Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically ***a lightweight sub-process***, ***a smallest unit of processing.*** Multiprocessing and multithreading, both are used to achieve multitasking.

But we use ***multithreading*** than ***multiprocessing*** because Threads **share a common 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 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 exception occur 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 by two ways:

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

Process-based Multitasking (*Multiprocessing)*

- Each process have its ***own address in memory*** i.e. each process allocates separate memory area.
- Process is ***heavyweight.***
- Cost of communication between the process is ***high.***
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

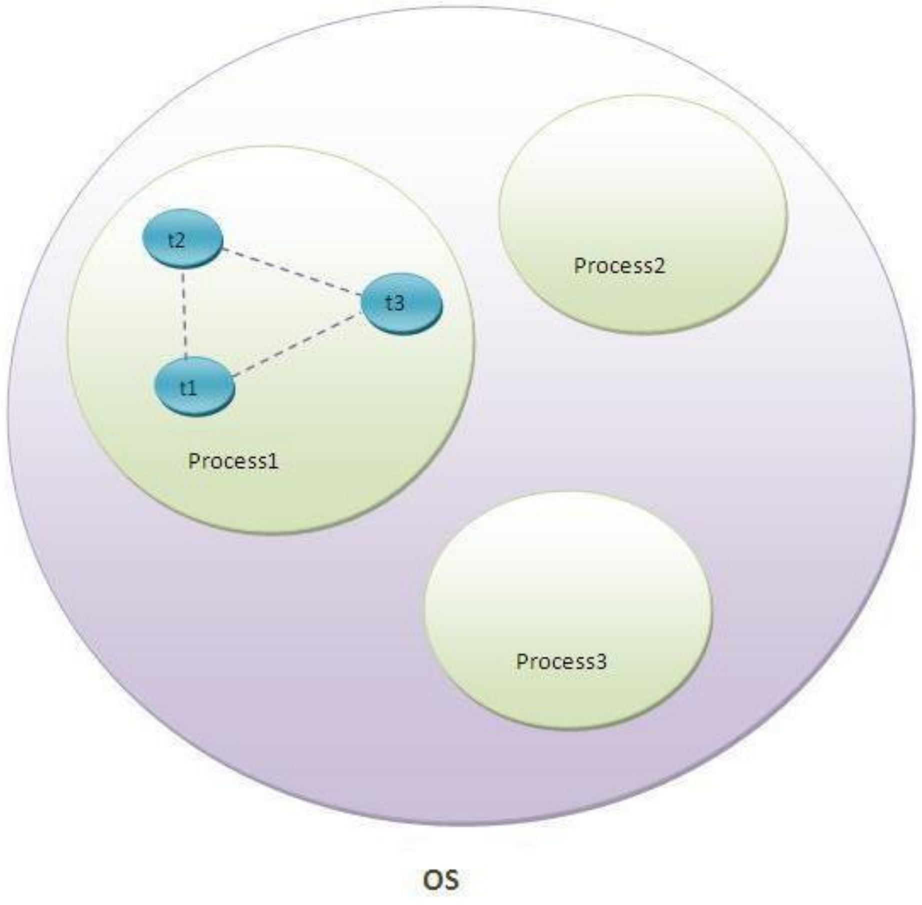
Thread-based Multitasking (*Multithreading*)

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

What is Thread in java

- A thread is a lightweight sub process, a 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 shares a common memory area.

As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

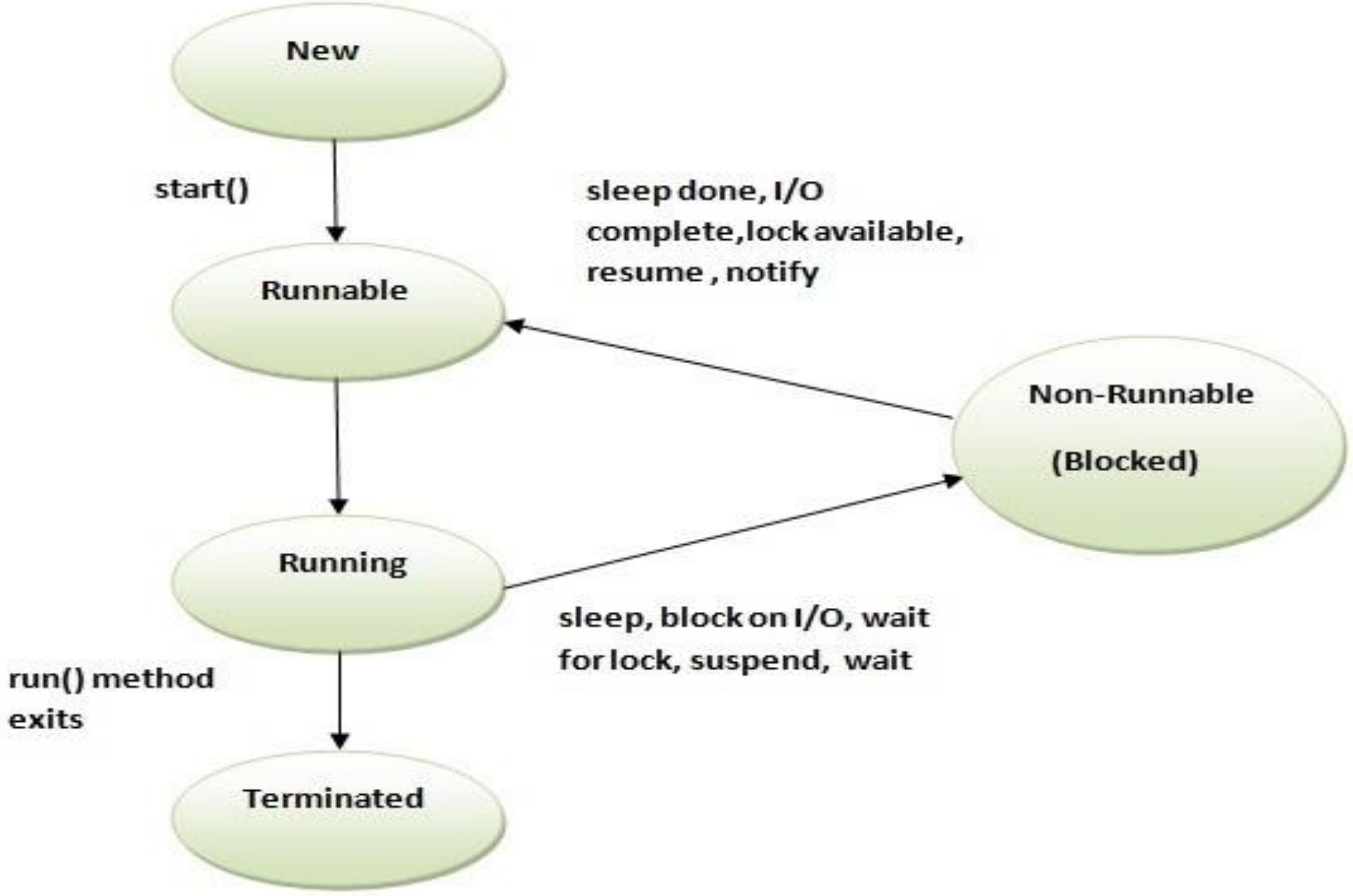


Life cycle of a Thread (*Thread States*)

- A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.
- But for better understanding the threads, we are explaining it in the 5 states.
- 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


New
start()
Runnable
sleep done, I/O complete, lock available, resume , notify
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

States of Threads

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.

How to create thread in java

There are two ways to create a thread:

- By ***extending Thread class***
- 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:

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
- **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **public void join():** waits for a thread to die.
- **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
- **public int getPriority():** returns the priority of the thread.
- **public int setPriority(int priority):** changes the priority of the thread.
- **public String getName():** returns the name of the thread.

Commonly used methods of Thread class:

- **public void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executing thread.
- **public int getId():** returns the id of the thread.
- **public Thread.State getState():** returns the state of the thread.
- **public boolean isAlive():** tests if the thread is alive.
- **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **public void suspend():** is used to suspend the thread(depricated).

Commonly used methods of Thread class:

- **public void resume():** is used to resume the suspended thread(depricated).
- **public void stop():** is used to stop the thread(depricated).
- **public boolean isDaemon():** tests if the thread is a daemon thread.
- **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
- **public void interrupt():** interrupts the thread.
- **public boolean isInterrupted():** tests if the thread has been interrupted.
- **public static boolean interrupted():** tests if the current thread has been interrupted.

- Threads can be created by using two mechanisms :
- Extending the Thread class
- Implementing the Runnable Interface

Extending the Thread class

```
 class MultithreadingDemo  extends Thread {
public void run()
   {
      for (int i=0;i<5;i++)
{
System.out.println(“DJSCE  IOT”);
}
   }
}

// Main Class
public class Multithread {
   public static void main(String[] args)
   {  for (int i=0;i<5;i++)
{
System.out.println(“BEST  STUDENTS”);
               MultithreadingDemo  object  = new  MultithreadingDemo();
                   object.start();
                   }
      }
   }
}
```

Implementing the Runnable Interface

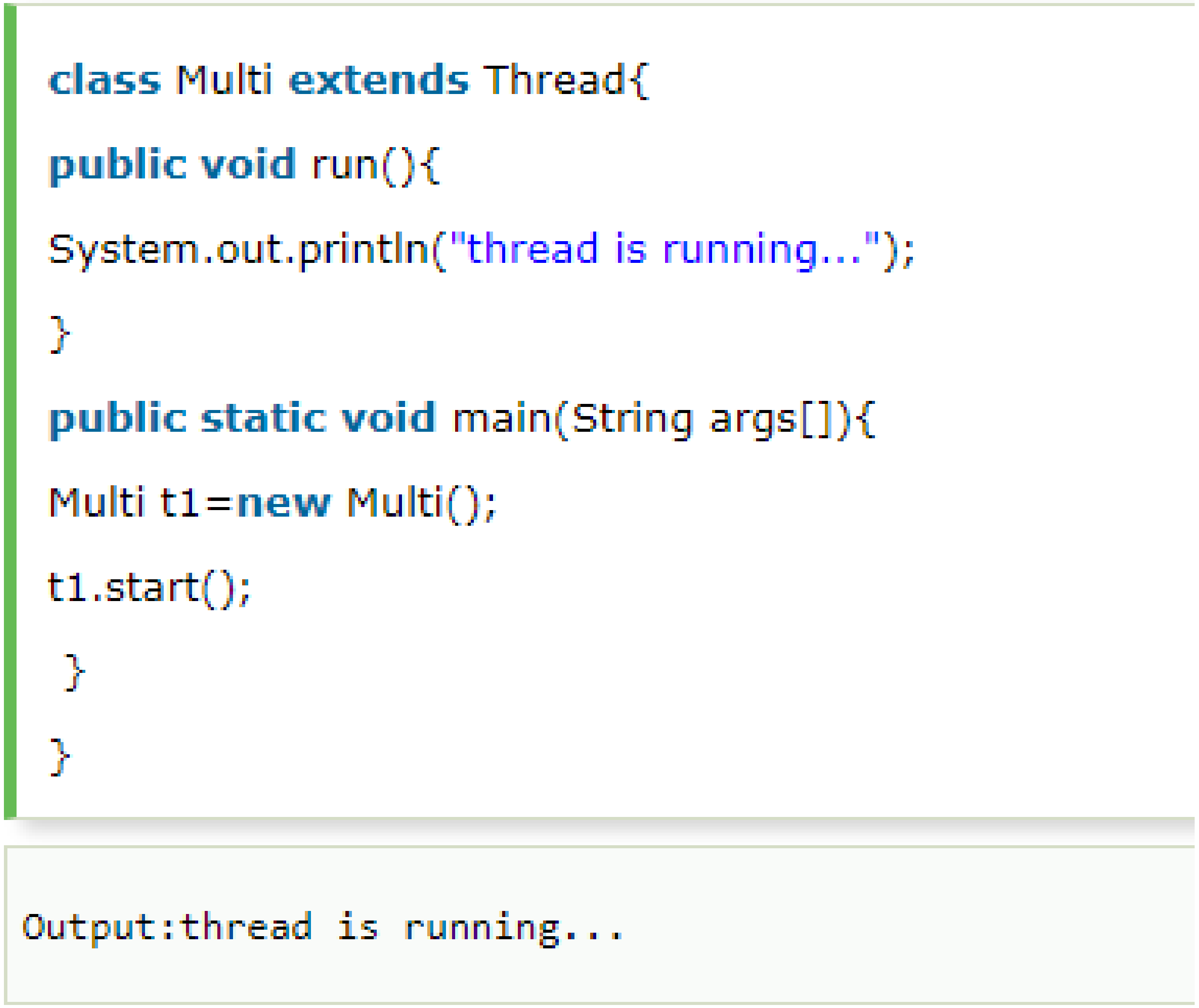
```
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println( "Thread " + Thread.currentThread().getId() + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}
```

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

- **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().
- **public void run():** is used to perform action for a thread.
- **Starting a thread:**
- **start() method** of Thread class is used to start a newly created thread. It performs following tasks:

  A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Java Thread Example by extending Thread class



```
class Multi extends Thread{

public void run(){

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

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start();

 }

}
```

```
Output:thread is running...
```

Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}


public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

```
Output:thread is running...
```

```
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
```

```
System.out.println(name + "Interrupted");
}
System.out.println(name + " exiting.");
}
}
class MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

Sleep method in java

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.
- Syntax of sleep() method in java
- The Thread class provides two methods for sleeping a thread:
- public static void sleep(**long miliseconds**)**throws InterruptedException**
- public static void sleep(**long miliseconds, int nanos**)**throws InterruptedException**

Example of sleep method in java

```java
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

  t1.start();
  t2.start();
 }
}
```

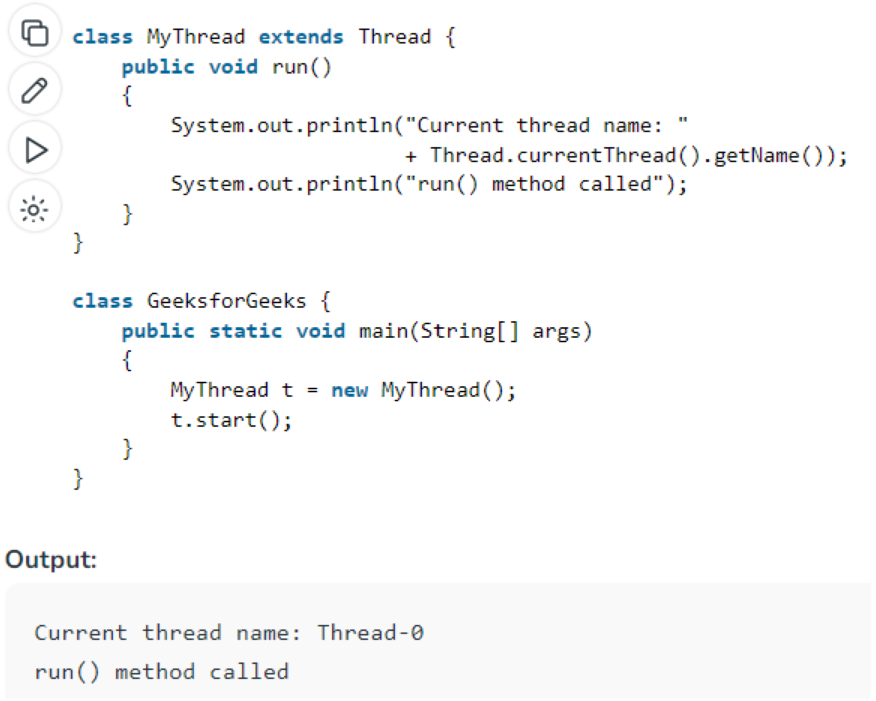
Output:

```
1
1
2
2
3
3
4
4
```

What if we call run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

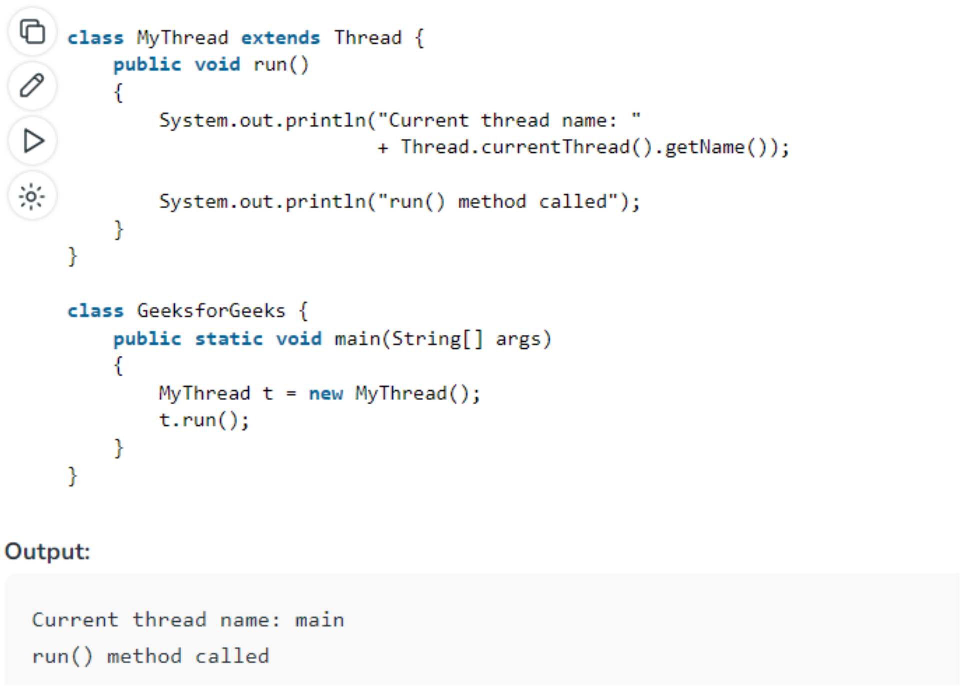
What if We call run() method with run() ; instead of start(); method ?



```
class MyThread extends Thread {
    public void run()
    {
        System.out.println("Current thread name: "
                           + Thread.currentThread().getName());
        System.out.println("run() method called");
    }
}

class GeeksforGeeks {
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

Output:

```
Current thread name: Thread-0
run() method called
```

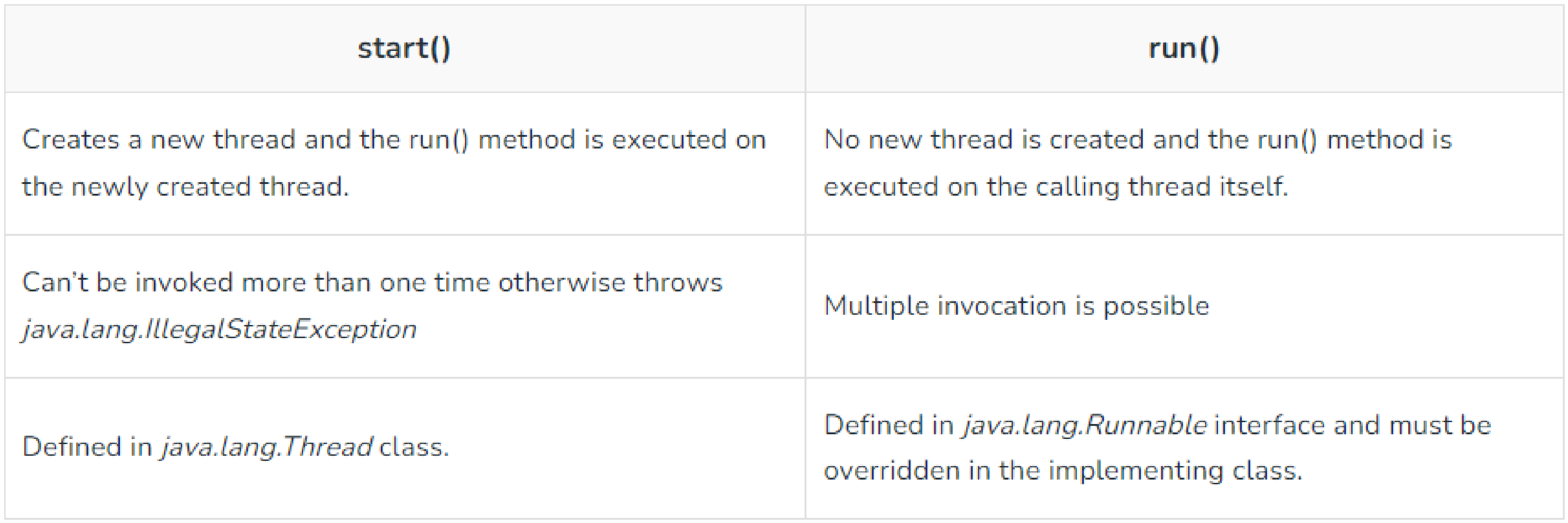


```
class MyThread extends Thread {
    public void run()
    {
        System.out.println("Current thread name: "
                           + Thread.currentThread().getName());

        System.out.println("run() method called");
    }
}

class GeeksforGeeks {
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.run();
    }
}
```

Output:

```
Current thread name: main
run() method called
```

start() Vs. run()

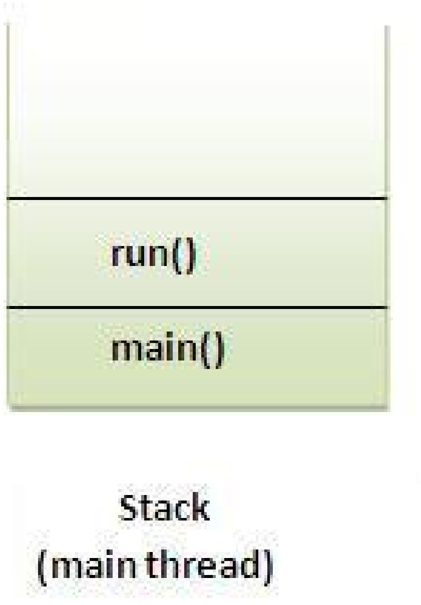


start()	run()
Creates a new thread and the run() method is executed on the newly created thread.	No new thread is created and the run() method is executed on the calling thread itself.
Can't be invoked more than one time otherwise throws *java.lang.IllegalStateException*	Multiple invocation is possible
Defined in *java.lang.Thread* class.	Defined in *java.lang.Runnable* interface and must be overridden in the implementing class.

Thread Class Methods:

Method Signature	Description
String getName()	Retrieves the name of running thread in the current context in String format
void start()	This method will start a new thread of execution by calling run() method of Thread/runnable object.
void run()	This method is the entry point of the thread. Execution of thread starts from this method.
void sleep(int sleeptime)	This method suspend the thread for mentioned time duration in argument (sleeptime in ms)
void yield()	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
void join()	This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution
boolean isAlive()	This method will check if thread is alive or dead

```
class TestCallRun1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestCallRun1 t1=new TestCallRun1();
  t1.run();//fine, but does not start a separate call stack
 }
}
```



Test it Now

```
Output:running...
```

Problem if you direct call run() method

```
class TestCallRun2 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestCallRun2 t1=new TestCallRun2();
  TestCallRun2 t2=new TestCallRun2();

  t1.run();
  t2.run();
 }
}
```

```
Output:1
       2
       3
       4
       5
       1
       2
       3
       4
       5
```

The join() method

- The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.
- **Syntax:**
- **public void join() throws InterruptedException**
- **public void join(long milliseconds) throws InterruptedException**

```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

```
Output:1
       2
       3
       4
       5
       1
       1
       2
       2
       3
       3
       4
       4
       5
       5
```

Example of join(long miliseconds) method

```
class TestJoinMethod2 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod2 t1=new TestJoinMethod2();
 TestJoinMethod2 t2=new TestJoinMethod2();
 TestJoinMethod2 t3=new TestJoinMethod2();
 t1.start();
 try{
  t1.join(1500);
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

```
Output:1
       2
       3
       1
       4
       1
       2
       5
       2
       3
       3
       4
       4
       5
       5
```

```
class demo_class implements Runnable
{public void run()
{ System. out.println("Content in the run method");
}
public static void main(String [] args)
{ demo_class d = new demo_class();
 Thread t = new Thread(d);
t.start();
System. out.println("Thread has started now"); }
 }
```

```
class demo_class implements Runnable {

    public void run() {
        for(int i=0; i<=3; i++) {
            System.out.println("Content in the run
method");
        }
    }
    public static void main(String[] args) {
```

```
demo_class d = new demo_class();
    Thread t = new Thread(d);
    t.start();
    try {
        t.join(); // Wait for thread t to finish
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

```
 for(int i=0; i<=3; i++) {
            System.out.println("Thread has started 
now");
        }
    }
}
```

Write a multithreaded program a java program to print Table of Five, Seven and Thirteen using Multithreading (Use Thread class for the implementation).

```
class TableOf5 extends Thread
{
public void run()
{
for(int i=1;i<=10;i++)
{
try
{
System.out.println("5 * "+i+"= "+(i*5));
Thread.sleep(1000);
}
catch(InterruptedException ie)
{
System.out.println(ie);
}
}
}
}
```

```
class TableOf7 extends Thread
{
public void run()
{
for(int i=1;i<=10;i++)
{
try
{
System.out.println("7 * "+i+"= "+(i*7));
Thread.sleep(1000);
}
catch(InterruptedException ie)
{
System.out.println(ie);
}
}
}
}
```

```
class TableOf13 extends Thread
{
public void run()
{
for(int i=1;i<=10;i++)
{
try
{
System.out.println("13 * "+i+"= "+(i*13));
Thread.sleep(1000);
}
catch(InterruptedException ie)
{
System.out.println(ie);
}
}
}
}
```

```
class PrintTables
{
public static void main(String args[])
{
TableOf5 five=new TableOf5();
TableOf7 seven =new TableOf7();
TableOf13 thirteen=new TableOf13();
five.start();
seven.start();
thirteen.start();
}
}
```

Write a multithreaded program to display /*/*/*/*/*/*/*/* using 2 child threads.

```
class Star extends Thread
{
public void run()
{
int i;
for(i=0;i<8;i++)
{
System.out.print("*");
try
{
sleep(1000);
}
catch(Exception e)
{
}
}
}
}
```

```
class Slash extends Thread
{
public void run()
{
int i;
for(i=0;i<8;i++)
{
System.out.print("/");
try
{
sleep(1000);
}
catch(Exception e)
{
}
}
}
}
```

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

Slash s1= new Slash();
Star s= new Star();
Thread t1=new Thread(s1);
Thread t2=new Thread(s);
t1.start();
t2.start();
}
}
```