Task 01:

What is a Process?

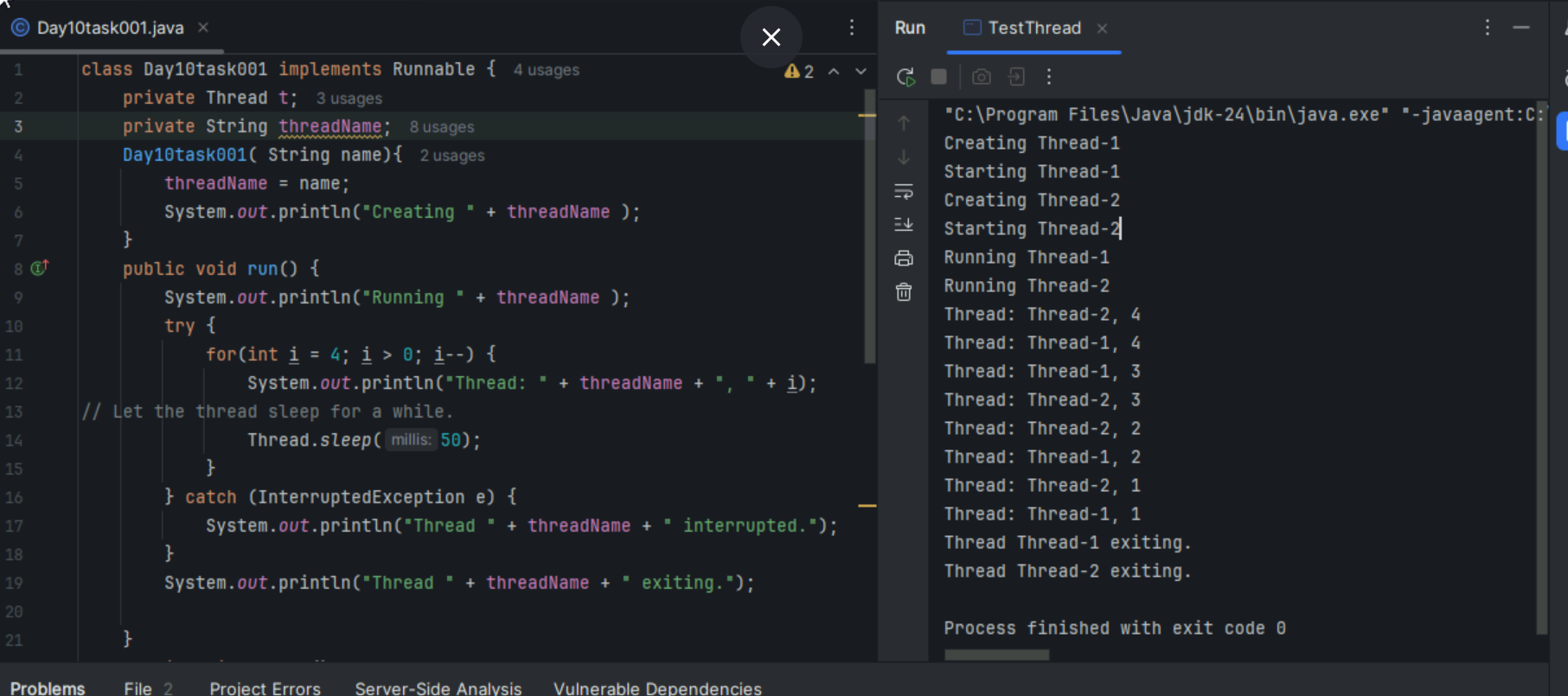
A process is like a program that is currently running in our system and when we open an application, it starts a process for it.

Task 02:

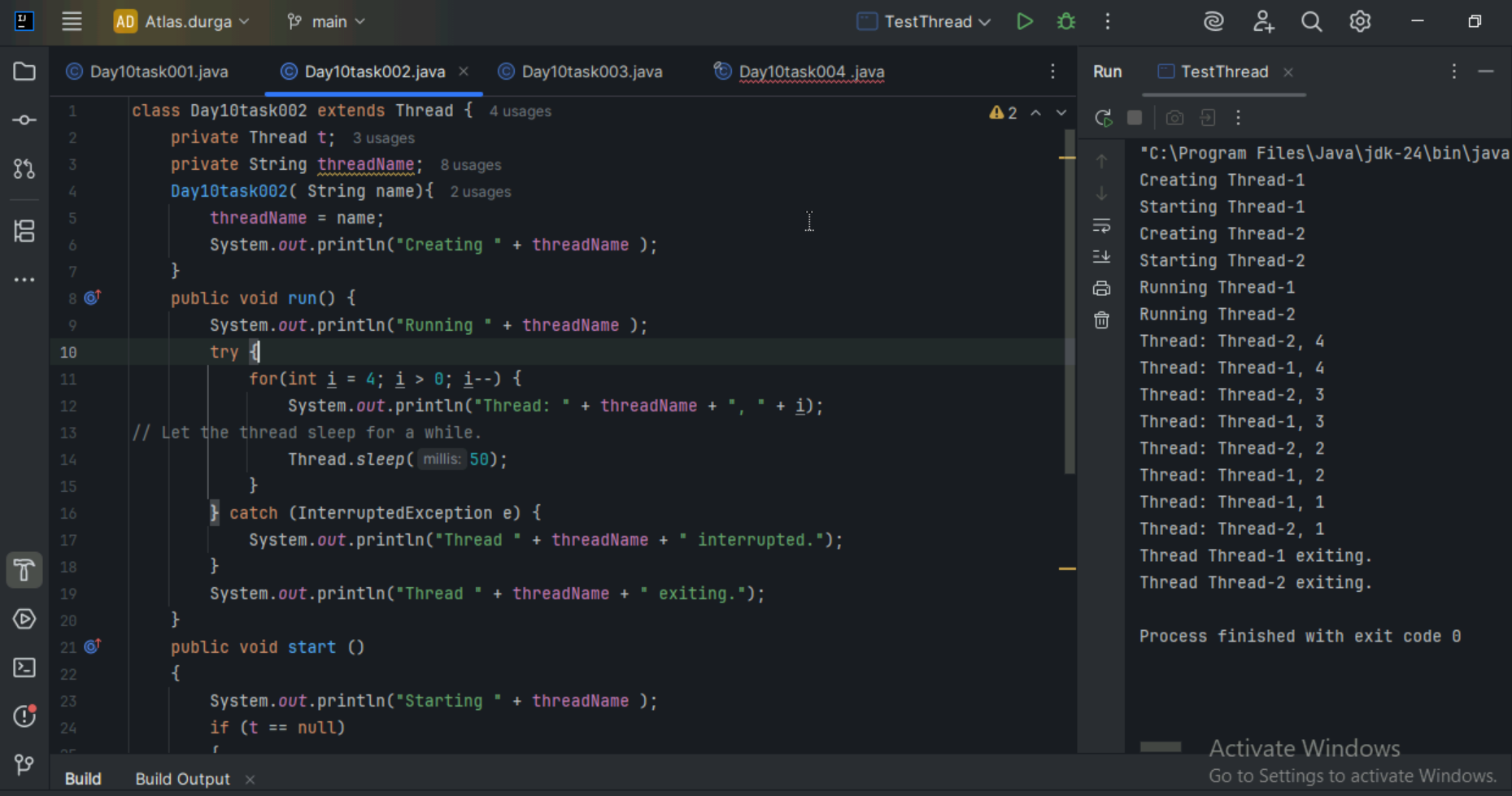
What is a Thread?

A thread is like a smaller unit within a process that helps a program do many tasks at the same time. For example, in a web browser (which is a process), one thread might load a webpage, another might play a video.

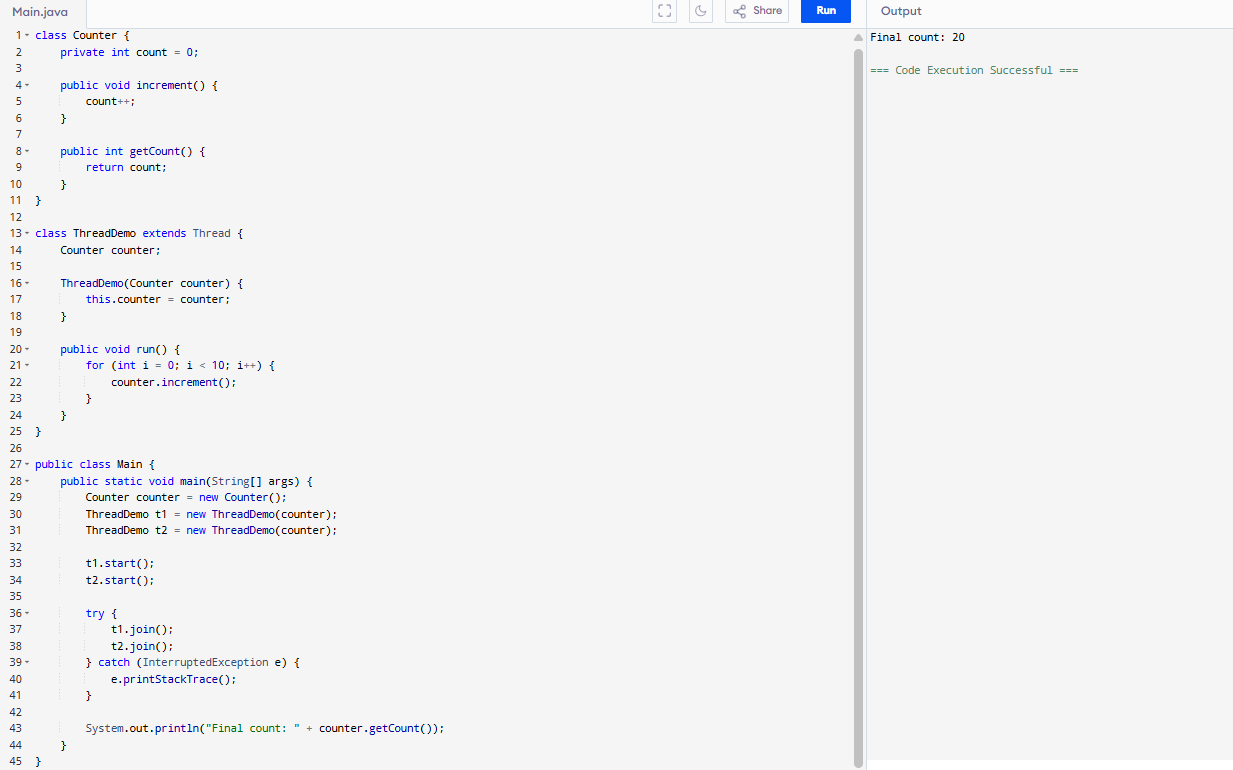
Task 003:



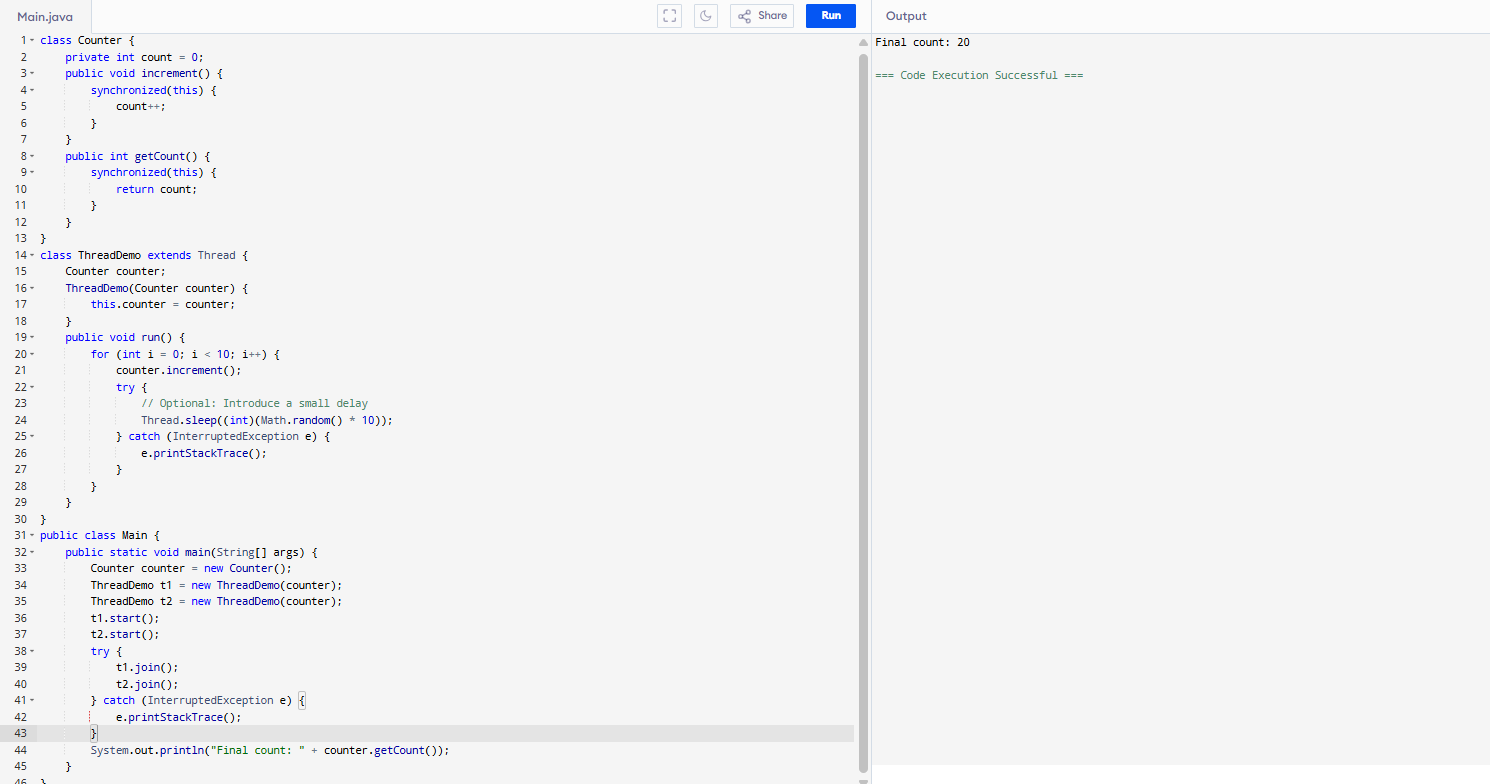
Task 004:



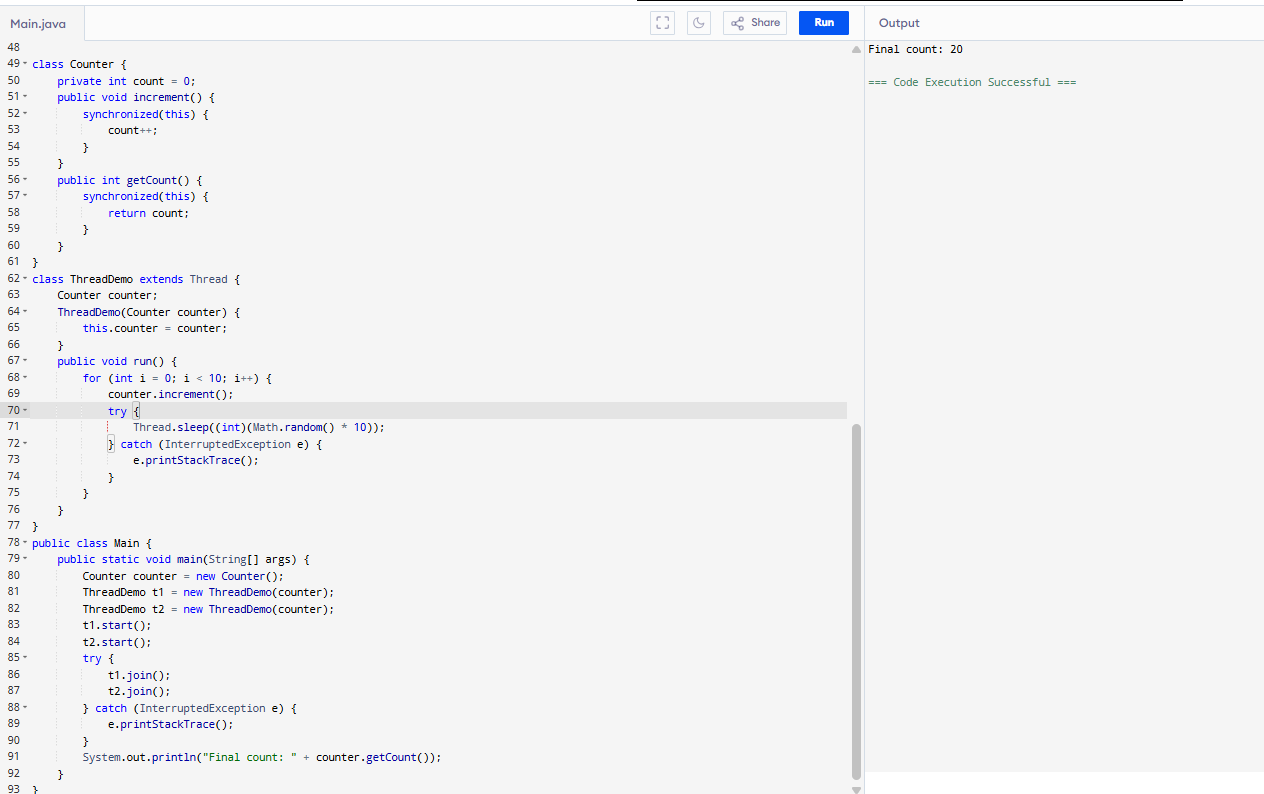
Task 005:



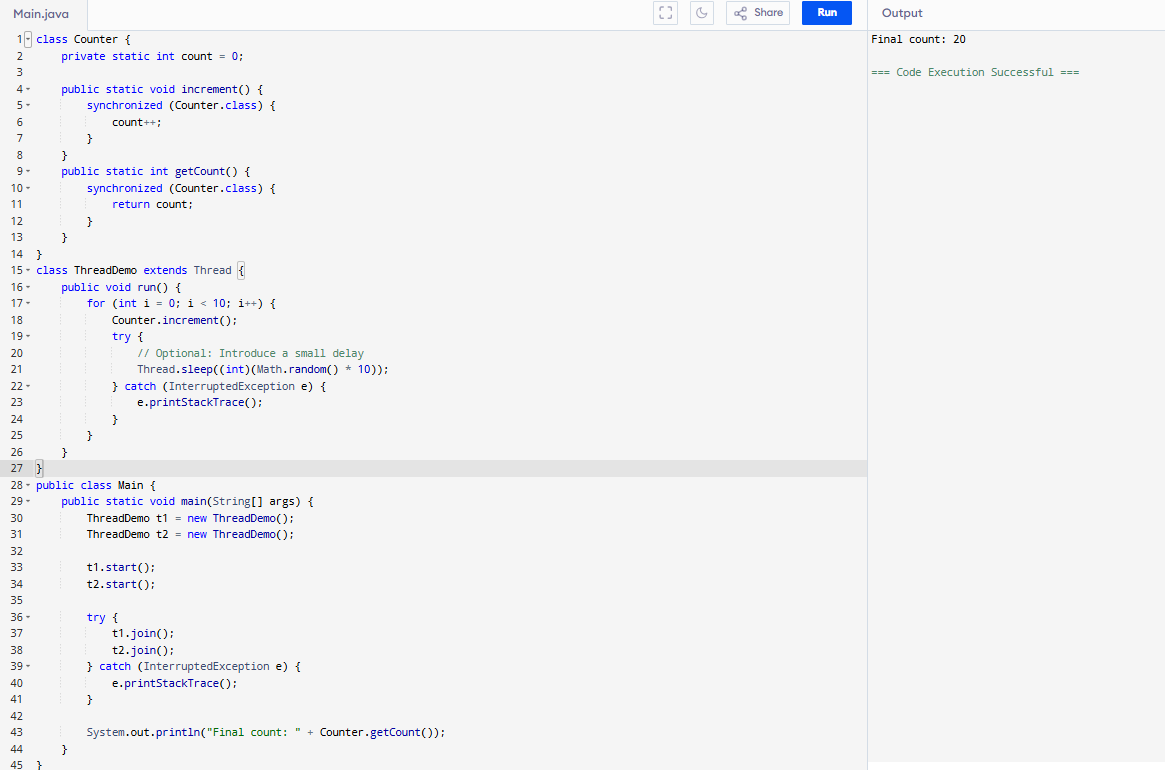
Task 006:



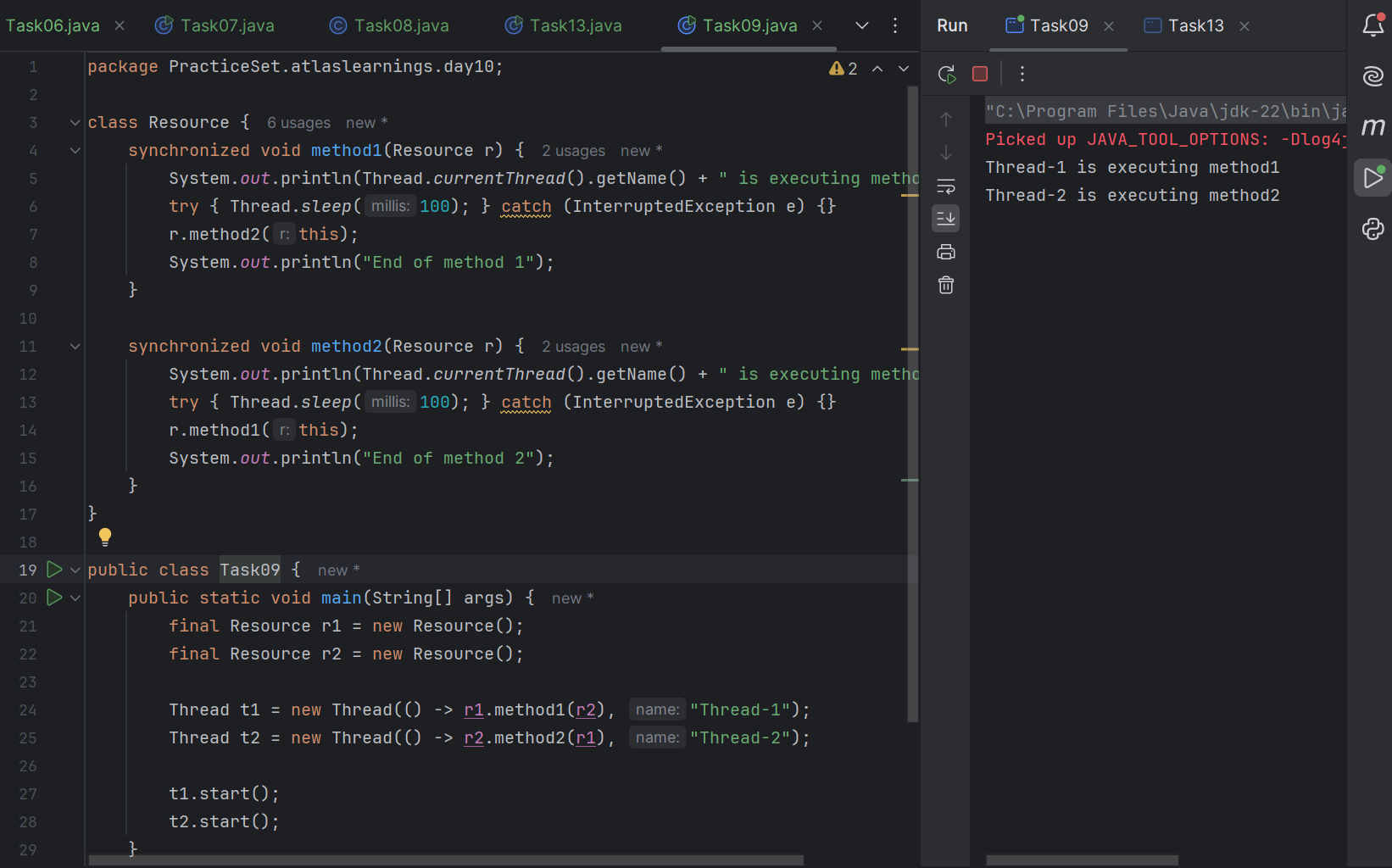
Task 007:



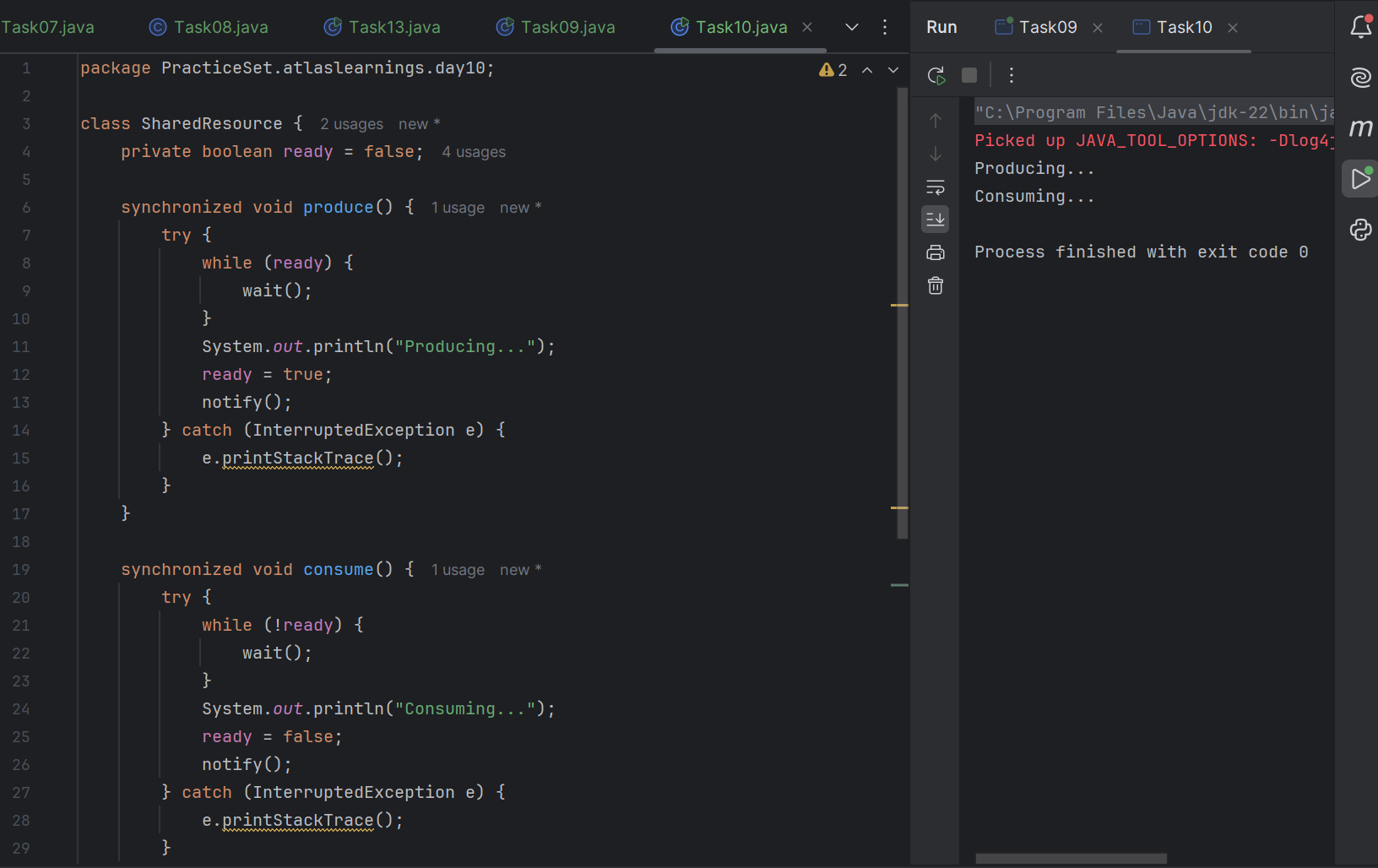
Task 008:



Task 009:



Task 10:



Task 11:

Inter- thread communication…

class SharedResource {

// Defines a class named SharedResource which will handle the shared resource logic between threads.

private boolean ready = false;

// Declares a private boolean ready initialized to false. This flag tracks whether the resource is ready for consumption.

synchronized void produce()

// Defines a synchronized method produce(), ensuring that only one thread can execute it at a time.

{

try

// Starts a try block to handle potential InterruptedException during execution.

{

while (ready) {

wait();

}

// Checks if ready is true. If so, the thread waits, releasing the lock on the object until notified.

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

// Produces the resource by printing "Producing...".

ready = true;

// Sets ready to true, indicating the resource is now ready for consumption.

notify();

// Notifies a waiting thread, waking it up to continue execution.

} catch (InterruptedException e) {

e.printStackTrace();

}

// Catches and prints any InterruptedException if the thread is interrupted during wait()

}

synchronized void consume() {

// Starts a try block to handle potential InterruptedException.

try {

while (!ready) {

wait();

}

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

// \*\*Checks if `ready` is false\*\*. If so, the thread waits, releasing the lock until notified.

ready = false;

notify();

// \*\*Sets `ready` to false\*\*, indicating the resource has been consumed and is no longer ready.

} catch (InterruptedException e)

// \*\*Catches and prints\*\* any `InterruptedException` during `wait()`.

// ### Main Method

{

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);

// \*\*Creates two threads\*\*: one for the `produce` method and one for `consume`.

producer.start();

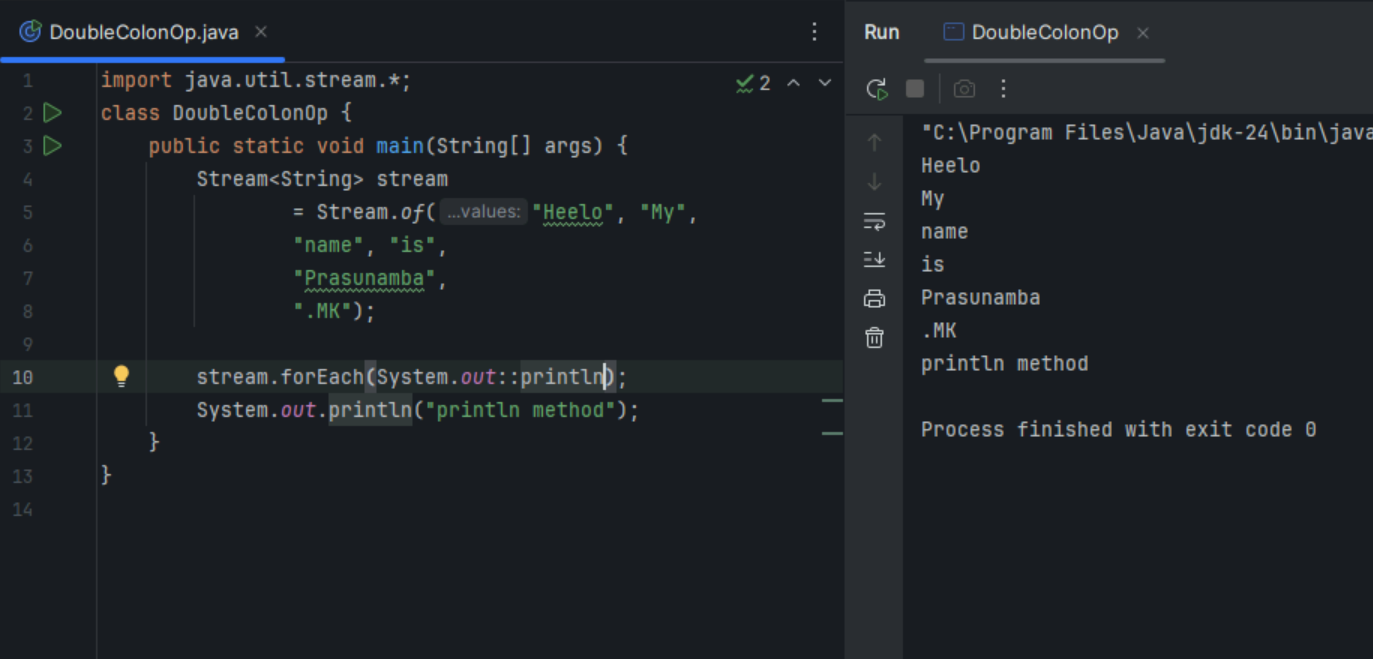
consumer.start();

}

}

This is what I understand the Inter- thread communication…

Task 12:



Task 13:

Interrupting a thread

**Example of Interrupting a Thread**

class InterruptibleThread extends Thread

// Defines a class InterruptibleThread that extends the Thread class, allowing it to be executed in a separate thread.

{

public void run()

// Overrides the run method from the Thread class. This is where the thread's work happens.

{

try

// Starts a try block to handle any InterruptedException that may occur.

{

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

// Loop condition: Continuously checks if the current thread has been interrupted.

{

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

// Prints a message indicating that the thread is alive and functioning.

Thread.sleep(100);

// Puts the thread to sleep for 100 milliseconds, simulating some work being done.

}

} catch (InterruptedException e) {

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

}

}

}

// Catches InterruptedException: If the thread gets interrupted during sleep(), it exits the loop and prints "Thread was interrupted".

public class InterruptExample

// Defines the main class InterruptExample that contains the main method to test thread interruption.

{

public static void main(String[] args) {

InterruptibleThread thread = new InterruptibleThread();

// \*\*Creates an instance\*\* of `InterruptibleThread`

thread.start();

// Starts the thread, triggering its run method.

try {

Thread.sleep(500);

thread.interrupt();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

// -Pauses the main thread for 500 milliseconds to allow `InterruptibleThread` to run.

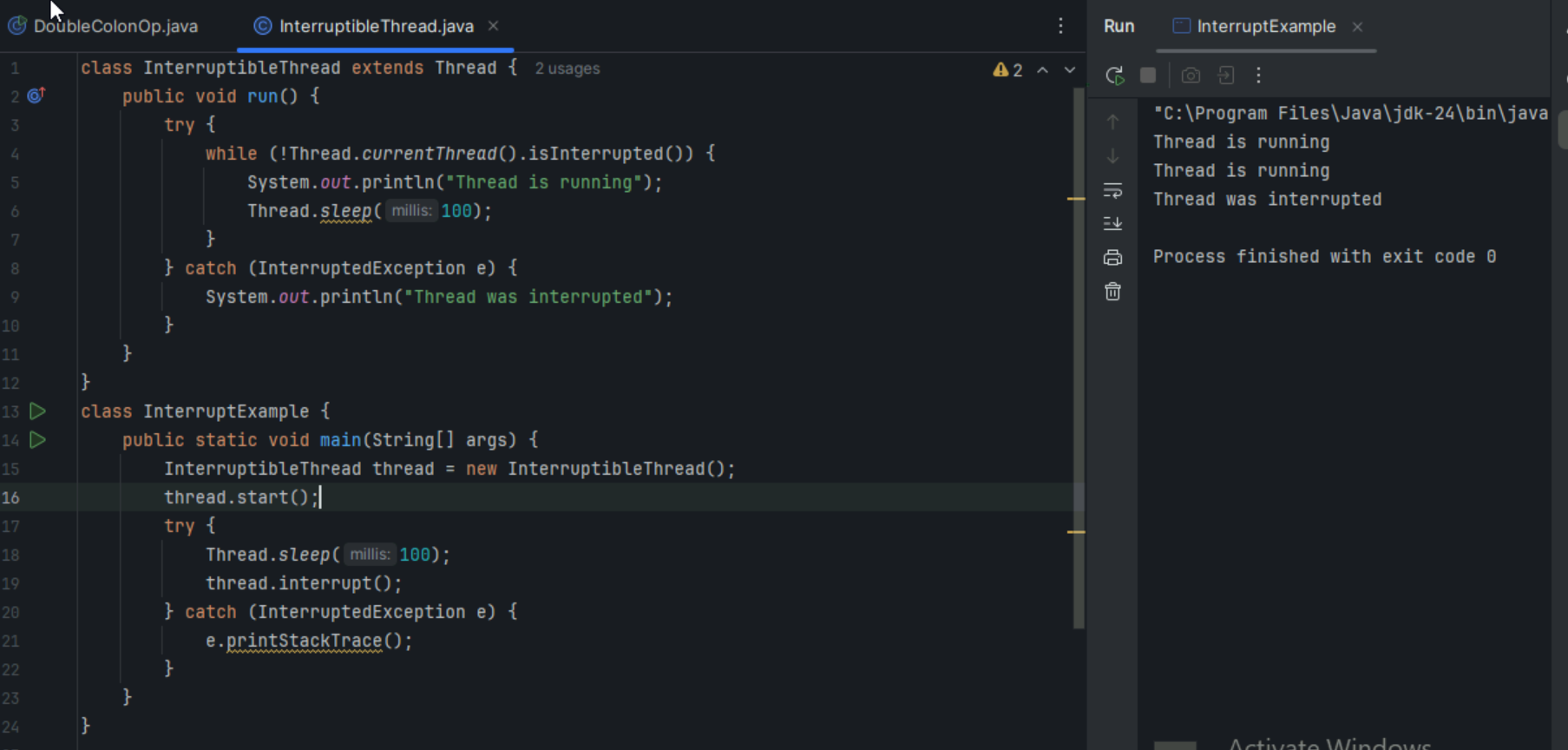
// - Interrupts the thread using `thread.interrupt()`, which causes the loop to break and the `run` method to end.

// - Catches and prints any `InterruptedException` that occurs in the main thread during `sleep()`.

// ### What It Does

// - Purpose: Demonstrates how to use thread interruption in Java. The thread runs in a loop, checking if it has been interrupted. If interrupted, the loop exits and the thread stops gracefully.

// - Usage: Useful when you need to terminate a running thread safely, ensuring resources are managed properly without abrupt stops.



Task 14:

What are Daemon threads? Explain…

Daemon Threads: These are background threads that run in the JVM. They provide services like garbage collection, and they don't prevent the JVM from exiting when the program finishes.

Background Tasks: They're ideal for tasks that should continue running in the background but don't need to block the program from shutting down.

Automatically Terminate: When all non-daemon threads (regular threads) finish, daemon threads automatically stop.

Create a Thread: Just like a regular thread.

Set as Daemon: Before starting the thread, call setDaemon(true)

Thread thread = new Thread(() -> {

// Task code

});

thread.setDaemon(true); // Marks it as a daemon thread

thread.start();

Non-blocking: Daemon threads don't block the JVM from exiting.

Use with Care: Since they can terminate unexpectedly when the main program ends, avoid using them for tasks that need to complete reliably.

Task 15

