Multithreading in Java

**Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

### Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

## Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

* Process-based Multitasking (Multiprocessing)
* Thread-based Multitasking (Multithreading)

### 1) Process-based Multitasking (Multiprocessing)

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading [registers](https://www.javatpoint.com/register-memory), memory maps, updating lists, etc.

### 2) Thread-based Multitasking (Multithreading)

* Threads share the same address space.
* A thread is lightweight.
* Cost of communication between the thread is low.

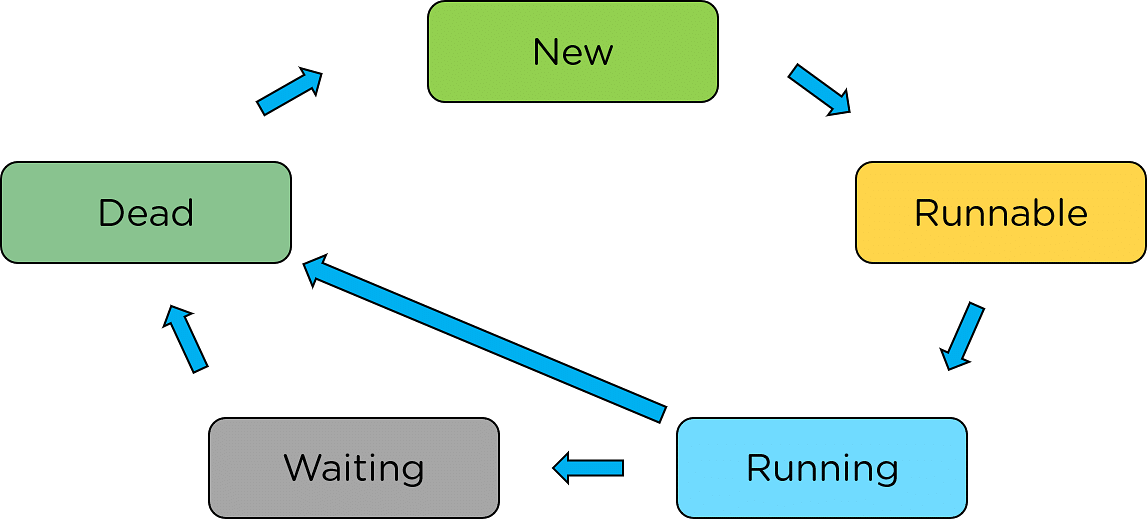
## Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides [constructors](https://www.javatpoint.com/java-constructor) and methods to create and perform operations on a thread. Thread class extends [Object class](https://www.javatpoint.com/object-class) and implements Runnable interface.

**Lifecycle of a Thread in Java**

The lifecycle of each thread in Java has five different stages. You will look into each one of those stages in detail. The Stages of the Lifecycle are mentioned below.

* New
* Runnable
* Running
* Waiting
* Dead



### New

The first stage is "New". This stage is where it initiates the thread. After that, every thread remains in the new state until the thread gets assigned to a new task.

### Runnable

The next stage is the runnable stage. Here, a thread gets assigned to the task and sets itself for running the task.

### Running

The third stage is the execution stage. Here, the thread gets triggered as control enters the thread, and the thread performs a task and continues the execution until it finishes the job.

### Waiting

At times, there is a possibility that one process as a whole might depend on another. During such an encounter, the thread might halt for an intermediate result because of its dependency on a different process. This stage is called the Waiting Stage.

### Dead

The final stage of the process execution with Multithreading in Java is thread termination. After it terminates the process, the[JVM](https://www.simplilearn.com/jvm-and-the-implications-of-sandbox-model-rar41-article) automatically declares the thread dead and terminates the thread. This stage is known as the dead thread stage.

Now that you learned about the thread and its life cycle, move ahead and understand multithreading in Java.

# How to create a thread in Java

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

### Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

### Commonly used Constructors of Thread class:

* Thread()
* Thread(String name)
* Thread(Runnable r)
* Thread(Runnable r,String name)

### Commonly used methods of Thread class:

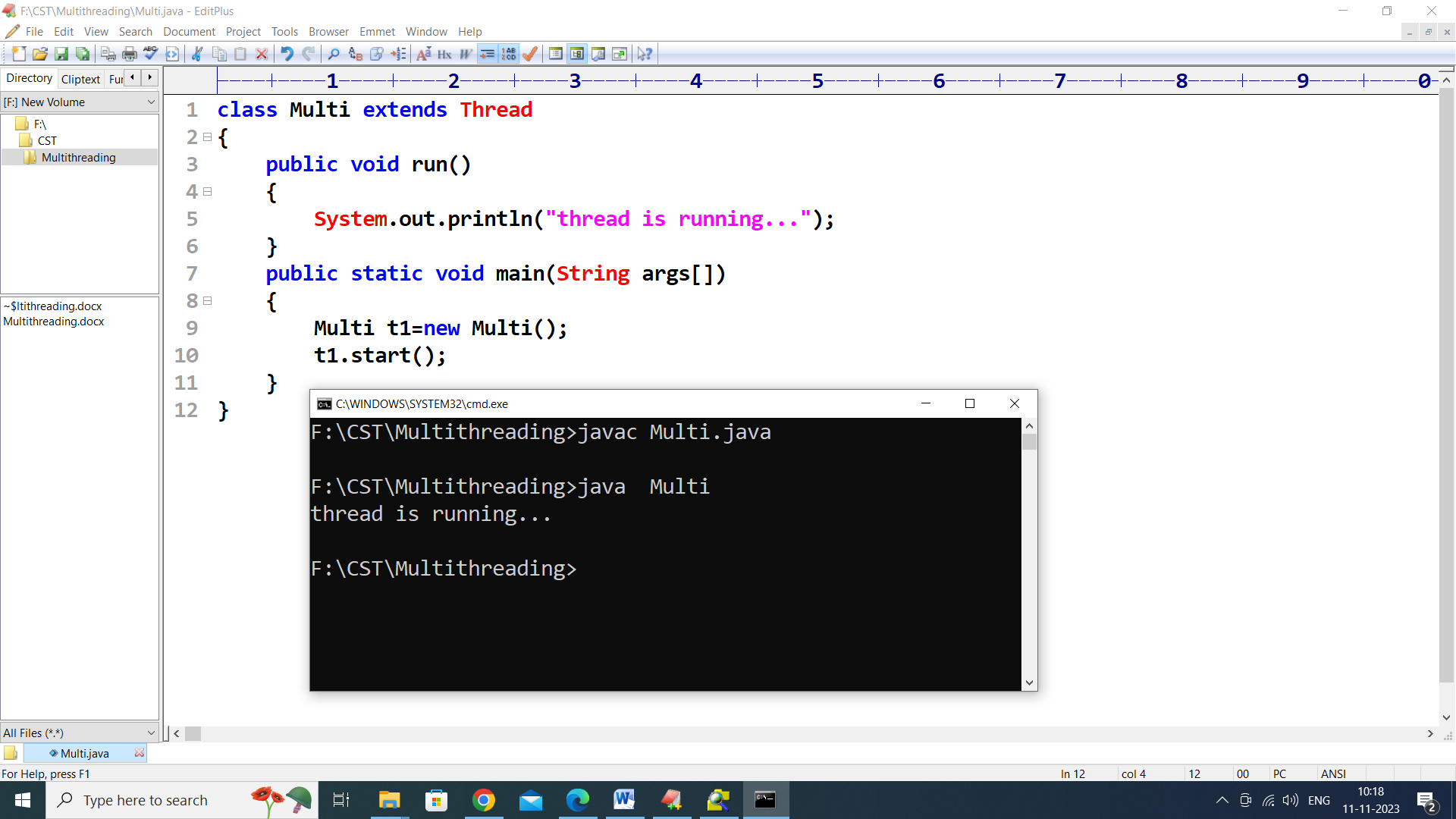
1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

### Runnable interface:

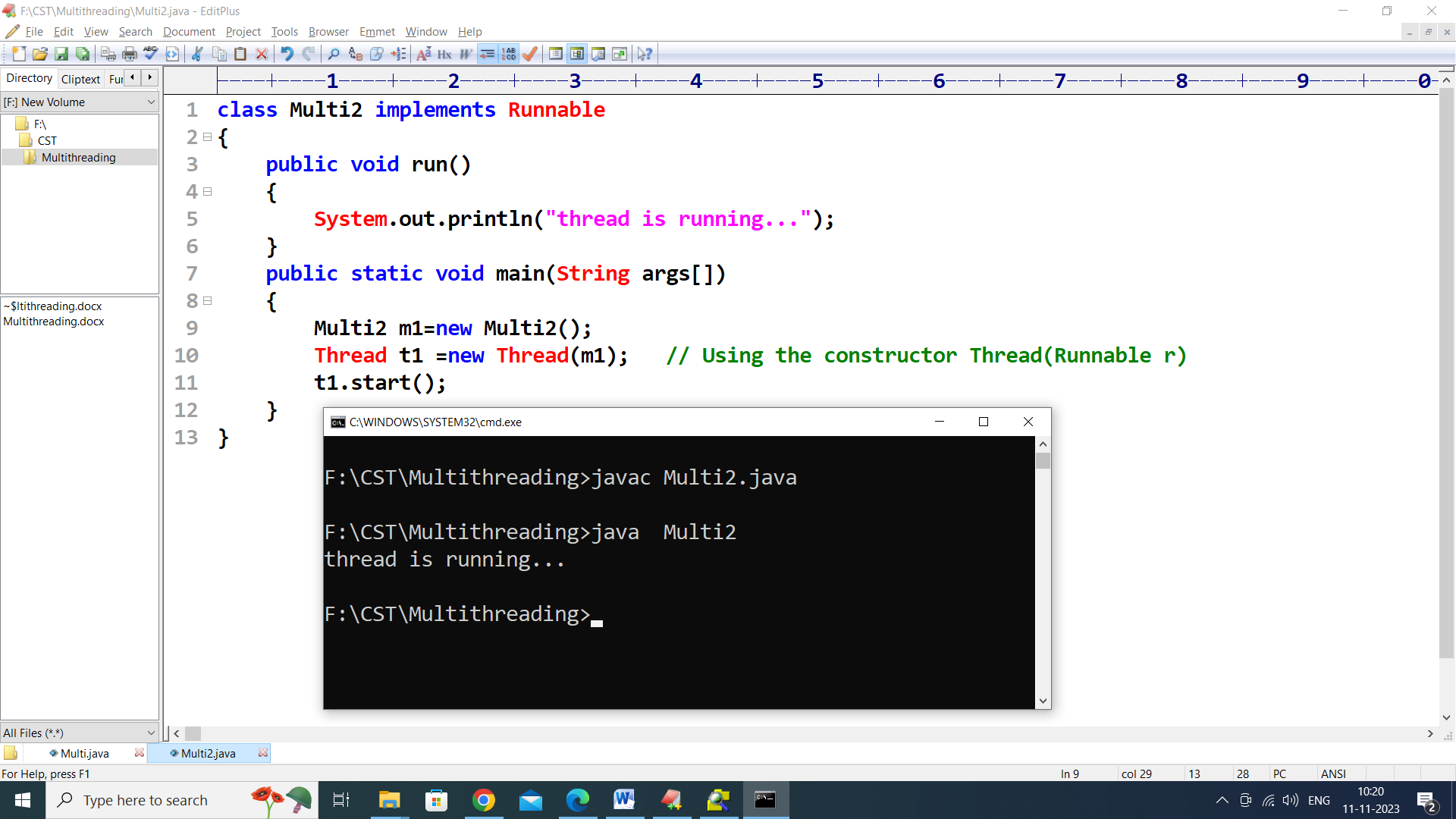
The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

### 1) Java Thread Example by extending Thread class



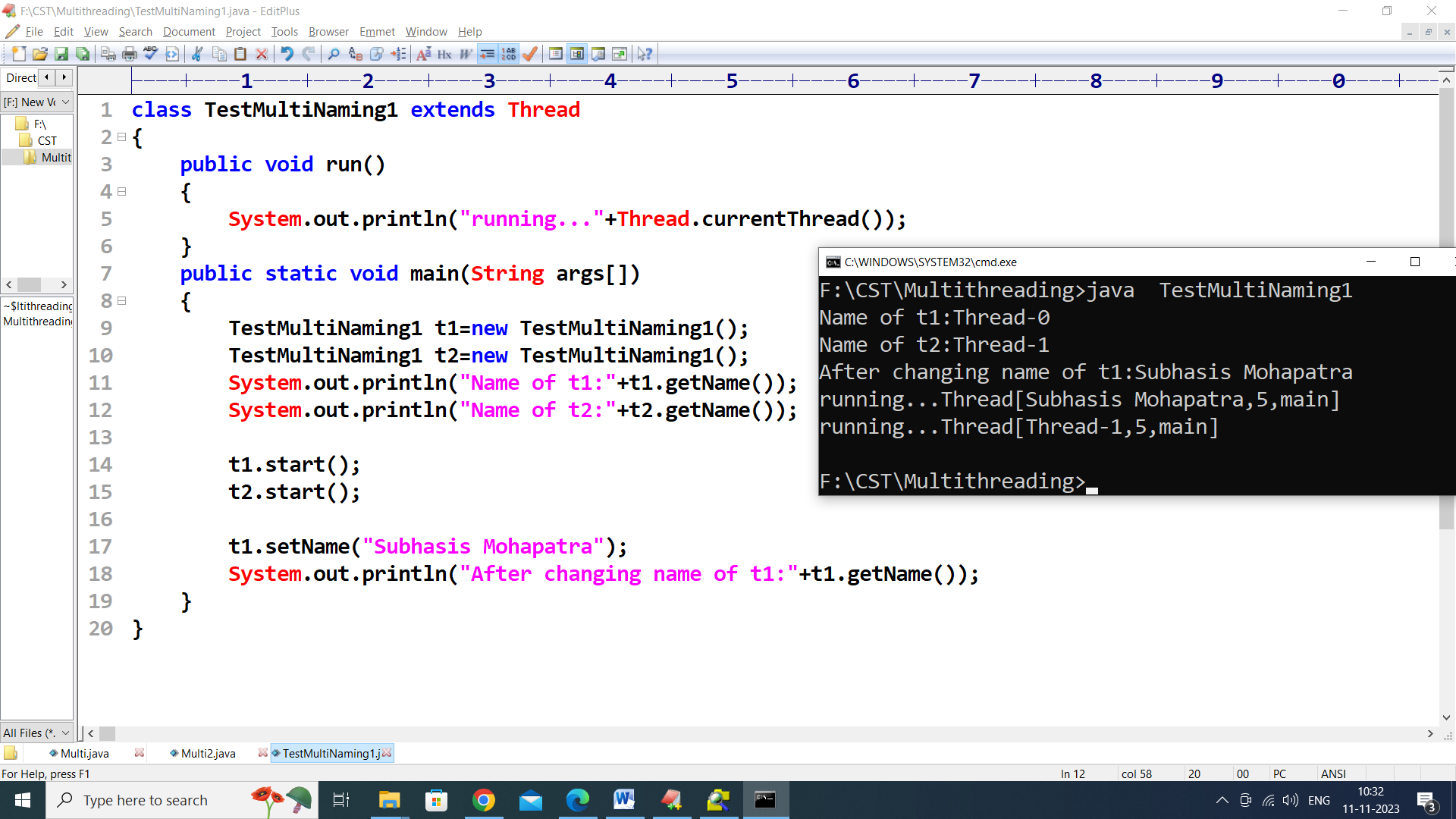
### 2) Java Thread Example by implementing Runnable interface

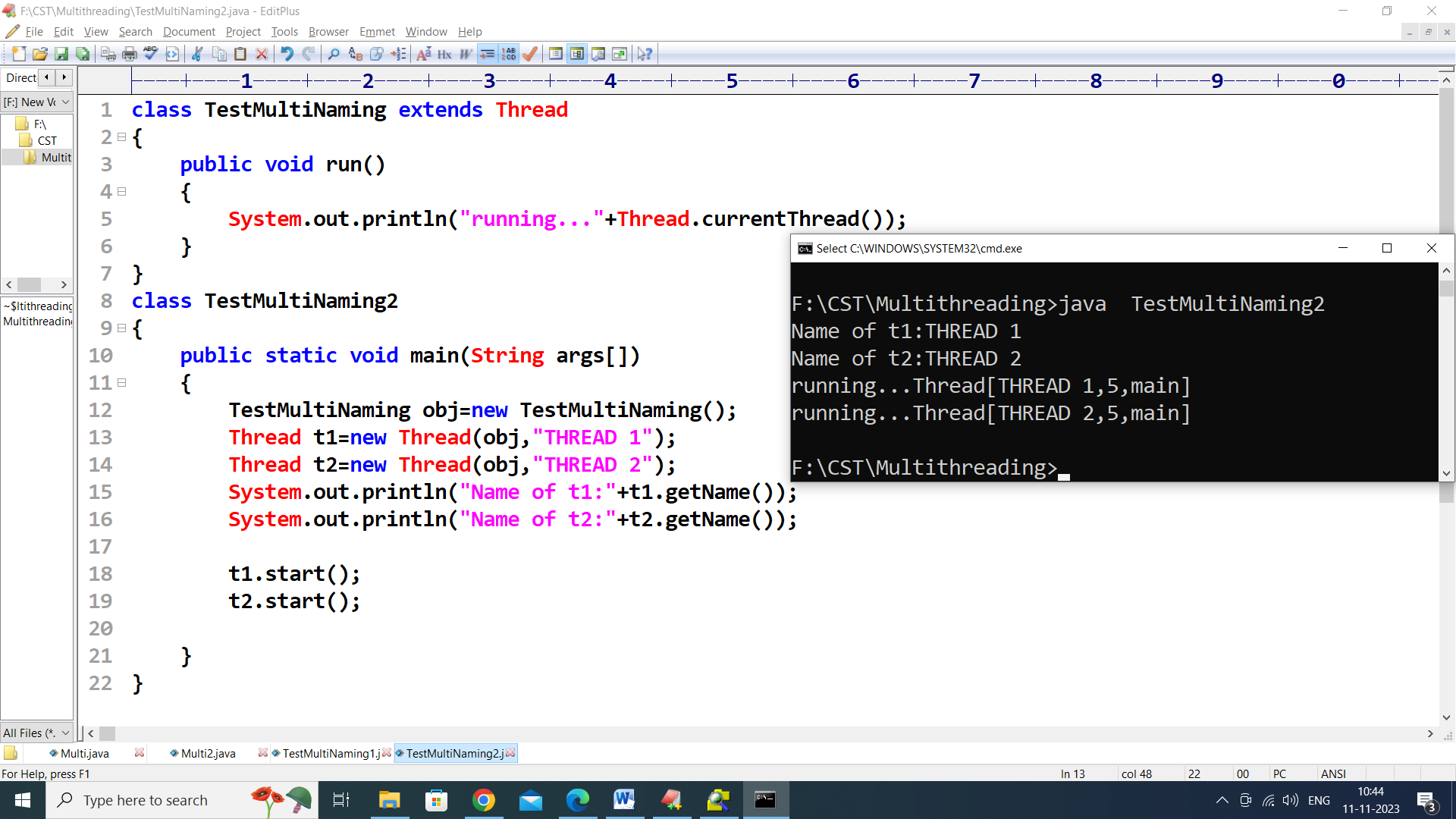


# Naming Thread and Current Thread

## Naming Thread

1. **public** String getName(): is used to **return** the name of a thread.
2. **public** **void** setName(String name): is used to change the name of a thread.



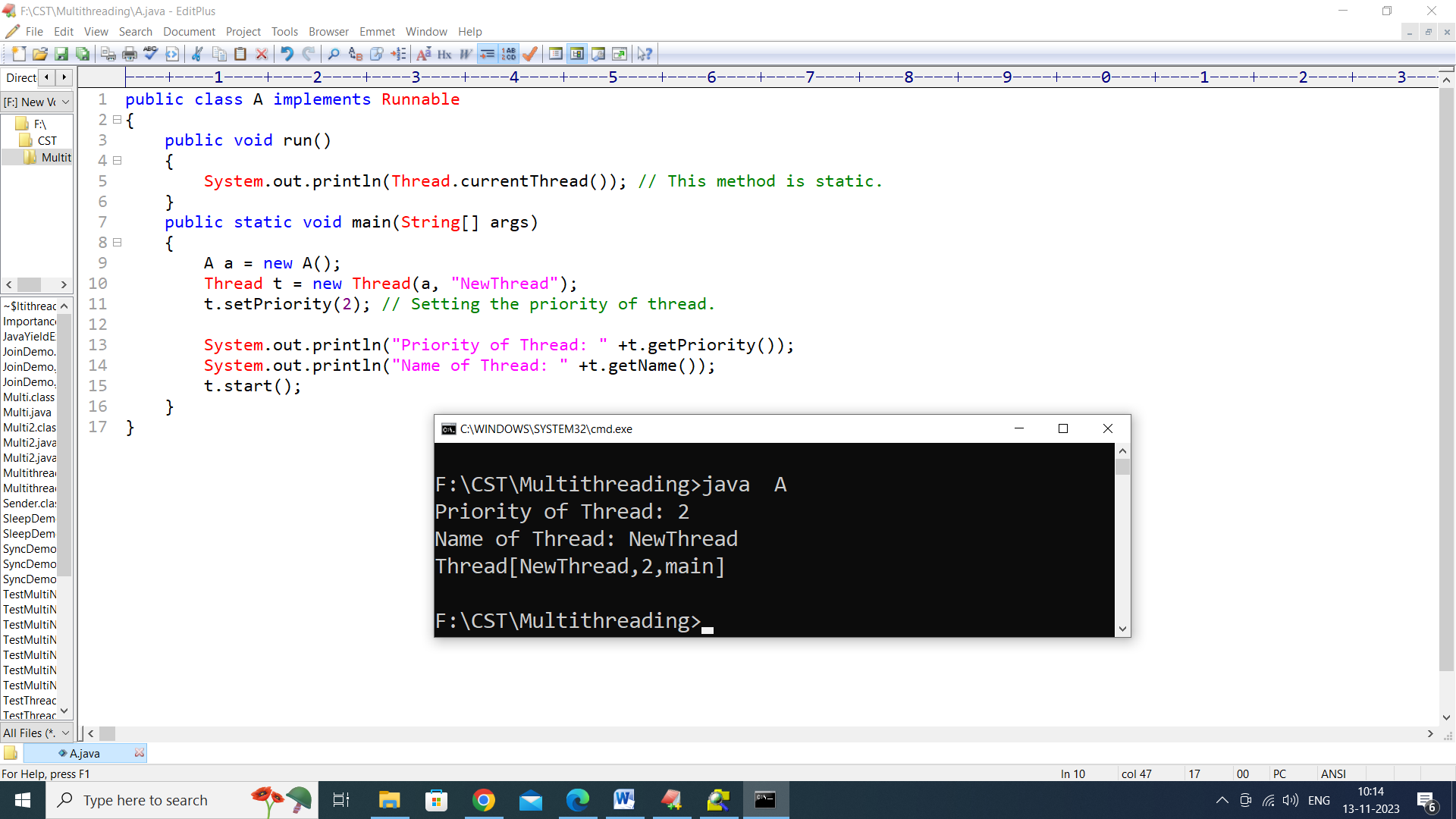


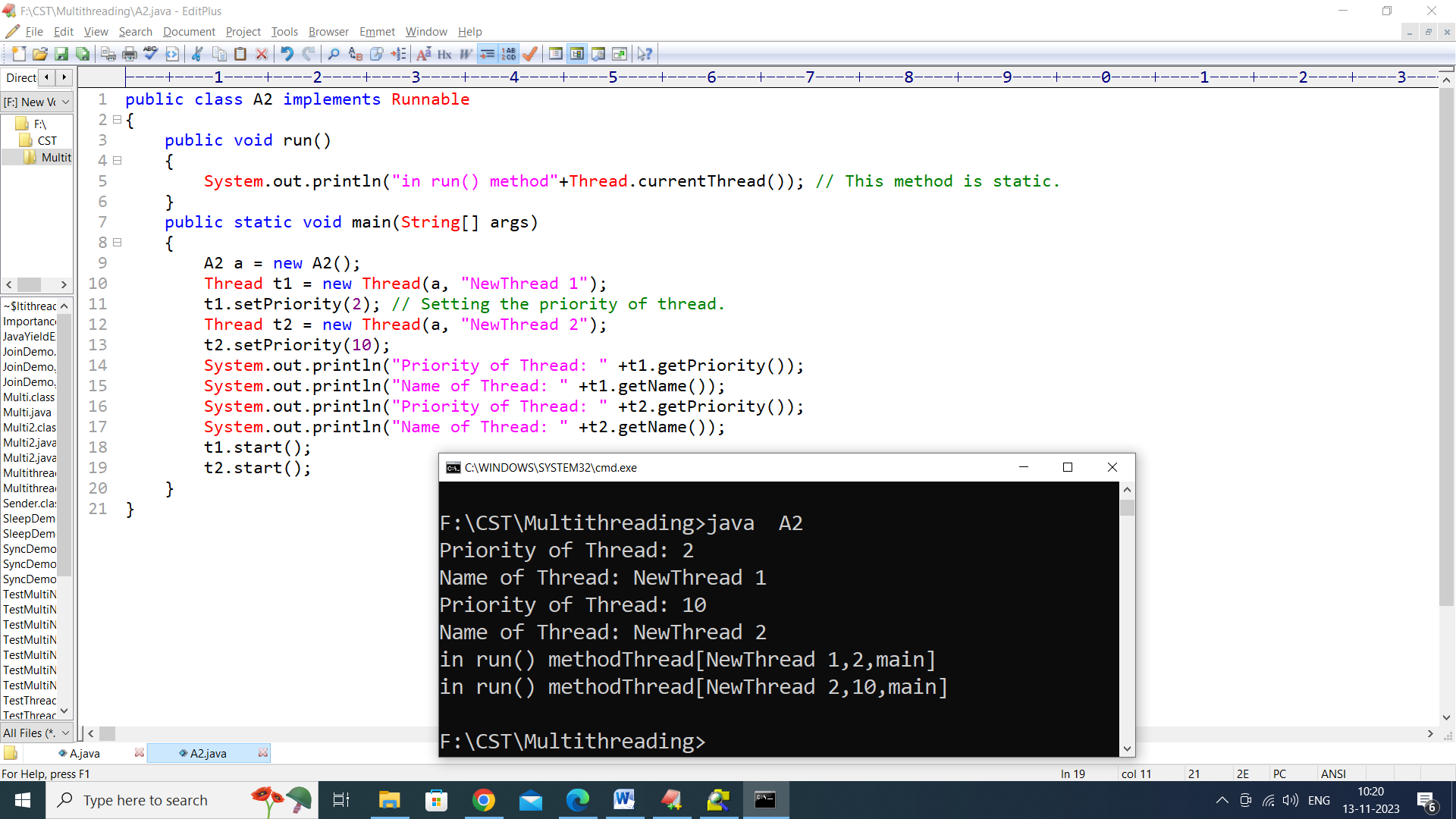
# Priority of a Thread (Thread Priority)

## 3 constants defined in Thread class:

1. public static int MIN\_PRIORITY
2. public static int NORM\_PRIORITY
3. public static int MAX\_PRIORITY

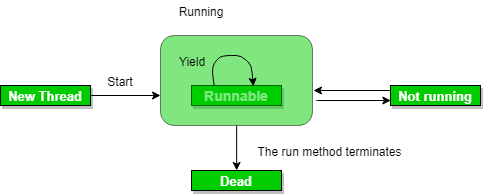
Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.





# 1.Java Thread yield() method

The **yield()** method of thread class causes the currently executing thread object to temporarily pause and allow other threads to execute.



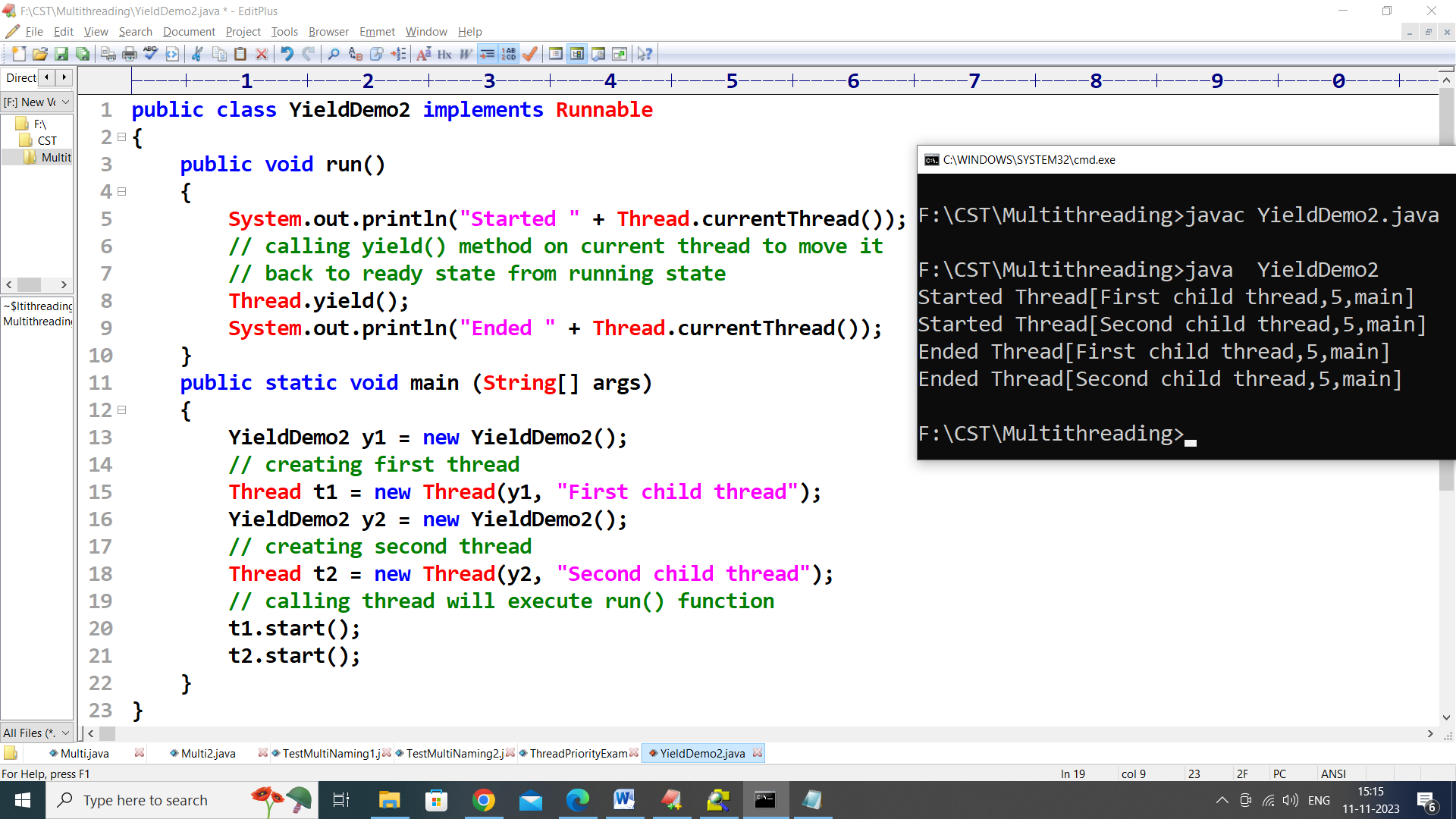
## Syntax

1. **public** **static** **void** yield()

## Return

This method does not return any value.

## Example



### 2. sleep() Method

This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

**Syntax:**

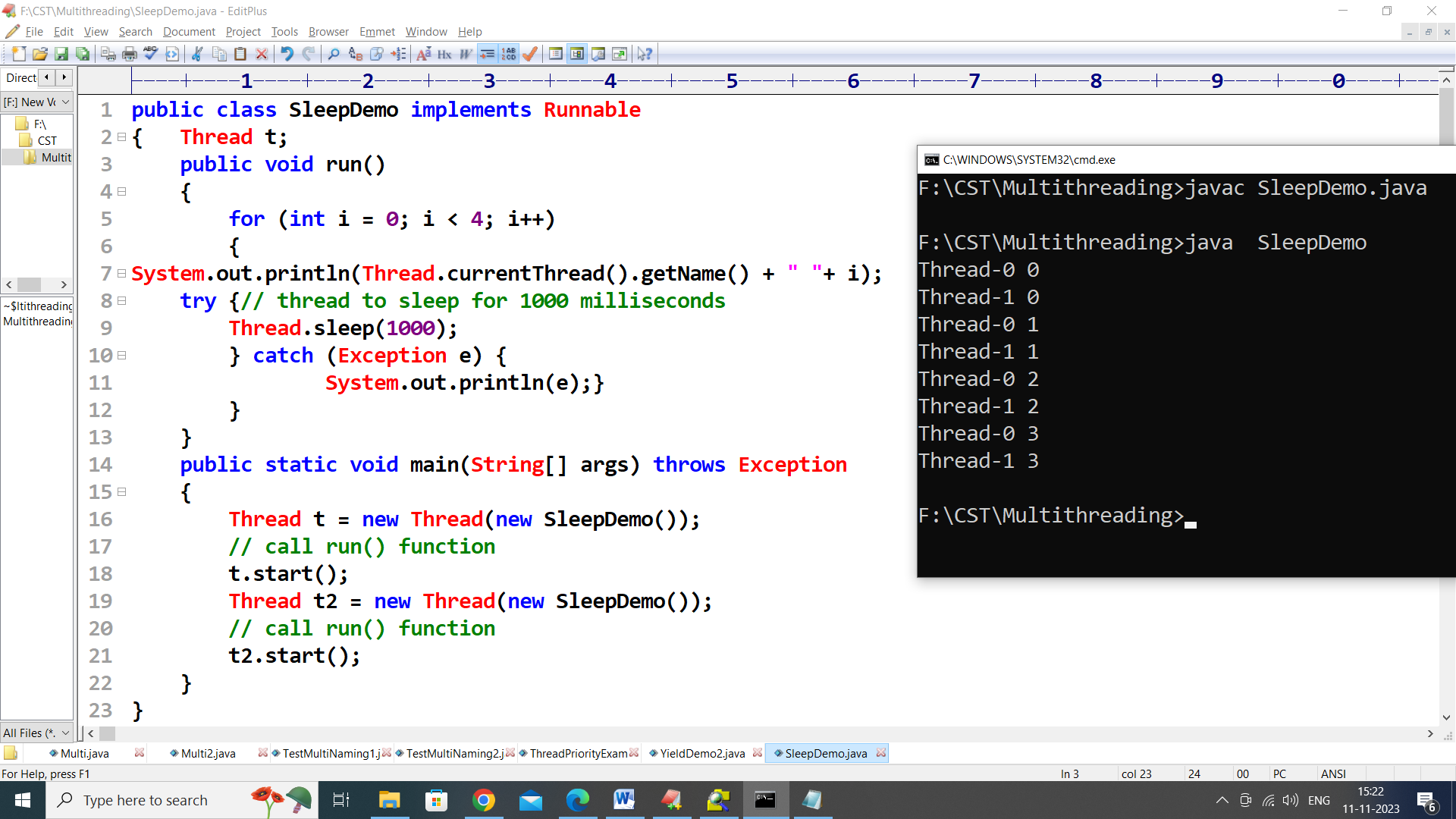
// sleep for the specified number of milliseconds

public static void sleep(long millis) throws InterruptedException

//sleep for the specified number of milliseconds plus nano seconds

public static void sleep(long millis, int nanos)

throws InterruptedException



### 3. join() Method

The join() method of a Thread instance is used to join the start of a thread’s execution to the end of another thread’s execution such that a thread does not start running until another thread ends. If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing. The join() method waits at most this many milliseconds for this thread to die. A timeout of 0 means to wait forever

**Syntax:**

// waits for this thread to die.

public final void join() throws InterruptedException

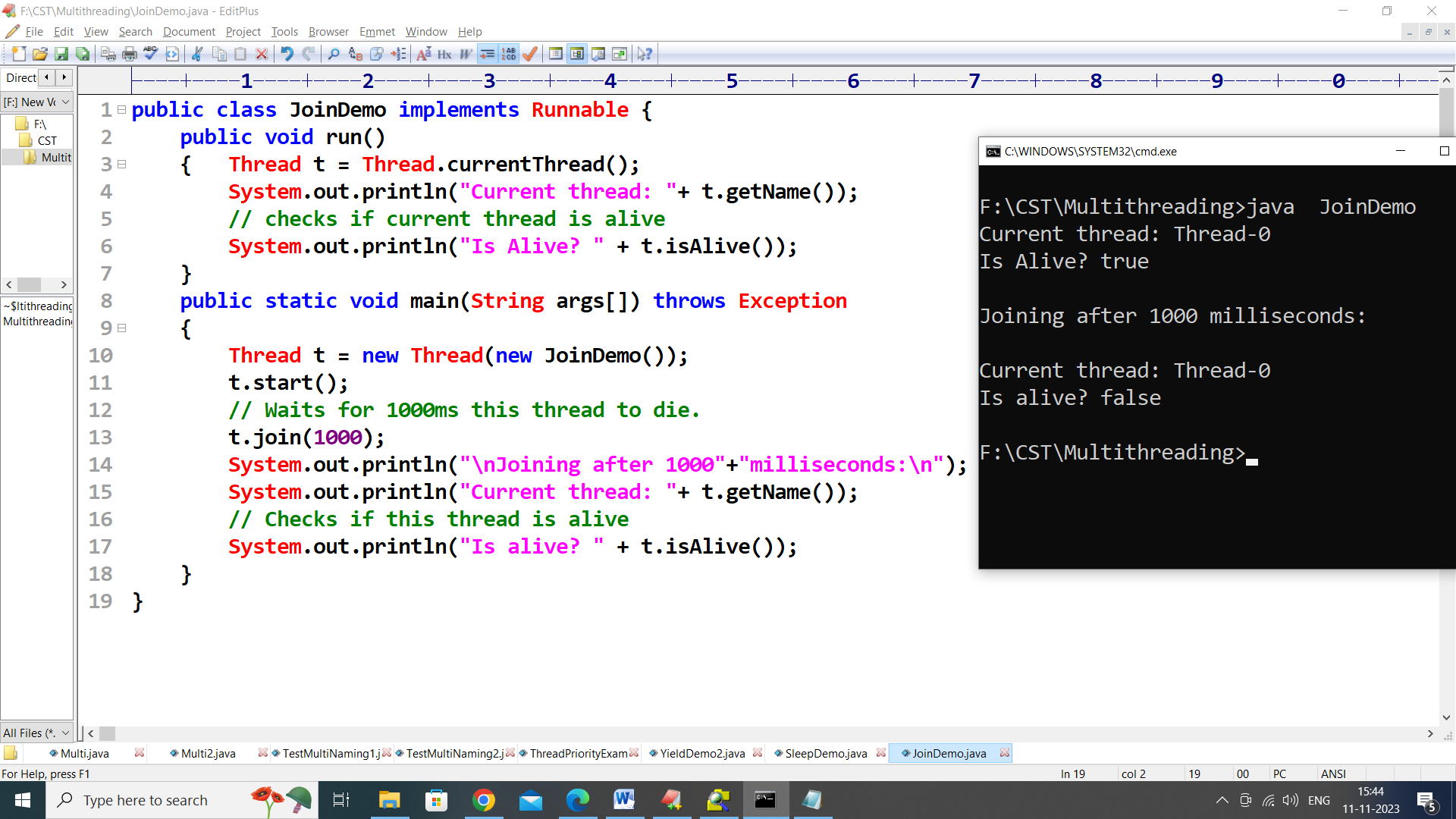
// waits at most this much milliseconds for this thread to die

public final void join(long millis)

throws InterruptedException

// waits at most milliseconds plus nanoseconds for this thread to die.

The java.lang.Thread.join(long millis, int nanos)



# Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

### Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

### Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

### Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   1. Synchronized method.
   2. Synchronized block.
   3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

### General Form of Synchronized Block

// Only one thread can execute at a time.

// sync\_object is a reference to an object

// whose lock associates with the [monitor](https://www.geeksforgeeks.org/monitors-in-process-synchronization/).

// The code is said to be synchronized on

// the monitor object

synchronized(sync\_object)

{

// Access shared variables and other

// shared resources

}

Example

// A Java program to demonstrate working of

// synchronized.

import java.io.\*;

import java.util.\*;

// A Class used to send a message

class Sender {

public void send(String msg)

{

System.out.println("Sending\t" + msg);

try {

Thread.sleep(1000);

}

catch (Exception e) {

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

}

System.out.println("\n" + msg + "Sent");

}

}

// Class for send a message using Threads

class ThreadedSend extends Thread {

private String msg;

Sender sender;

// Receives a message object and a string

// message to be sent

ThreadedSend(String m, Sender obj)

{

msg = m;

sender = obj;

}

public void run()

{

// Only one thread can send a message

// at a time.

synchronized (sender)

{

// synchronizing the send object

sender.send(msg);

}

}

}

// Driver class

class SyncDemo {

public static void main(String args[])

{

Sender send = new Sender();

ThreadedSend S1 = new ThreadedSend(" Hi ", send);

ThreadedSend S2 = new ThreadedSend(" Bye ", send);

// Start two threads of ThreadedSend type

S1.start();

S2.start();

// wait for threads to end

try {

S1.join();

S2.join();

}

catch (Exception e) {

System.out.println("Interrupted");

}

}

}

Output-

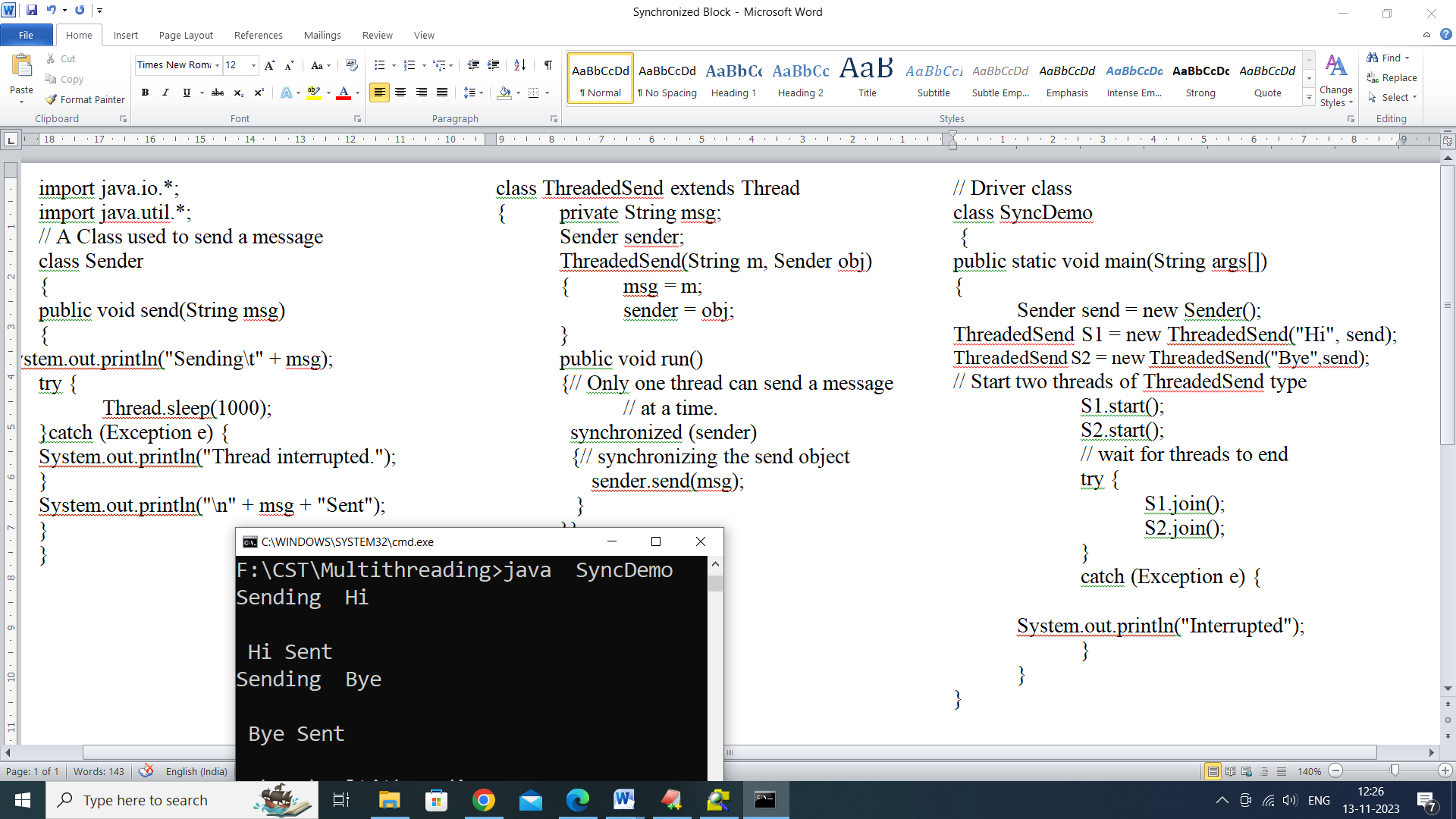
F:\CST\Multithreading>java SyncDemo

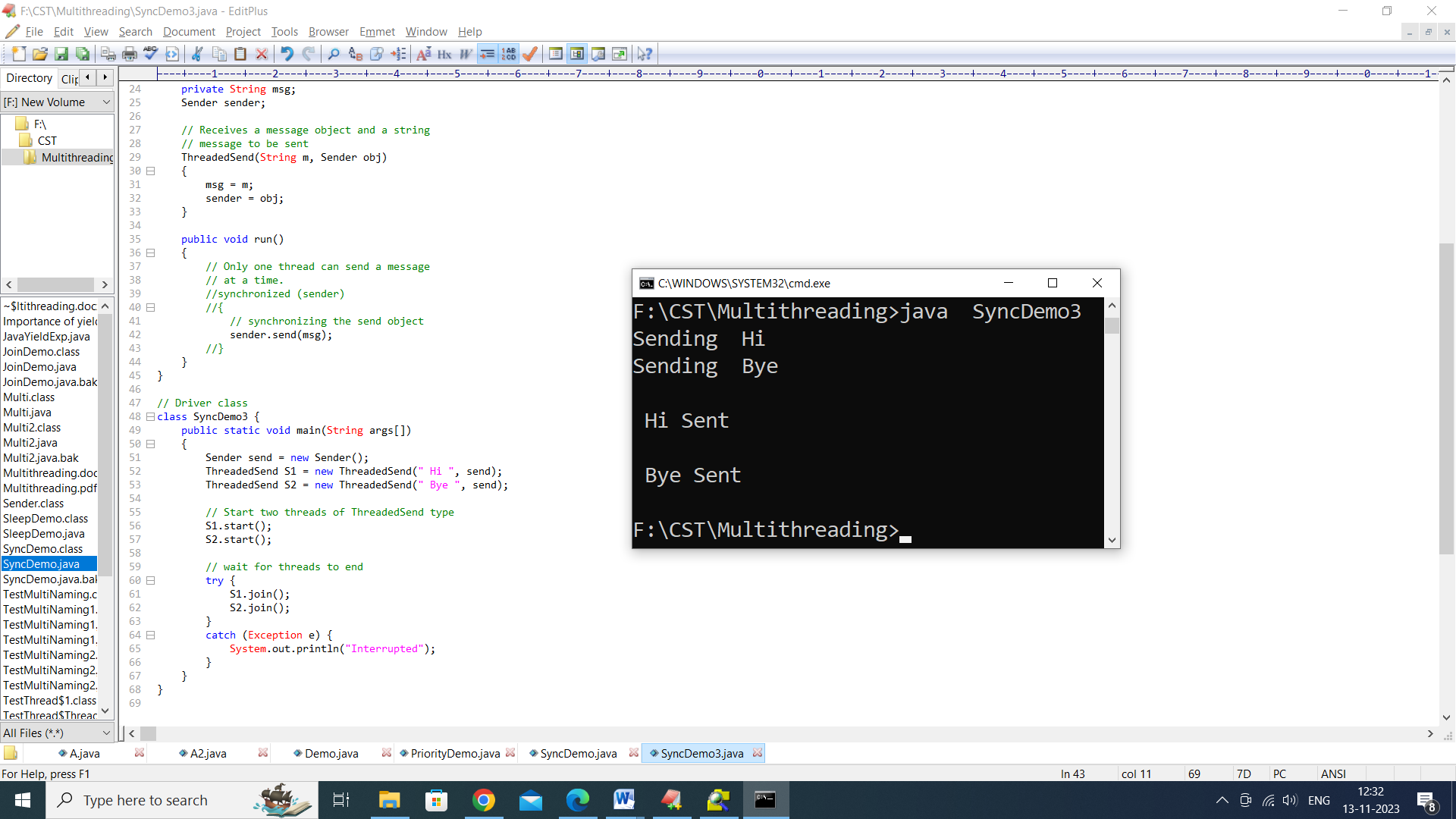
Sending Hi

Hi Sent

Sending Bye

Bye Sent





# Java - Thread Deadlock

# Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object. Here is an example.

**public class TestThread {**

**public static Object Lock1 = new Object();**

**public static Object Lock2 = new Object();**

**public static void main(String args[]) {**

**ThreadDemo1 T1 = new ThreadDemo1();**

**ThreadDemo2 T2 = new ThreadDemo2();**

**T1.start();**

**T2.start();**

**}**

**private static class ThreadDemo1 extends Thread {**

**public void run() {**

**synchronized (Lock1) {**

**System.out.println("Thread 1: Holding lock 1...");**

**try { Thread.sleep(10); }**

**catch (InterruptedException e) {}**

**System.out.println("Thread 1: Waiting for lock 2...");**

**synchronized (Lock2) {**

**System.out.println("Thread 1: Holding lock 1 & 2...");**

**}**

**}**

**}**

**}**

**private static class ThreadDemo2 extends Thread {**

**public void run() {**

**synchronized (Lock2) {**

**System.out.println("Thread 2: Holding lock 2...");**

**try { Thread.sleep(10); }**

**catch (InterruptedException e) {}**

**System.out.println("Thread 2: Waiting for lock 1...");**

**synchronized (Lock1) {**

**System.out.println("Thread 2: Holding lock 1 & 2...");**

**}**

**}**

**}**

**}**

**}**

