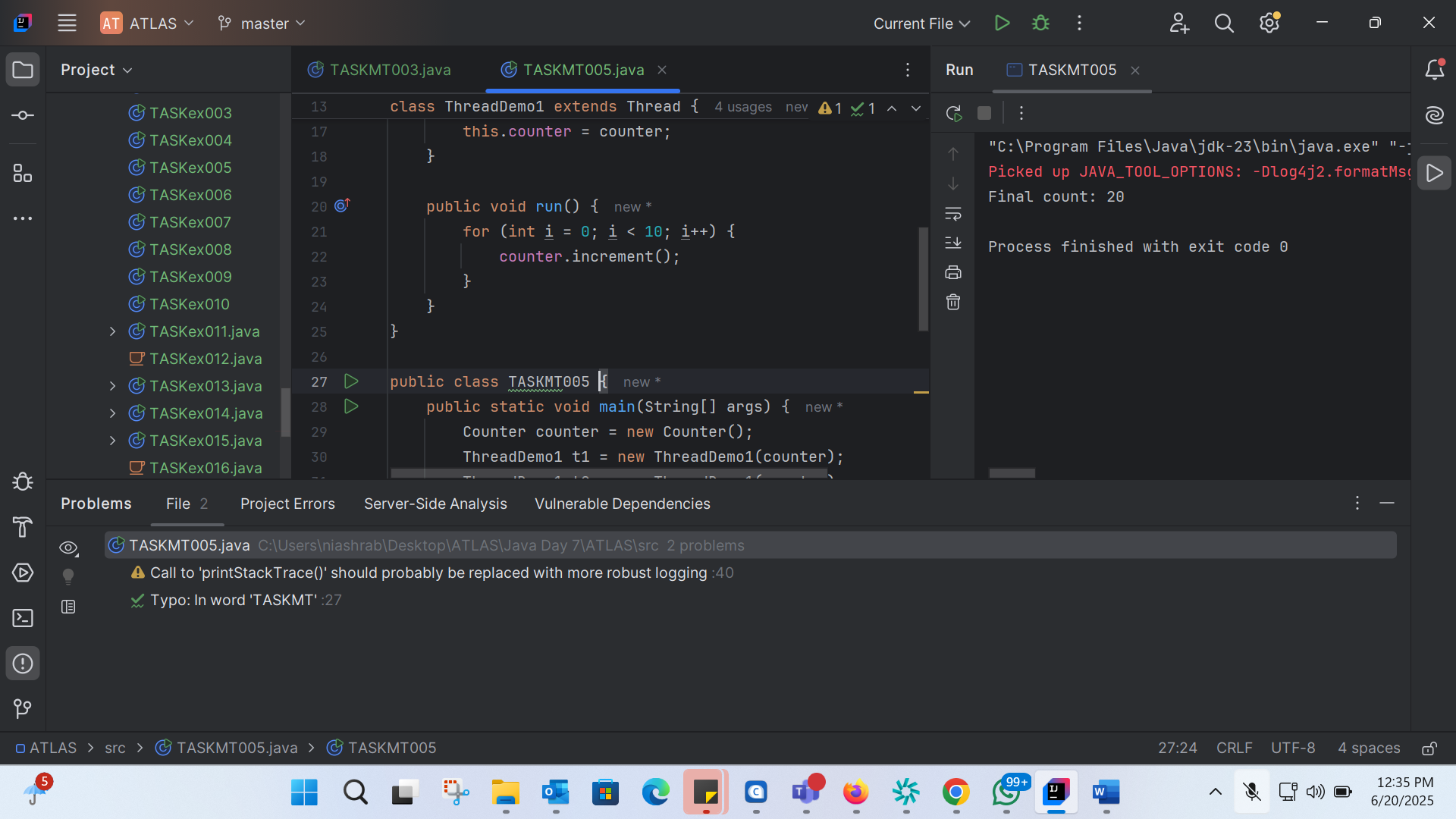
1. Creates a new thread and starts it running:



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 TASKMT003 {  
 public static void main(String args[]) {  
 RunnableDemo R1 = new RunnableDemo( "Thread-1");  
 R1.start();  
 RunnableDemo R2 = new RunnableDemo( "Thread-2");  
 R2.start();  
 }  
}

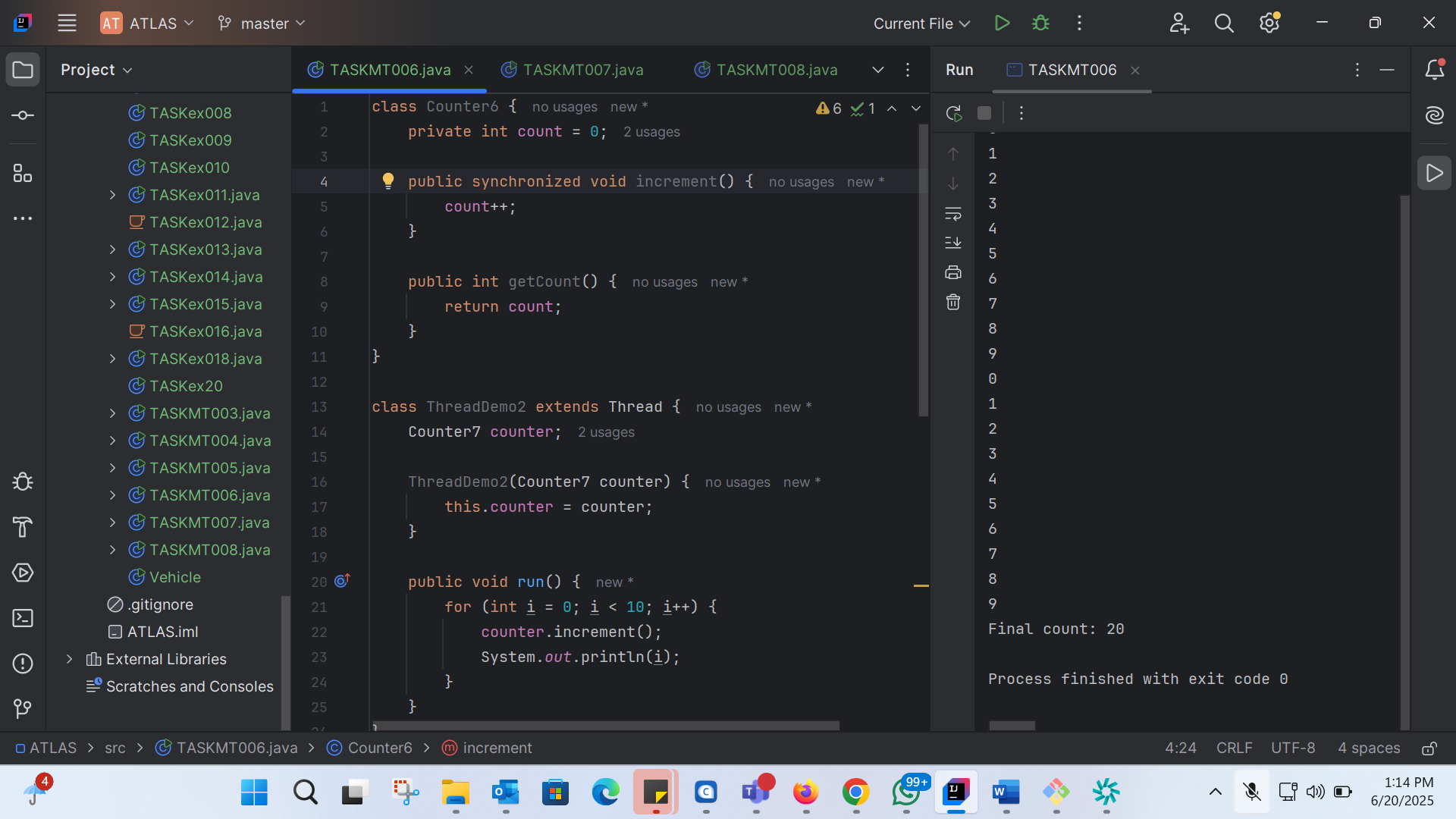
1. Extending thread class:
2. class RunnableDemo implements Runnable {  
    private Thread t;  
    private String threadName;  
    RunnableDemo( String name){  
    threadName = name;  
    System.*out*.println("Creating " + threadName );  
    }  
    public void run() {  
    System.*out*.println("Running " + threadName );  
    try {  
    for(int i = 4; i > 0; i--) {  
    System.*out*.println("Thread: " + threadName + ", " + i);  
   // Let the thread sleep for a while.  
    Thread.*sleep*(5000);  
    }  
    } catch (InterruptedException e) {  
    System.*out*.println("Thread " + threadName + " interrupted.");  
    }  
    System.*out*.println("Thread " + threadName + " exiting.");  
     
    }  
    public void THstart()  
    {  
    System.*out*.println("Starting " + threadName );  
    if (t == null)  
    {  
    t = new Thread (this, threadName);  
    t.start();  
    }  
    }  
   }  
   public class TASKMT003 {  
    public static void main(String args[]) {  
    RunnableDemo R1 = new RunnableDemo( "Thread-1");  
    R1.THstart();  
    RunnableDemo R2 = new RunnableDemo( "Thread-2");  
    R2.THstart();  
    }  
   }

TASK 5:

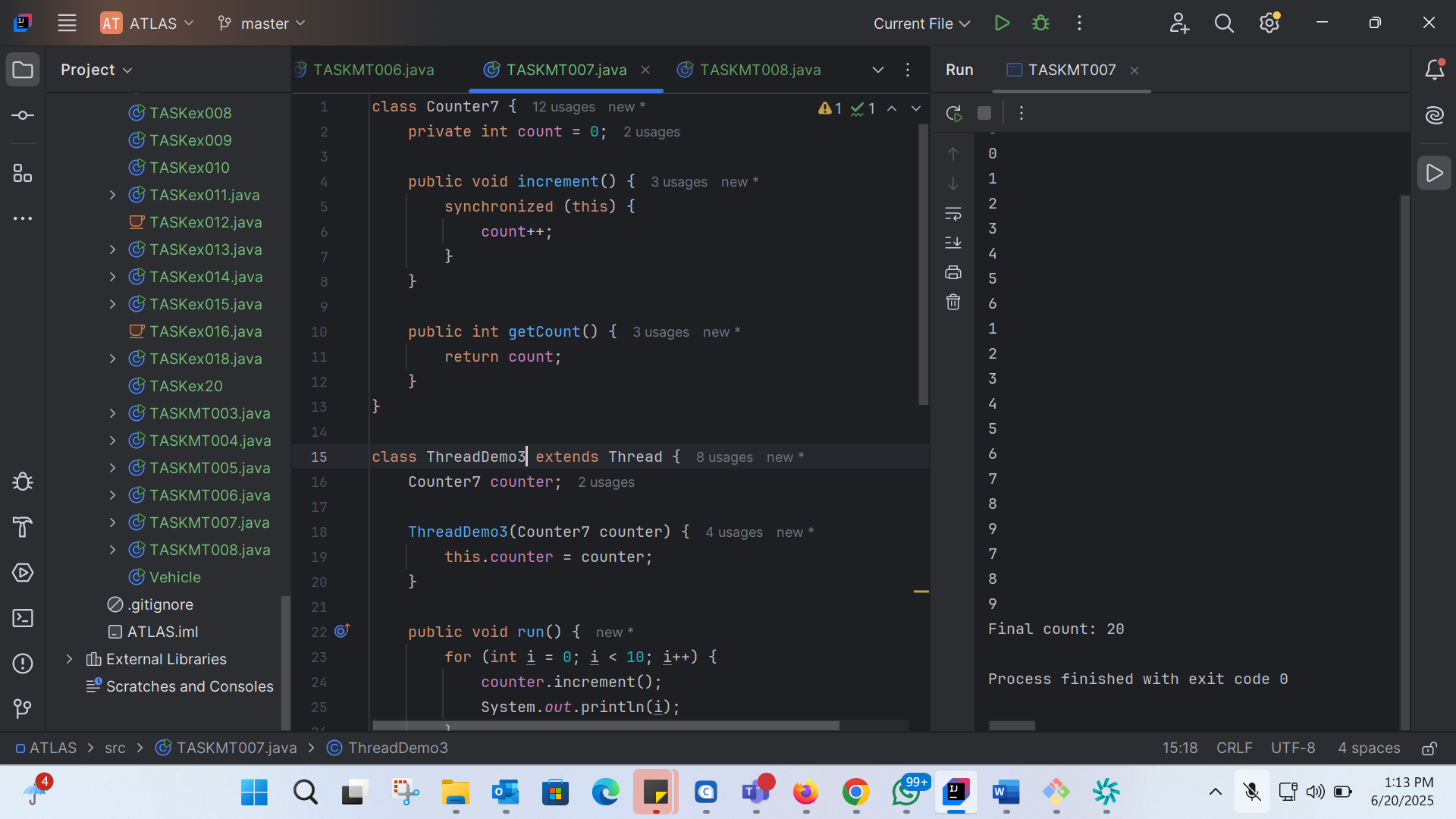


class Counter {  
 private int count = 0;  
  
 public void increment() {  
 count++;  
 }  
  
 public int getCount() {  
 return count;  
 }  
}  
  
class ThreadDemo1 extends Thread {  
 Counter counter;  
  
 ThreadDemo1(Counter counter) {  
 this.counter = counter;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 counter.increment();  
 }  
 }  
}  
  
public class TASKMT005 {  
 public static void main(String[] args) {  
 Counter counter = new Counter();  
 ThreadDemo1 t1 = new ThreadDemo1(counter);  
 ThreadDemo1 t2 = new ThreadDemo1(counter);  
  
 t1.start();  
 t2.start();  
  
 try {  
 t1.join();  
 t2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Final count: " + counter.getCount());  
 }  
}

TASK 11:

TASK6:

class Counter6 {  
 private int count = 0;  
  
 public synchronized void increment() {  
 count++;  
 }  
  
 public int getCount() {  
 return count;  
 }  
}  
  
class ThreadDemo2 extends Thread {  
 Counter7 counter;  
  
 ThreadDemo2(Counter7 counter) {  
 this.counter = counter;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 counter.increment();  
 System.*out*.println(i);  
 }  
 }  
}  
  
public class TASKMT006 {  
 public static void main(String[] args) {  
 Counter7 counter = new Counter7();  
 ThreadDemo3 t1 = new ThreadDemo3(counter);  
 ThreadDemo3 t2 = new ThreadDemo3(counter);  
  
 t1.start();  
 t2.start();  
  
 try {  
 t1.join();  
 t2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Final count: " + counter.getCount());  
 }  
}

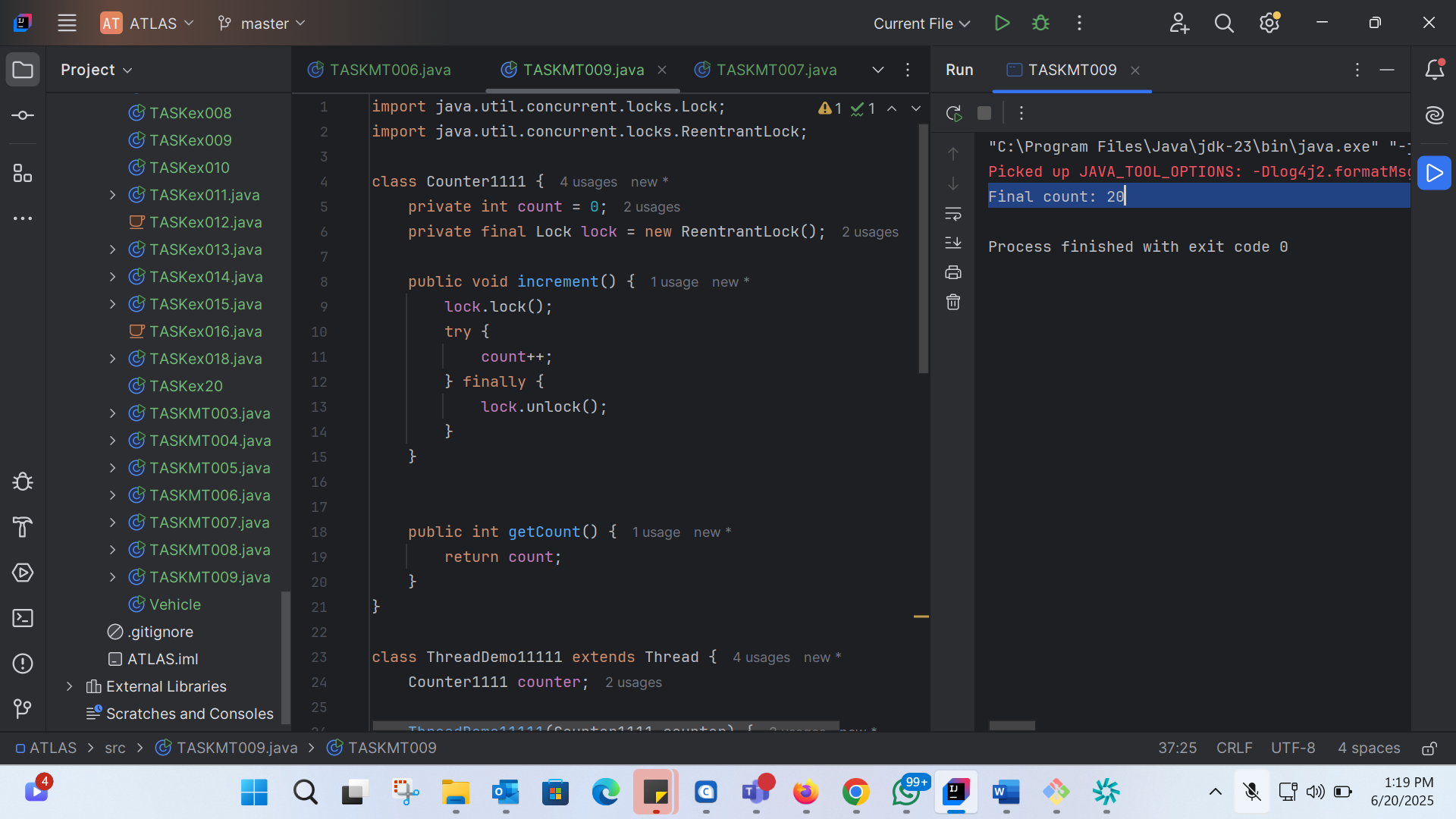
TASK7:

class Counter7 {  
 private int count = 0;  
  
 public void increment() {  
 synchronized (this) {  
 count++;  
 }  
 }  
  
 public int getCount() {  
 return count;  
 }  
}  
  
class ThreadDemo3 extends Thread {  
 Counter7 counter;  
  
 ThreadDemo3(Counter7 counter) {  
 this.counter = counter;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 counter.increment();  
 System.*out*.println(i);  
 }  
 }  
}  
  
public class TASKMT007 {  
 public static void main(String[] args) {  
 Counter7 counter = new Counter7();  
 ThreadDemo3 t1 = new ThreadDemo3(counter);  
 ThreadDemo3 t2 = new ThreadDemo3(counter);  
  
 t1.start();  
 t2.start();  
  
 try {  
 t1.join();  
 t2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Final count: " + counter.getCount());  
 }  
}

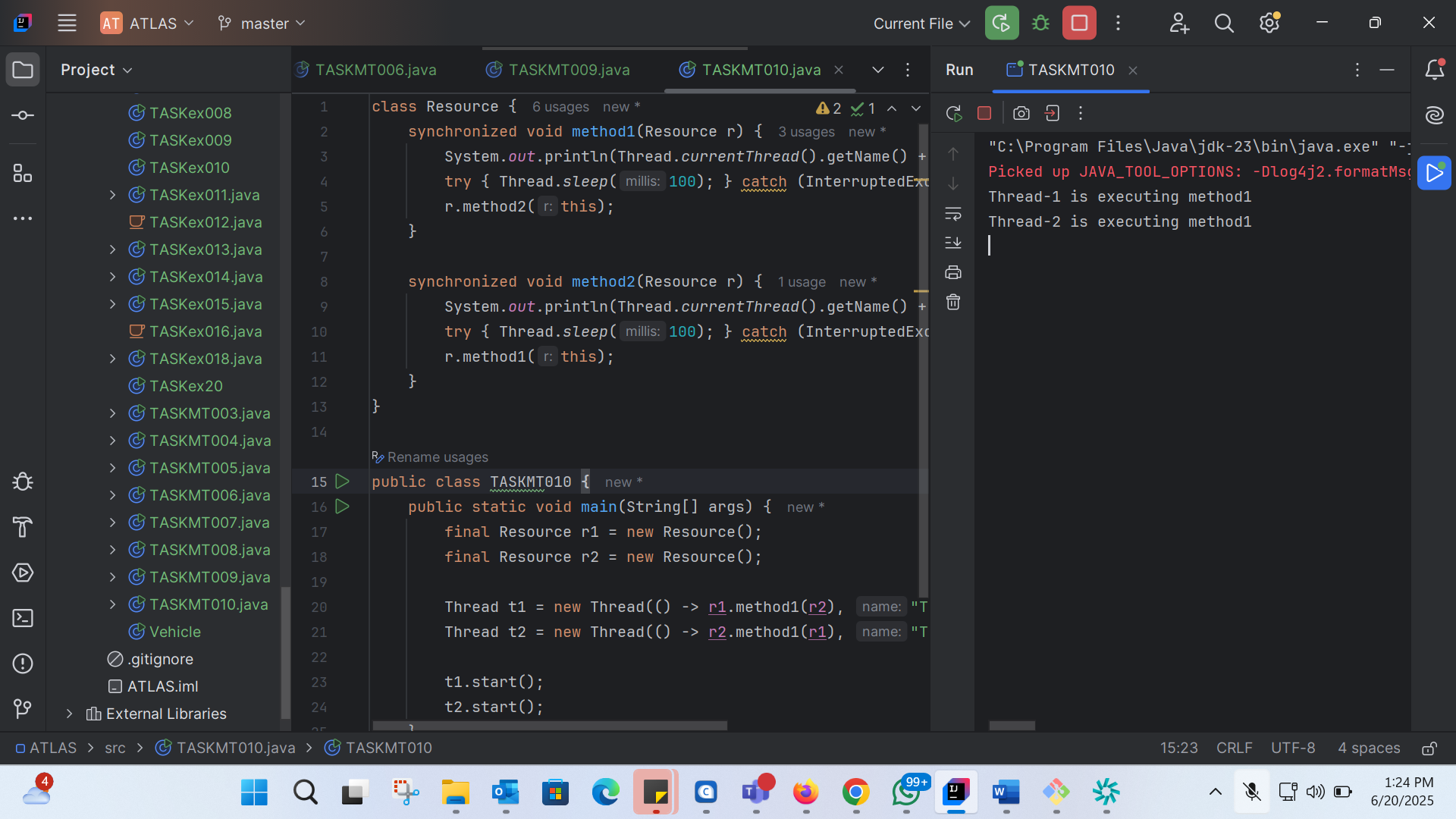
TASK 8:

class Counter6 {  
 private int count = 0;  
  
 public static synchronized void increment() {  
 count++;  
 }  
  
 public int getCount() {  
 return count;  
 }  
}  
  
class ThreadDemo4 extends Thread {  
 Counter6 counter;  
  
 ThreadDemo4(Counter6 counter) {  
 this.counter = counter;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 counter.*increment*();  
 System.*out*.println(i);  
 }  
 }  
}  
  
public class TASKMT006 {  
 public static void main(String[] args) {  
 Counter6 counter = new Counter6();  
 ThreadDemo4 t1 = new ThreadDemo4(counter);  
 ThreadDemo4 t2 = new ThreadDemo4(counter);  
  
 t1.start();  
 t2.start();  
  
 try {  
 t1.join();  
 t2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Final count: " + counter.getCount());  
 }  
}

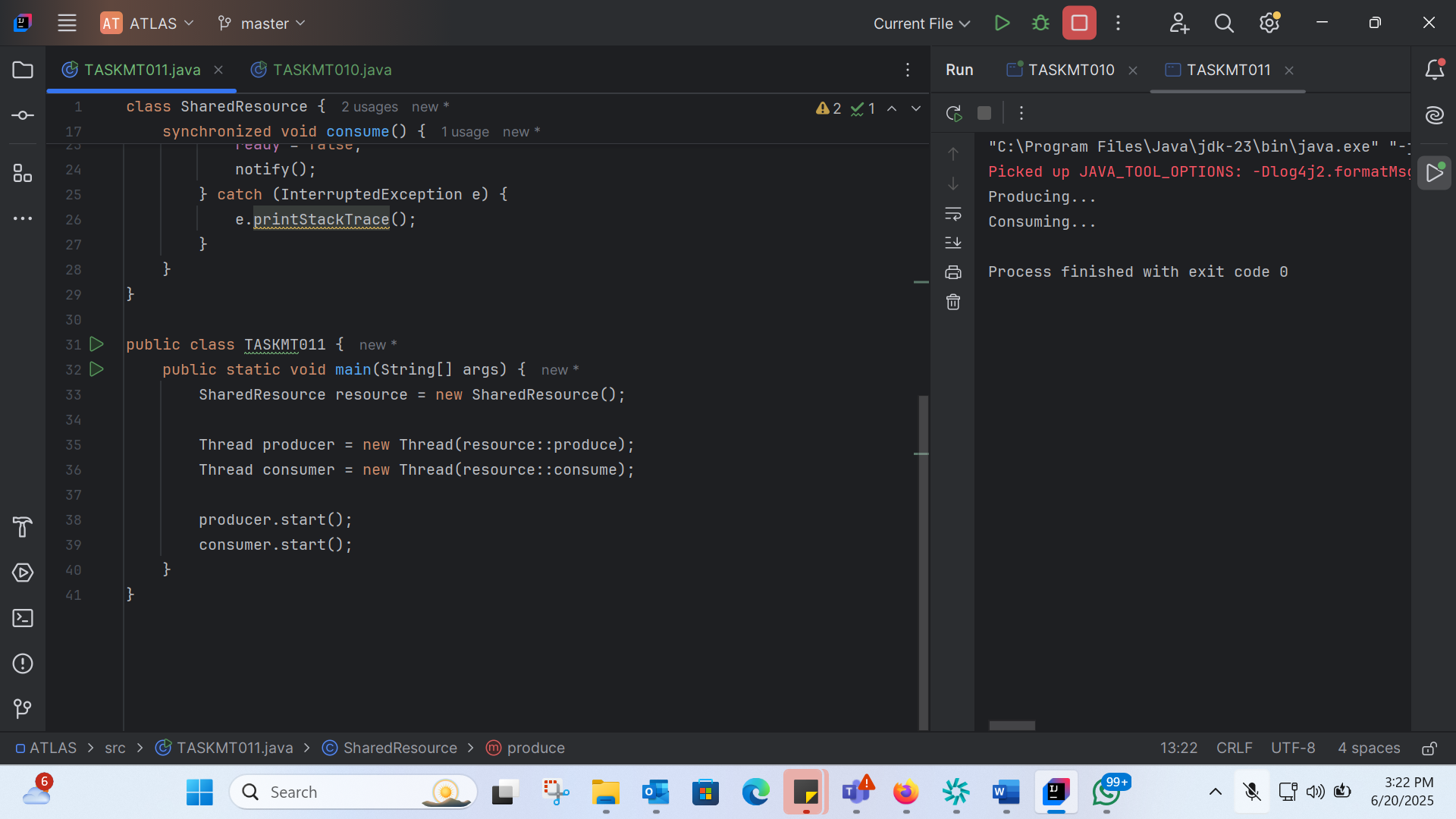
TASK 09:



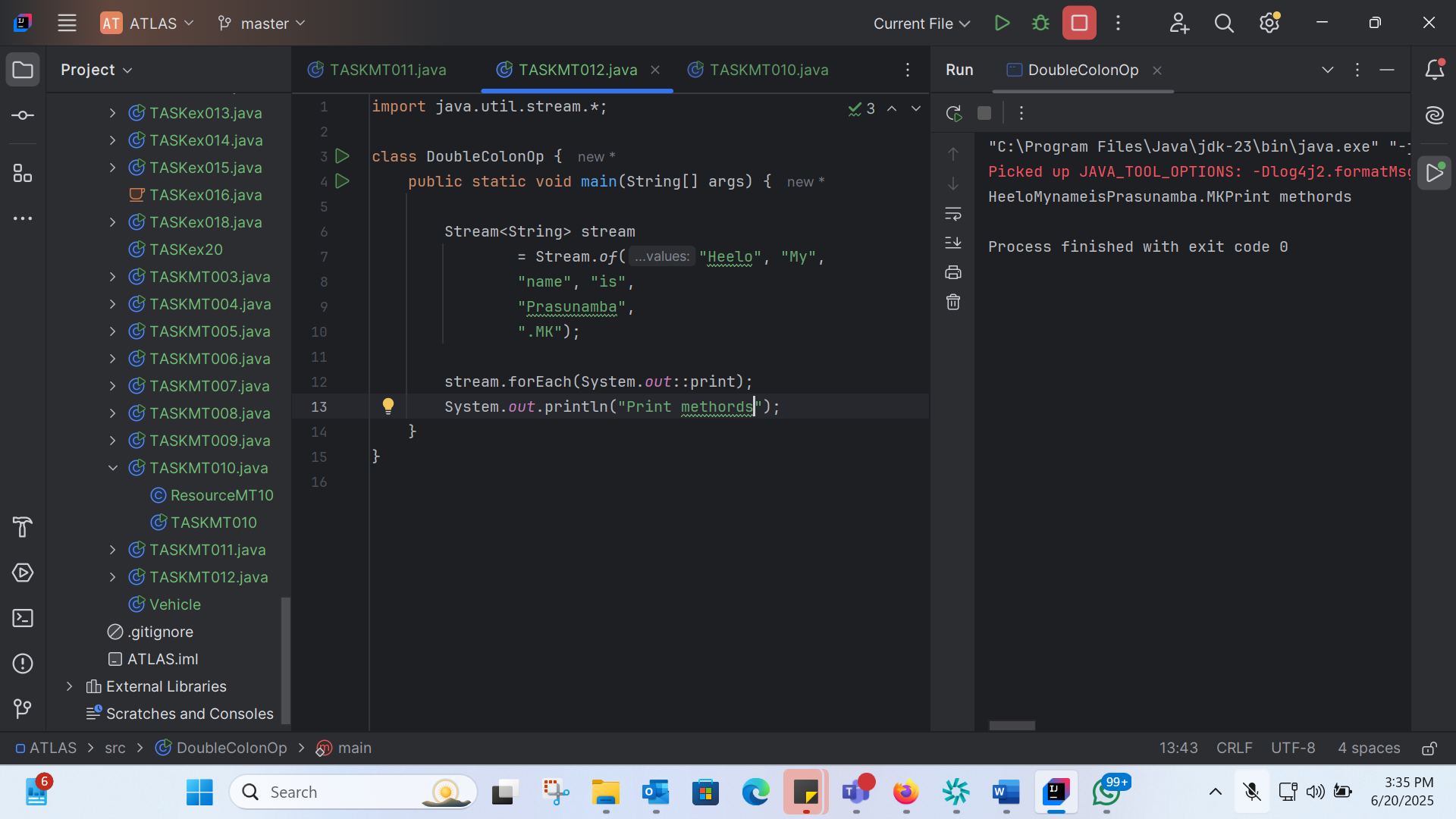
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
class Counter1111 {  
 private int count = 0;  
 private final Lock lock = new ReentrantLock();  
  
 public void increment() {  
 lock.lock();  
 try {  
 count++;  
 } finally {  
 lock.unlock();  
 }  
 }  
  
  
 public int getCount() {  
 return count;  
 }  
}  
  
class ThreadDemo11111 extends Thread {  
 Counter1111 counter;  
  
 ThreadDemo11111(Counter1111 counter) {  
 this.counter = counter;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 counter.increment();  
 }  
 }  
}  
  
public class TASKMT009 {  
 public static void main(String[] args) {  
 Counter1111 counter = new Counter1111();  
 ThreadDemo11111 t1 = new ThreadDemo11111(counter);  
 ThreadDemo11111 t2 = new ThreadDemo11111(counter);  
  
 t1.start();  
 t2.start();  
  
 try {  
 t1.join();  
 t2.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.*out*.println("Final count: " + counter.getCount());  
 }  
}

TASK 10: 

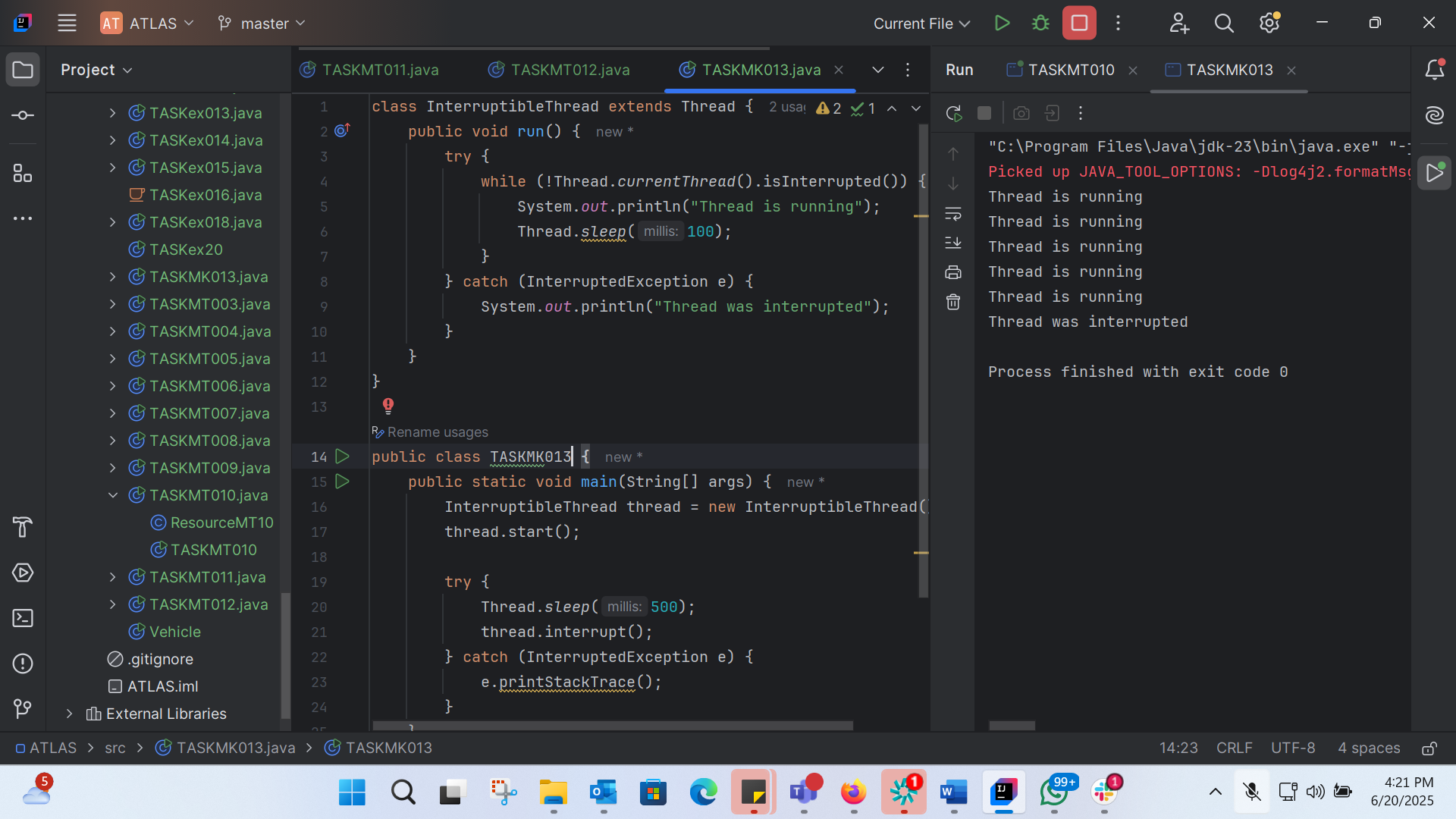
class Resource {  
 synchronized void method1(Resource r) {  
 System.*out*.println(Thread.*currentThread*().getName() + " is executing method1");  
 try { Thread.*sleep*(100); } catch (InterruptedException e) {}  
 r.method2(this);  
 }  
  
 synchronized void method2(Resource r) {  
 System.*out*.println(Thread.*currentThread*().getName() + " is executing method2");  
 try { Thread.*sleep*(100); } catch (InterruptedException e) {}  
 r.method1(this);  
 }  
}  
  
public class TASKMT010 {  
 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();  
 }  
}

TASK 11: 

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 TASKMT011 {  
 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();  
 }  
}

TASK 12: 

import java.util.stream.\*;  
  
class DoubleColonOp {  
 public static void main(String[] args) {  
  
 Stream<String> stream  
 = Stream.*of*("Heelo", "My",  
 "name", "is",  
 "Prasunamba",  
 ".MK");  
  
 stream.forEach(System.*out*::print);  
 System.*out*.println("Print methords");  
 }  
}

TASK 13: 

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 TASKMK013 {  
 public static void main(String[] args) {  
 InterruptibleThread thread = new InterruptibleThread();  
 thread.start();  
  
 try {  
 Thread.*sleep*(500);  
 thread.interrupt();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

TASK 14:

What are Daemon threads? Explain…

a daemon thread is a type of thread that runs in the background and provides some kind of service or support to other threads in the application. Daemon threads are typically used for tasks that are not critical to the main application logic, such as:

- Garbage collection

- Monitoring or logging

- Background tasks

Key characteristics:

- Low priority: Daemon threads have a lower priority than non-daemon threads.

- Background execution: Daemon threads run in the background and do not prevent the JVM from exiting when the main application threads finish.

- No guarantee of completion: Daemon threads may not complete their task if the JVM exits before they finish.

When to use daemon threads:

- Background tasks: Use daemon threads for tasks that can be safely interrupted or abandoned when the main application exits.

- Support tasks: Use daemon threads for tasks that provide support or services to other threads, such as monitoring or logging.

TASK 15:

What are the debugging tools in Java.. list down a few..

1. Eclipse Debugger: Eclipse provides a built-in debugger that allows you to set breakpoints, inspect variables, and step through code.

2. IntelliJ IDEA Debugger: IntelliJ IDEA also has a built-in debugger with features like breakpoints, variable inspection, and code stepping.

3. Java Mission Control (JMC): JMC is a profiling and monitoring tool that provides detailed information about Java applications, including CPU and memory usage.

4. VisualVM: VisualVM is a visual tool that provides detailed information about Java applications, including CPU and memory profiling, heap dump analysis, and thread analysis.

5. jconsole: jconsole is a command-line tool that allows you to monitor and manage Java applications, including viewing thread and memory information.

6. jdb: jdb is a command-line debugger that allows you to set breakpoints, inspect variables, and step through code.

7. Java Debugger (jdb) in JDK: The JDK includes a command-line debugger called jdb that can be used to debug Java applications.

8. YourKit Java Profiler: YourKit Java Profiler is a commercial profiling tool that provides detailed information about Java applications, including CPU and memory usage.

9. Java Flight Recorder (JFR): JFR is a profiling and monitoring tool that provides detailed information about Java applications, including CPU and memory usage.

TASK 16:

Try to understand the error Messages.. What are they and when to you?

Few are given below.

Error Messages in Java

Compile time and run time

Compile time :  grammatical mistakes … ;, {} , missing the code

Run time error or exceptions

Stack overflow error

Array index out of bounds

IO exception

Nulpointer exception

Compile-time errors:

- Grammatical mistakes: These errors occur when the Java code does not follow the syntax rules of the language. Examples include:

- Missing or mismatched brackets {}

- Missing or mismatched parentheses ()

- Missing semicolons ;

- Typographical errors in keywords or identifiers

Run-time errors (Exceptions):

- StackOverflowError: This error occurs when a method calls itself recursively without a proper termination condition, causing the stack to overflow.

- ArrayIndexOutOfBoundsException: This exception occurs when you try to access an array element with an index that is outside the bounds of the array.

- IOException: This exception occurs when there is an input/output error, such as reading or writing to a file that does not exist or is inaccessible.

- NullPointerException: This exception occurs when you try to access or manipulate a null (non-existent) object reference.

When do these errors occur?

- Compile-time errors: These errors occur when you compile your Java code, and the compiler detects syntax errors or other issues that prevent the code from being compiled.

- Run-time errors (Exceptions): These errors occur when you run your Java program, and the JVM encounters an issue that prevents it from executing the code correctly.

How to handle these errors?

- Compile-time errors: Fix the syntax errors or other issues reported by the compiler.

- Run-time errors (Exceptions): Use try-catch blocks to catch and handle exceptions, providing meaningful error messages or recovery actions.

By understanding the types of error messages and when they occur, you can write more robust Java code and handle errors effectively.

TASK 17:

What is Stack trace.. What will it do?

Understand the below points..

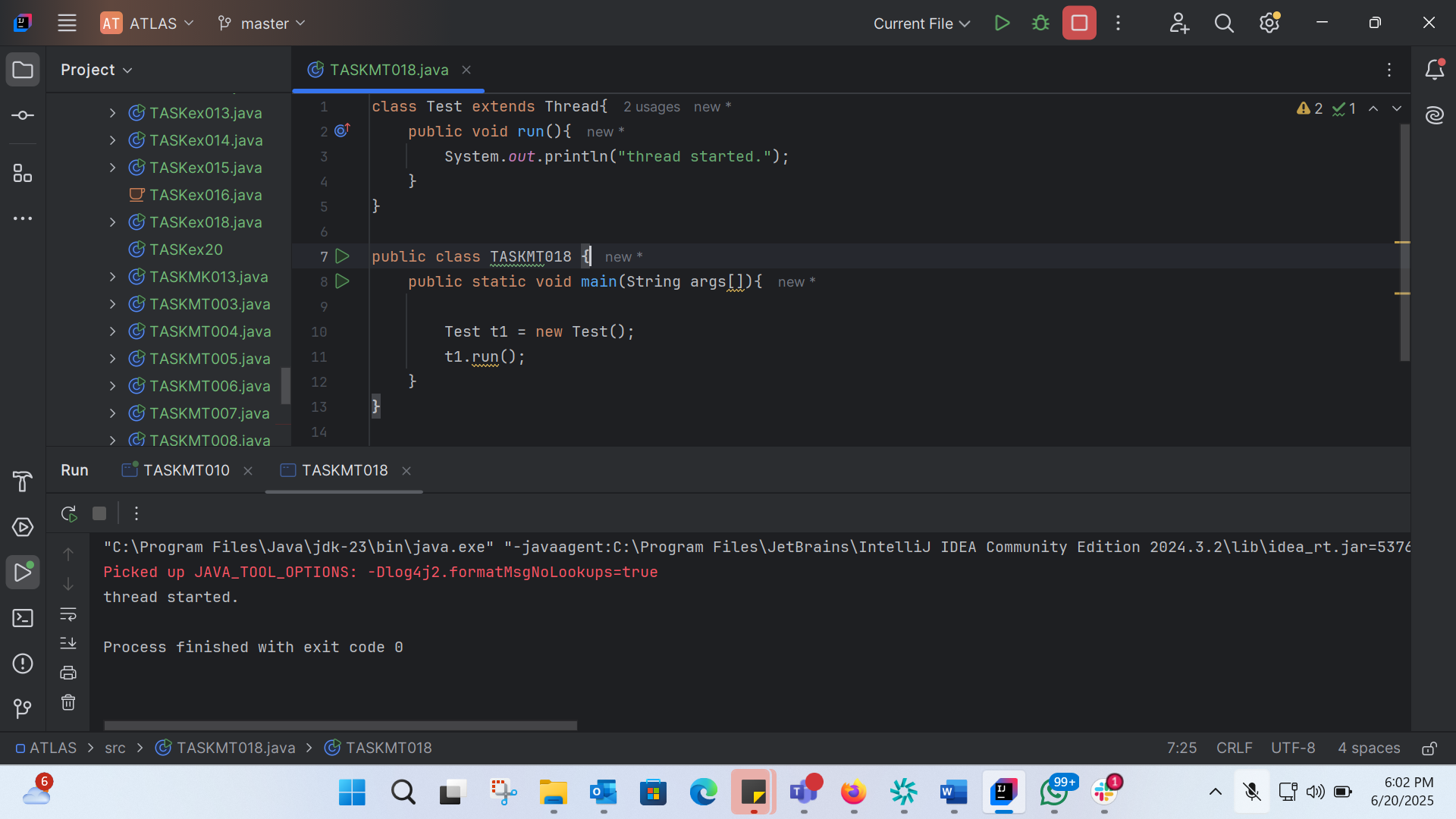
Identify the error

Locate the code

Analyze the code

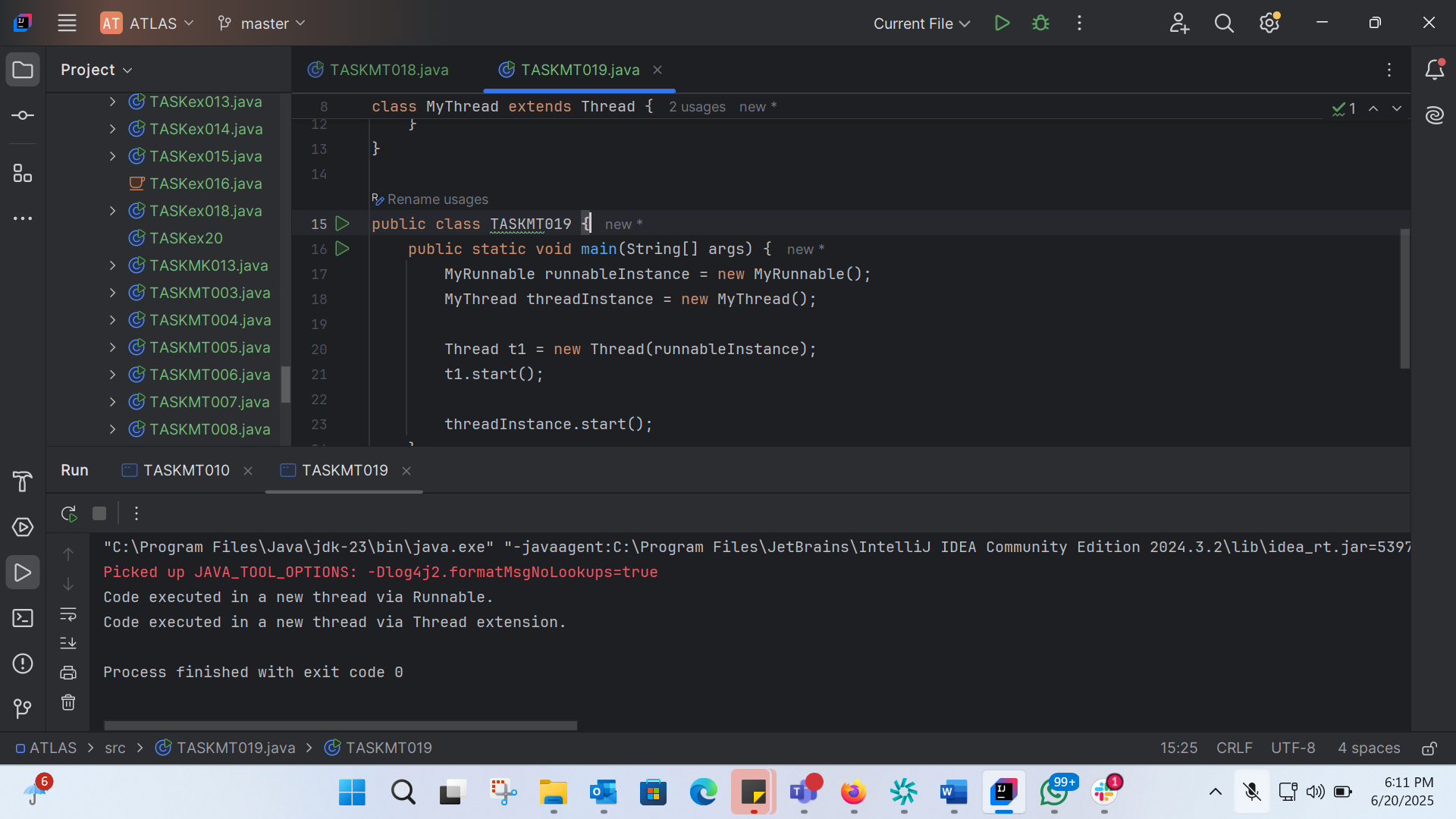
Solution also

TASK 18:



class Test extends Thread{  
 public void run(){  
 System.*out*.println("thread started.");  
 }  
}  
  
public class TASKMT018 {  
 public static void main(String args[]){  
  
 Test t1 = new Test();  
 t1.run();  
 }  
}

TASK 19:



class MyRunnable implements Runnable {  
 @Override  
 public void run() {  
 System.*out*.println("Code executed in a new thread via Runnable.");  
 }  
}  
  
class MyThread extends Thread {  
 @Override  
 public void run() {  
 System.*out*.println("Code executed in a new thread via Thread extension.");  
 }  
}  
  
public class TASKMT019 {  
 public static void main(String[] args) {  
 MyRunnable runnableInstance = new MyRunnable();  
 MyThread threadInstance = new MyThread();  
  
 Thread t1 = new Thread(runnableInstance);  
 t1.start();  
  
 threadInstance.start();  
 }  
}

TASK 20:



public class TASKMT020 {  
 public static void main(String[] args) {  
 *method1*();  
 }  
  
 public static void method1() {  
 *method2*();  
 }  
  
 public static void method2() {  
 *method3*();  
 }  
  
 public static void method3() {  
 StackTraceElement[] stackTrace = new Throwable().getStackTrace();  
 System.*out*.println("Thread Stack Trace:");  
 // Iterate through the StackTraceElement array and print details  
 for (StackTraceElement element : stackTrace) {  
 System.*out*.println(" Class: " + element.getClassName() + ", Method: " + element.getMethodName() + ", Line: " + element.getLineNumber());  
 }  
 }  
}