



## CET1042B: Object Oriented Programming with C++ and Java

**SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY**

---

**S. Y. B. TECH. COMPUTER SCIENCE AND ENGINEERING  
(CYBERSECURITY AND FORENSICS)**

# Lab. Assignment No. 7

Multithreading using Thread Class and Runnable Interface

# Objective of the Assignment

- To understand Multithreading in Java
- To learn two different ways to create threads in Java.

# Problem Statement (For 7th Assignment)

**Write a program to create a multithreaded calculator that does addition, subtraction, multiplication, and division using separate threads.**

**Additionally also handle '/ by zero' exception by the division method.**

# Problem Statement (For 7th Assignment)

**Print even and odd numbers in increasing order  
using two threads in Java**

# Multitasking vs Multithreading vs Multiprocessing vs Parallel processing

- **Multitasking:** Ability to execute more than one task at the same time is known as multitasking.
- **Multiprocessing:** It is same as multitasking, however in multiprocessing more than one CPUs are involved. On the other hand one CPU is involved in multitasking.
- **Parallel Processing:** It refers to the utilization of multiple CPUs in a single computer system.

**Multithreading:** It is a process of executing multiple threads simultaneously. Multithreading is also known as Thread-based Multitasking.

# Multitasking

- Multitasking is a process of executing multiple tasks simultaneously.
- We use multitasking to utilize the CPU.
- Multitasking can be achieved in two ways:
  1. Process-based Multitasking (Multiprocessing)
  2. 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, memory maps, updating lists, etc.
- It is the feature that allows your computer to run two or more programs concurrently.
- For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.
- In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.



## Conti..

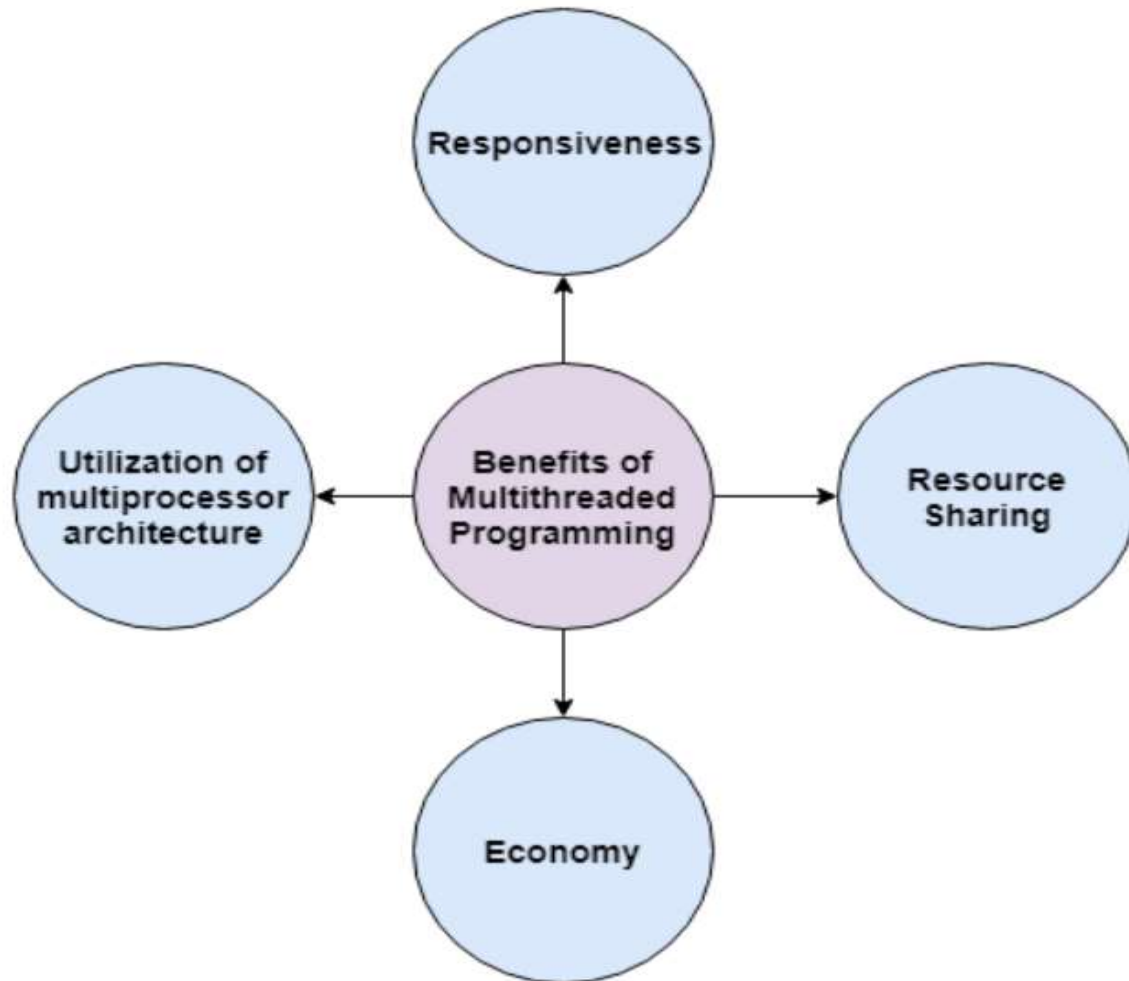
### 2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.
- For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

## Cont...

- Java provides built-in support for multithreaded programming.
- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.
- It is a specialized form of multitasking.

# Need of Multithreading



## Resource Sharing

- All the threads of a process share its resources such as memory, data, files etc. A single application can have different threads within the same address space using resource sharing.

## Responsiveness

- Program responsiveness allows a program to run even if part of it is blocked using multithreading. This can also be done if the process is performing a lengthy operation. For example - A web browser with multithreading can use one thread for user contact and another for image loading at the same time.

## Utilization of Multiprocessor Architecture

In a multiprocessor architecture, each thread can run on a different processor in parallel using multithreading. This increases concurrency of the system. This is in direct contrast to a single processor system, where only one process or thread can run on a processor at a time.

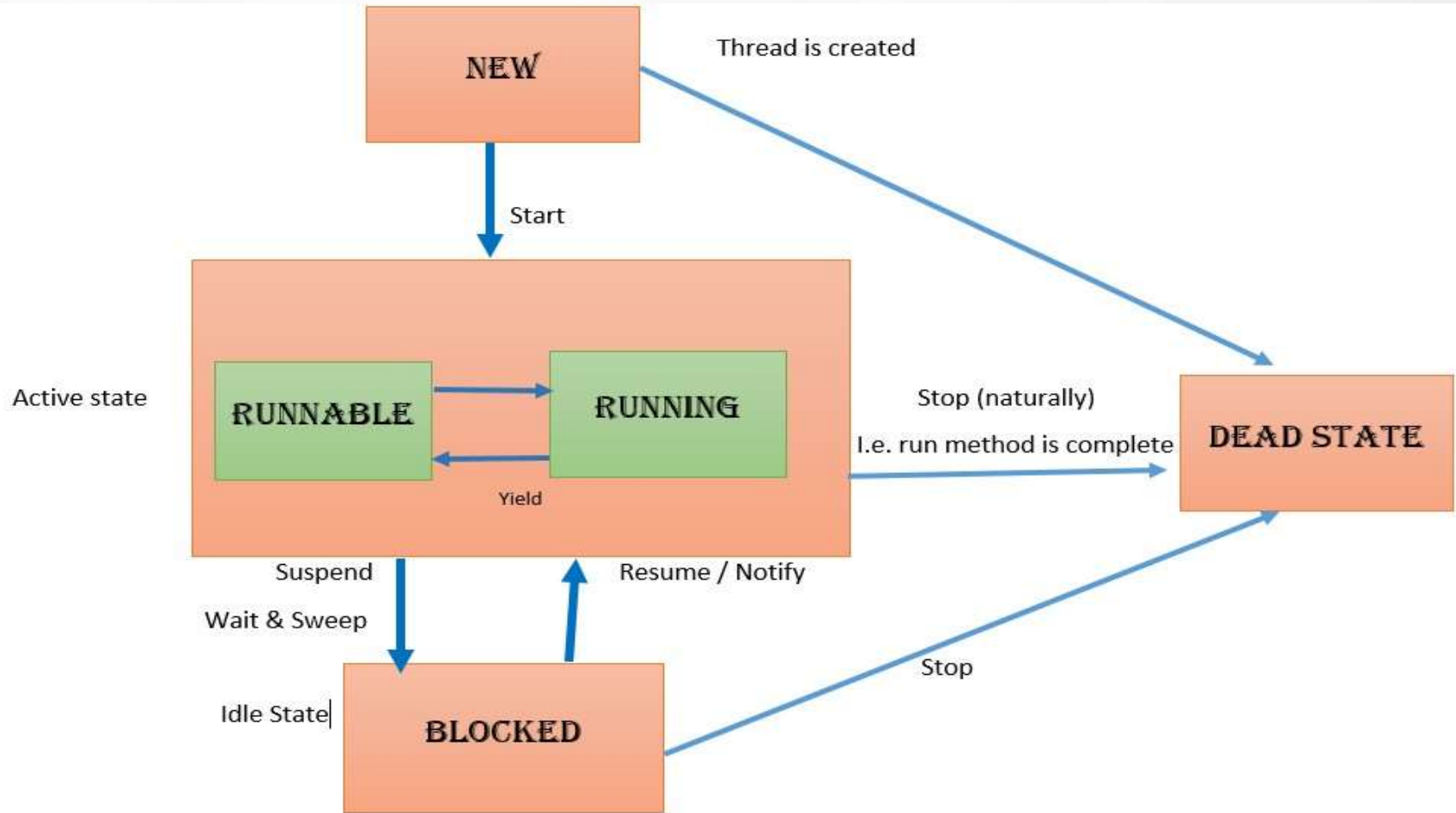
### Economy

- It is more economical to use threads as they share the process resources. Comparatively, it is more expensive and time-consuming to create processes as they require more memory and resources. The overhead for process creation and management is much higher than thread creation and management.

# What is Thread in java

- A thread is a lightweight subprocess, the smallest unit of processing.
- It is a separate path of execution.
- Threads are independent.
- If there occurs exception in one thread, it doesn't affect other threads.
- It uses a shared memory area.

# Life cycle of a Thread





- The life cycle of the thread in java is controlled by JVM.
- The java thread states are as follows:
  - New
  - Runnable
  - Running
  - Non-Runnable (Blocked)
  - Terminated

## Cont..

### 1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

### 2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### 3) Running

The thread is in running state if the thread scheduler has selected it.

### 4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

### 5) Terminated

A thread is in terminated or dead state when its run() method exits.

## Threads can be created by using two mechanisms

1. Extending the Thread class
2. Implementing the Runnable Interface

# Extending the Thread class

Steps:-

1. We create a class that extends the **java.lang.Thread** class.
2. This class overrides the run() method available in the Thread class.
3. A thread begins its life inside run() method.
4. We create an object of our new class and call start() method to start the execution of a thread.
5. Start() invokes the run() method on the Thread object.

# 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:
  1. Thread()
  2. Thread(String name)
  3. Thread(Runnable r)
  4. Thread(Runnable r,String name)

# Thread class Methods

- `getName()`: It is used for Obtaining a thread's name
- `getPriority()`: Obtain a thread's priority
- `isAlive()`: Determine if a thread is still running
- `join()`: Wait for a thread to terminate
- `run()`: Entry point for the thread
- `sleep()`: suspend a thread for a period of time
- `start()`: start a thread by calling its `run()` method

# Creation of Thread in Java

```
public class ThreadDemo {  
  
    public static void main(String[]  
args) {  
        NewThread t1 = new NewThread();  
        t1.start();  
    }  
    class NewThread extends Thread  
    {  
        NewThread()  
        {  
            super("Demo thread"); //Create  
            second thread  
            System.out.println("Child Thread  
"+this);  
        }  
    }  
}
```

```
public void run()  
//This is entry point for second thread  
{ try {  
    for(int i=5;i>0;i--)  
    {  
        System.out.println("Child Thread "+i);  
        Thread.sleep(500);}  
    }  
    catch(InterruptedException e){  
        System.out.println("Child intruptted");  
    }  
    System.out.println("Existing Child thread");  
} }
```

```
Child Thread Thread[Demo thread,5,main]  
Child Thread 5  
Child Thread 4  
Child Thread 3  
Child Thread 2  
Child Thread 1  
Existing Child thread
```



// Java code for thread creation by extending the Thread class

```
class MultithreadingDemo extends Thread {
```

```
    public void run()
    {
```

```
        try {
```

// Displaying the thread that is running

```
        System.out.println(
            "Thread " +
            Thread.currentThread()
                + " is running");
        }
```

```
        catch (Exception e) {
```

// Throwing an exception

```
        System.out.println("Exception is caught");
    }
```

```
    }
}
```

//Main Class

```
public class Multithread {
    public static void main(String[] args)
    {
```

```
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
```

```
            MultithreadingDemo object
                = new
```

```
            MultithreadingDemo();
            object.start();
        }
```

```
    }
}
```

Terminated: Multithread Java Application, C:\Program Fi...

```
Thread Thread[Thread-0,5,main] is running
Thread Thread[Thread-5,5,main] is running
Thread Thread[Thread-4,5,main] is running
Thread Thread[Thread-3,5,main] is running
Thread Thread[Thread-2,5,main] is running
Thread Thread[Thread-1,5,main] is running
Thread Thread[Thread-6,5,main] is running
Thread Thread[Thread-7,5,main] is running
```

# Thread creation by implementing the Runnable Interface

- We create a new class which implements `java.lang.Runnable` interface and override `run()` method.
- Then we instantiate a `Thread` object and call `start()` method on this object.

```
//Java code for thread creation
by implementing
//the Runnable Interface
class MultithreadingDemo1
implements Runnable {
public void run()
{
try {
// Displaying the thread that is
running
System.out.println(
"Thread " +
Thread.currentThread()
+ " is running");
}
catch (Exception e) {
// Throwing an exception
System.out.println("Exception is
caught");
}
}
}
}
```

```
//Main Class
class MultithreadInterface {
public static void main(String[]
args)
{
int n = 8; // Number of threads
for (int i = 0; i < n; i++) {
Thread object
= new Thread(new
MultithreadingDemo1());
object.start();
}
}
}
```

Problems Javadoc Declaration Console × Debug

<terminated> MultithreadInterface [Java Application] C:\Program File

```
Thread Thread[Thread-0,5,main] is running
Thread Thread[Thread-5,5,main] is running
Thread Thread[Thread-4,5,main] is running
Thread Thread[Thread-3,5,main] is running
Thread Thread[Thread-7,5,main] is running
Thread Thread[Thread-1,5,main] is running
Thread Thread[Thread-2,5,main] is running
Thread Thread[Thread-6,5,main] is running
```

# Thread Class vs Runnable Interface

1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like `yield()`, `interrupt()` etc. that are not available in Runnable interface.
3. Using runnable will give you an object that can be shared amongst multiple threads.

# Thread priorities

- **Thread priorities are the integers** which decide how one thread should be treated with respect to the others.
- Thread priority decides **when to switch from one running thread to another**, process is called context switching
- A thread can **voluntarily release control** and the highest priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher priority thread no matter what the lower priority thread is doing. Whenever a higher priority thread wants to run it does.
- To set the priority of the thread ***setPriority()*** method is used which is a method of the class ***Thread Class***.
- In place of defining the priority in integers, we can use ***MIN\_PRIORITY, NORM\_PRIORITY or MAX\_PRIORITY***.

## Cont..

- Priorities in threads is a concept where each thread is having a priority which in layman's language one can say every object is having priority here which is represented by numbers ranging from **1 to 10**.

**The default priority is set to 5 as excepted.**

- Minimum priority is set to 1.
- Maximum priority is set to 10.

Let us do discuss how to get and set priority of a thread in java.

1. **public final int getPriority():** `java.lang.Thread.getPriority()` method returns priority of given thread.
2. **public final void setPriority(int newPriority):**

`java.lang.Thread.setPriority()` method changes the priority of thread to the value `newPriority`.

This method throws `IllegalArgumentException` if value of parameter `newPriority` goes beyond minimum(1) and maximum(10) limit.



## // Java Program to Illustrate Priorities in Multithreading // via help of `getPriority()` and `setPriority()` method

```
// Importing required classes
import java.lang.*;

// Main class
class ThreadDemo1 extends Thread {

    // Method 1
    // run() method for the thread that is called
    // as soon as start() is invoked for thread in
    // main()
    public void run()
    {
        // Print statement
        System.out.println("Inside run method");
    }

    // Main driver method
    public static void main(String[] args)
    {
        // Creating random threads
        // with the help of above class
        ThreadDemo1 t1 = new ThreadDemo1();
        ThreadDemo1 t2 = new ThreadDemo1();
        ThreadDemo1 t3 = new ThreadDemo1();

        // Thread 1
        // Display the priority of above thread
        // using getPriority() method
        System.out.println("t1 thread priority : "
        + t1.getPriority());
```

```
// Thread 2
// Display the priority of above thread
System.out.println("t2 thread priority : "
+ t2.getPriority());

// Thread 3
System.out.println("t3 thread priority : "
+ t3.getPriority());

// Setting priorities of above threads by
// passing integer arguments
t1.setPriority(2);
t2.setPriority(5);
t3.setPriority(8);

// t3.setPriority(21); will throw
// IllegalArgumentException

// 2
System.out.println("t1 thread priority : "
+ t1.getPriority());

// 5
System.out.println("t2 thread priority : "
+ t2.getPriority());

// 8
System.out.println("t3 thread priority : "
+ t3.getPriority());

// Main thread
```



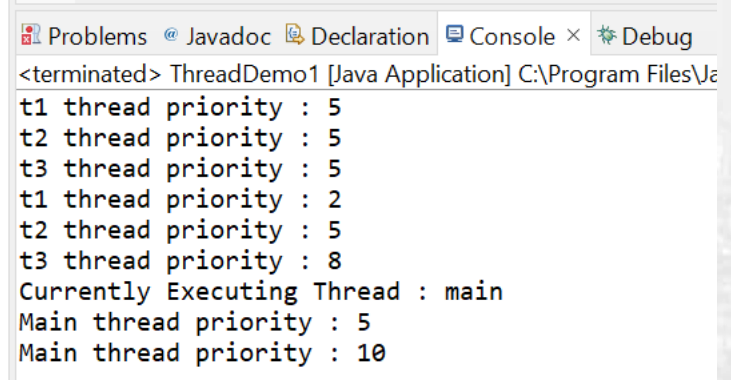
```
// Main thread

// Displays the name of
// currently executing Thread
System.out.println(
    "Currently Executing Thread : "
    + Thread.currentThread().getName());

System.out.println(
    "Main thread priority : "
    + Thread.currentThread().getPriority());

// Main thread priority is set to 10
Thread.currentThread().setPriority(10);

System.out.println(
    "Main thread priority : "
    + Thread.currentThread().getPriority());
}
}
```



The screenshot shows an IDE console window with the following tabs: Problems, Javadoc, Declaration, Console, and Debug. The console output is as follows:

```
<terminated> ThreadDemo1 [Java Application] C:\Program Files\Ja
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Currently Executing Thread : main
Main thread priority : 5
Main thread priority : 10
```

# isAlive() and join()

## isAlive()

- The **isAlive()** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise

*final boolean isAlive()*

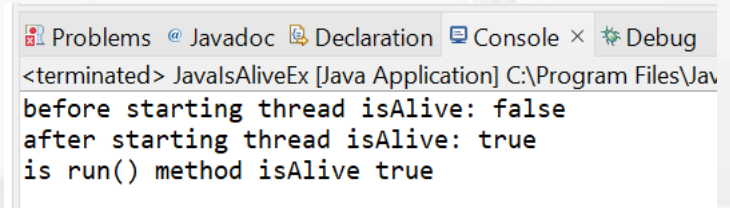
## join()

- This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.
- Additional forms of **join()** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

*final void join() throws InterruptedException*

# Program to check which thread is alive.

```
public class JavaIsAliveEx extends Thread
{
    public void run()
    {
        try
        {
            Thread.sleep(300);
            System.out.println("is run() method isAlive  
"+Thread.currentThread().isAlive());
        }
        catch (InterruptedException ie) {
        }
    }
    public static void main(String[] args)
    {
        JavaIsAliveEx t1 = new JavaIsAliveEx();
        System.out.println("before starting thread isAlive: "+t1.isAlive());
        t1.start();
        System.out.println("after starting thread isAlive: "+t1.isAlive());
    }
}
```



Problems @ Javadoc Declaration Console × Debug  
<terminated> JavaIsAliveEx [Java Application] C:\Program Files\Java  
before starting thread isAlive: false  
after starting thread isAlive: true  
is run() method isAlive true

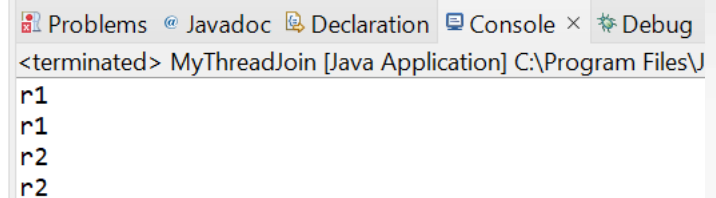
In this example, we are using join() method to ensure that thread finished its execution before starting other thread. It is helpful when we want to executes multiple threads based on our requirement.

```
public class MyThreadJoin extends Thread
{
    public void run()
    {
        System.out.println("r1 ");
        try {
            Thread.sleep(500);
        } catch (InterruptedException ie) { }
        System.out.println("r2 ");
    }
    public static void main(String[] args)
    {
        MyThreadJoin t1=new MyThreadJoin();
        MyThreadJoin t2=new MyThreadJoin();
        t1.start();

        /*try{
            t1.join();//Waiting for t1 to finish
        }catch(InterruptedException ie){}*/

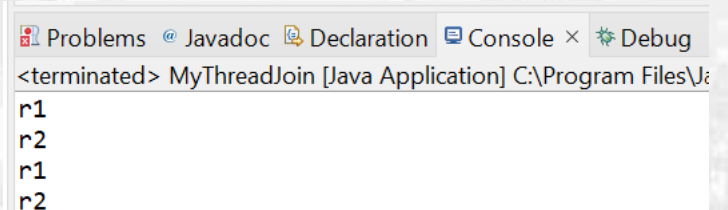
        t2.start();
    }
}
```

Before Join()



```
Problems @ Javadoc Declaration Console × Debug
<terminated> MyThreadJoin [Java Application] C:\Program Files\J
r1
r1
r2
r2
```

After join()



```
Problems @ Javadoc Declaration Console × Debug
<terminated> MyThreadJoin [Java Application] C:\Program Files\J
r1
r2
r1
r2
```

# Key learnings

- Multithreading
- Life Cycle of a Thread
- Two ways to create a Thread
- How to perform multiple tasks by multiple threads
- Thread Scheduler
- Joining a thread

# References

- <https://www.javatpoint.com/multithreading-in-java>
- <https://www.geeksforgeeks.org/multithreading-in-java/>
- <https://beginnersbook.com/2013/03/multithreading-in-java/>
- <https://www.guru99.com/multithreading-java.html>
- [Herbert Schildt](#) Java: A Beginner's Guide, Eighth Edition



Thank You!!