Multithreading in Java

Multitasking

- Multitasking is a process of executing multiple tasks simultaneously.
- We use multitasking to utilize the CPU.
- To reduce processor ideal time and to improve performance.

Multitasking can be achieved by two ways:

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

1) 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 processes is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

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

Note: 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.

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 then 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.

Advantages of Threading

- Better utilization of system resources, including the CPU.
- Reduces the computation time.
- Improves performance of an application.
- Threads share the same address space so it saves the memory.

- Context switching between threads is usually less expensive then between processes.
- Cost of communication between threads is relatively low.
- It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
- You **can perform many operations together so it saves time**.
- Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Thread States or Life cycle of a Thread

The life cycle of the thread in java is controlled by JVM. When any thread is created, it goes to different states before it completes its task and is dead. The different states are:

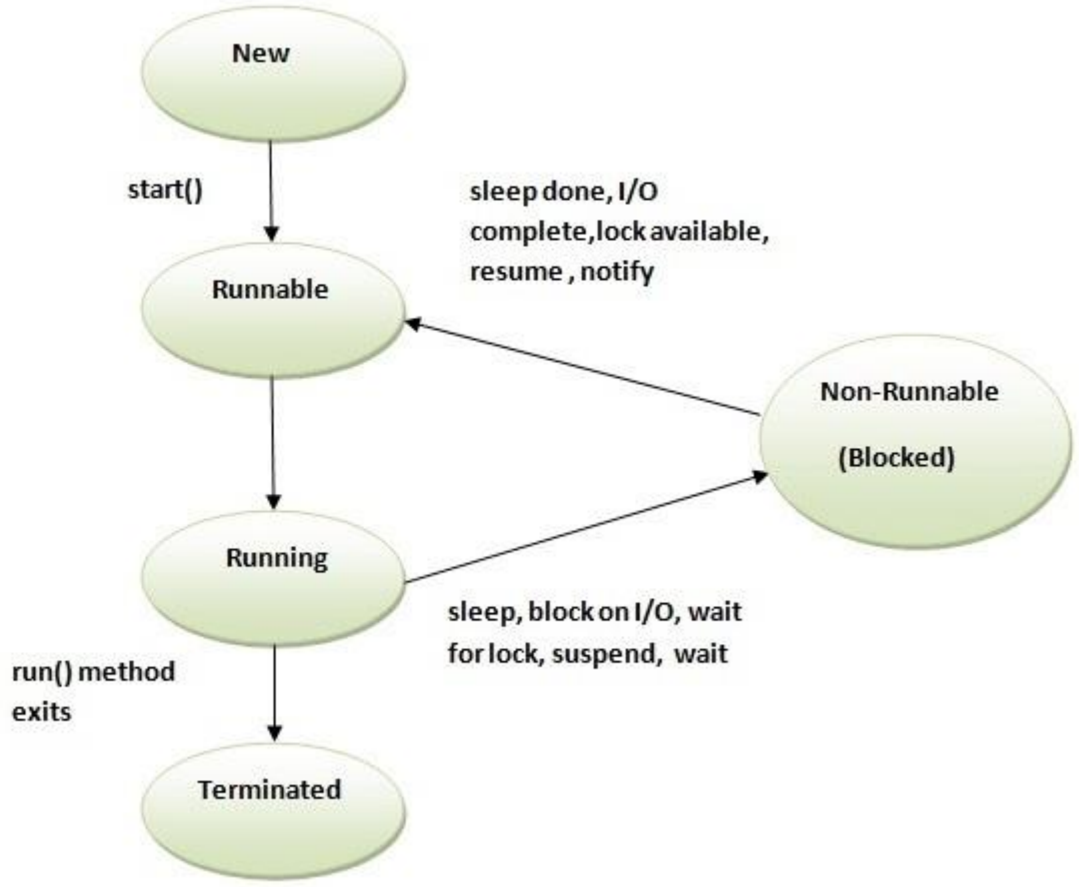
1) New
2) Runnable
3) Running
4) Non-Runnable (Blocked)
5) Terminated

1) New/Born: When a thread is created, it is in new state, in this state thread will not be executed and not sharing time from processor.
2) Runnable/Ready: When the start() method is called on the thread object, the thread is in runnable state. In this state the thread is executing and sharing time from processor.
3) Running: If thread scheduler allocates processor to a thread it will go in Running state OR A thread currently being executed by the CPU is in running state.
4) **Non-Runnable (Blocked):** This is the state when the thread is still alive, but is currently not eligible to run. A running thread may go to a blocked state due to any of the following conditions.

- wait() or sleep() method is called
- The thread perform I/O operation.

When a blocked thread is unblocked, it goes to runnable state and not to running state.
5) **Terminated:** A thread is in terminated or dead state when its run() method exits. A thread becomes dead on two occasions.

- If a thread completes its task, exit the running state.
- run() method is aborted(due to exception etc.)


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

Note: According to sun microsystem , there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state.

CREATION OF THREADS

There are two ways to create a thread:

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

1. By extending 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)

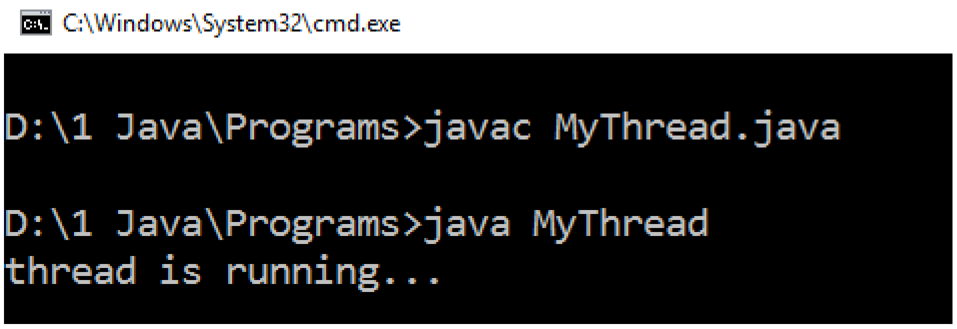
Thread(Runnable r,String name)

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.
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).
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.

Simple Example of Thread creation

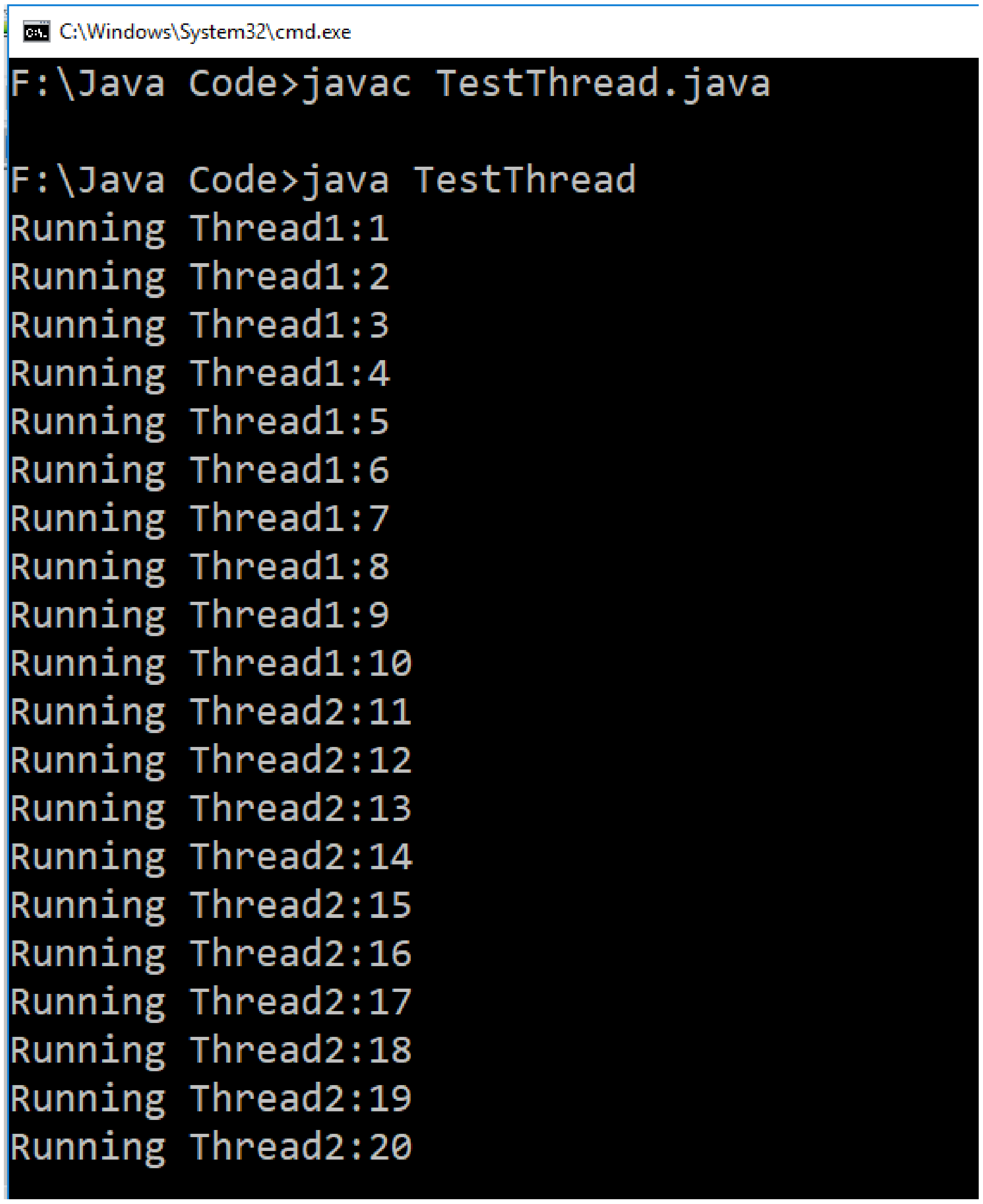
```
class MyThread extends Thread{
   public void run()
{
   System.out.println("thread is running...");
   }
   public static void main(String args[]){
   MyThread t1=new MyThread();

        t1.run();

    }
   }
```



```
D:\1 Java\Programs>javac MyThread.java

D:\1 Java\Programs>java MyThread
thread is running...
```

Example 2:Thread 1,Thread 2 using run() method (call by user).

```
class MyThread1 extends Thread
{
public void run()
        {
 for (int i=1;i<=10;i++)
                {System.out.println("Running Thread1:"+i);
                }}}

class MyThread2 extends Thread
{public void run()
        {for(int i=11;i<=20;i++)
                {System.out.println("Running Thread2:"+i);
                }}}

class TestThread
{
public static void main(String arg[])
        {
          MyThread1 mt1=new MyThread1();
         mt1.run();  //1-10
           MyThread2 mt2=new MyThread2();
         mt2.run();//11-20
        } }
```

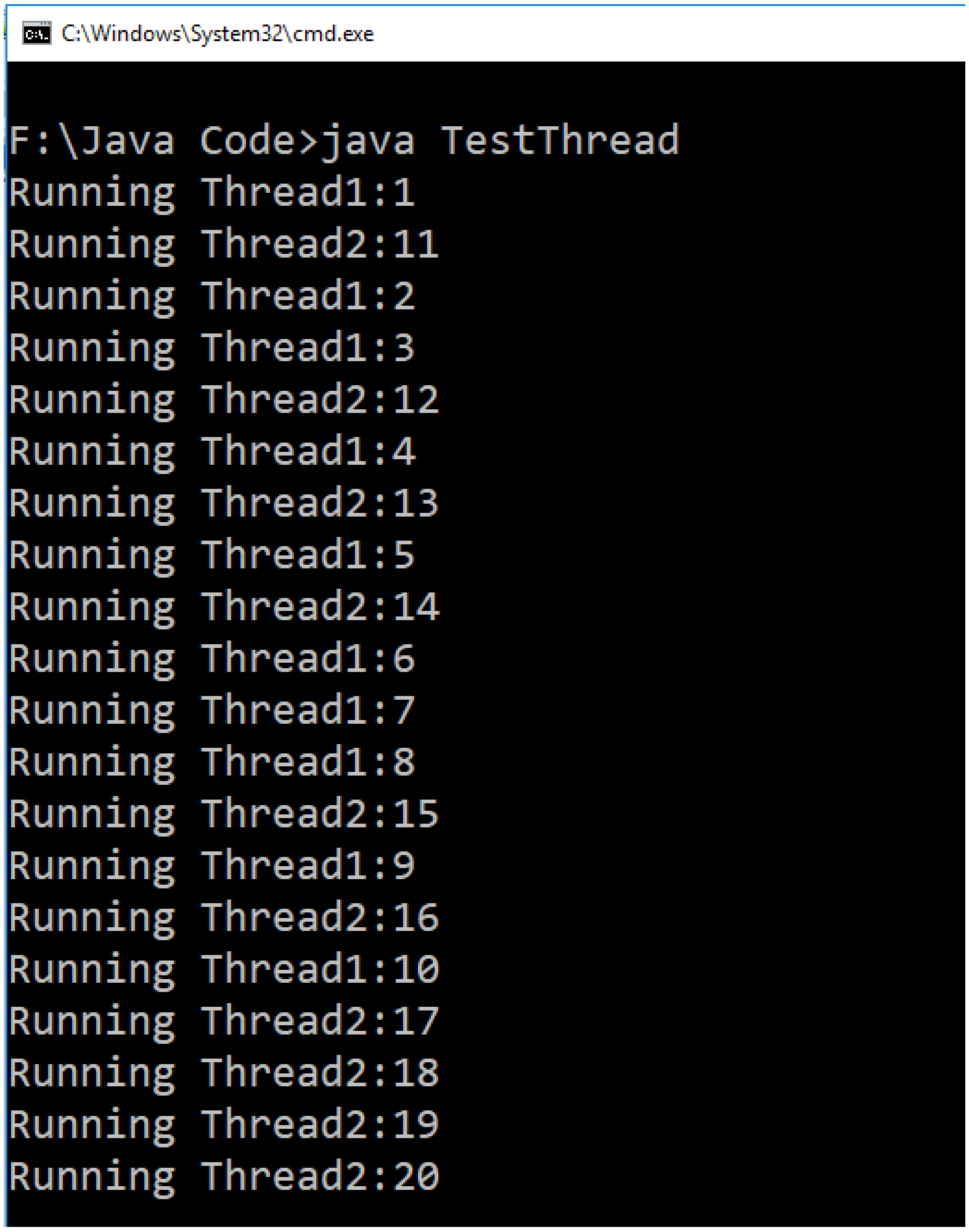


C:\Windows\System32\cmd.exe

```
F:\Java Code>javac TestThread.java

F:\Java Code>java TestThread
Running Thread1:1
Running Thread1:2
Running Thread1:3
Running Thread1:4
Running Thread1:5
Running Thread1:6
Running Thread1:7
Running Thread1:8
Running Thread1:9
Running Thread1:10
Running Thread2:11
Running Thread2:12
Running Thread2:13
Running Thread2:14
Running Thread2:15
Running Thread2:16
Running Thread2:17
Running Thread2:18
Running Thread2:19
Running Thread2:20
```

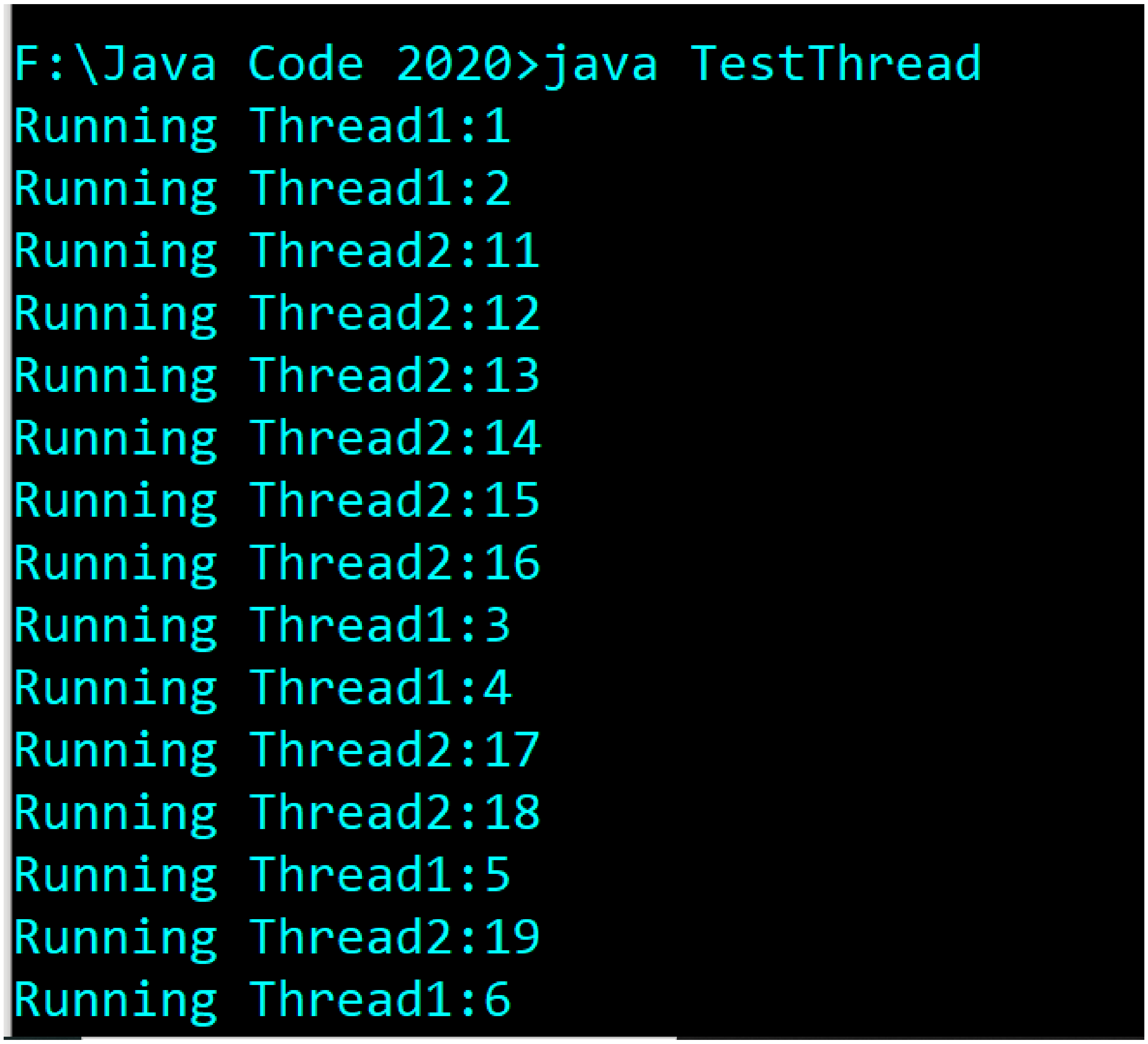
Example 4:Thread 1,Thread 2 using start() method.

```
class MyThread1 extends Thread
{
        public void run()
        {
                for(int i=1;i<=10;i++)
                {
                        System.out.println("Running Thread1:"+i);
                }
        }
}
class MyThread2 extends Thread
{
        public void run()
        {
                for(int i=11;i<=20;i++)
                {
                        System.out.println("Running Thread2:"+i);
                }
        }
}
class TestThread
{
        public static void main(String arg[])
        {
          MyThread1 mt1=new MyThread1();

         mt1.start();

           MyThread2 mt2=new MyThread2();

         mt2.start();
        } }

//will get mixed output.
```



```
F:\Java Code>java TestThread
Running Thread1:1
Running Thread2:11
Running Thread1:2
Running Thread1:3
Running Thread2:12
Running Thread1:4
Running Thread2:13
Running Thread1:5
Running Thread2:14
Running Thread1:6
Running Thread1:7
Running Thread1:8
Running Thread2:15
Running Thread1:9
Running Thread2:16
Running Thread1:10
Running Thread2:17
Running Thread2:18
Running Thread2:19
Running Thread2:20
```

Note: If you run this program again, then this program may give different output:



```
F:\Java Code 2020>java TestThread
Running Thread1:1
Running Thread1:2
Running Thread2:11
Running Thread2:12
Running Thread2:13
Running Thread2:14
Running Thread2:15
Running Thread2:16
Running Thread1:3
Running Thread1:4
Running Thread2:17
Running Thread2:18
Running Thread1:5
Running Thread2:19
Running Thread1:6
```

Output is different; there is no guarantee to get similar output. It varies to system-to-system.

run() vs start()

t.run()

If we use run() method instead of start() method then run() method will be executed just like a normal method (call by main method).

t.start()

if we use start() method then a new thread will be created which is responsible for the execution of run() method.

t.run():

Note:

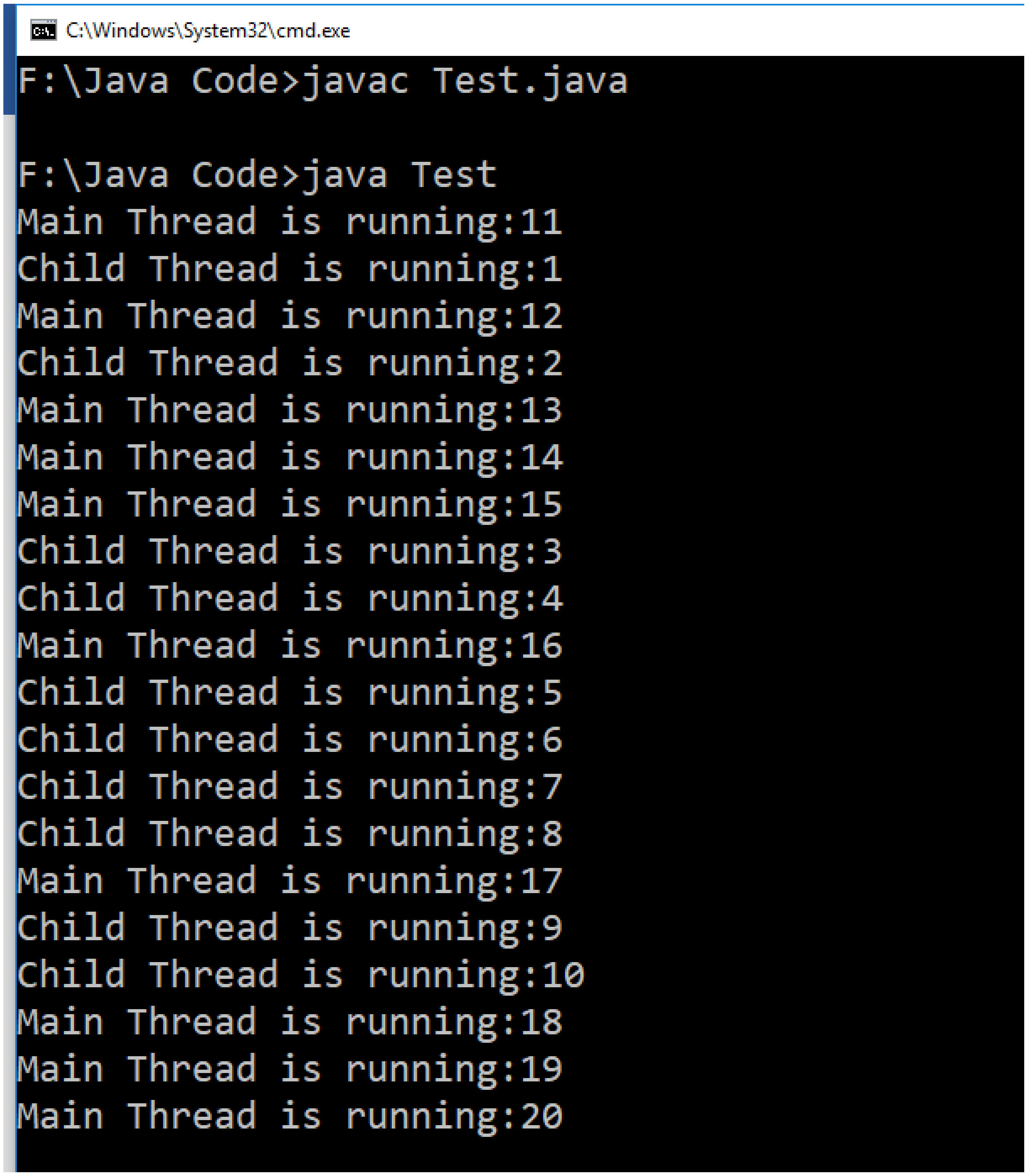
Internal Definition of start()

```
start()
{
        1. Register this thread with thread scheduler.
        2. Perform other mandatory activity
        3. Invoke run() method.
           }
```

Example 1:Child Thread, main Thread using run() method (call by user).

```
1.  class MyThread extends Thread
2.  {
3.  public void run()//overridden method
4.   {
5.  for(int i=1;i<=10;i++)
6.  {
7.   System.out.println("Child Thread is running:"+i);
8.   }
9.  }
10.}
    //1-10 lines defining a thread
    //line 5-8 is job of child thread
```

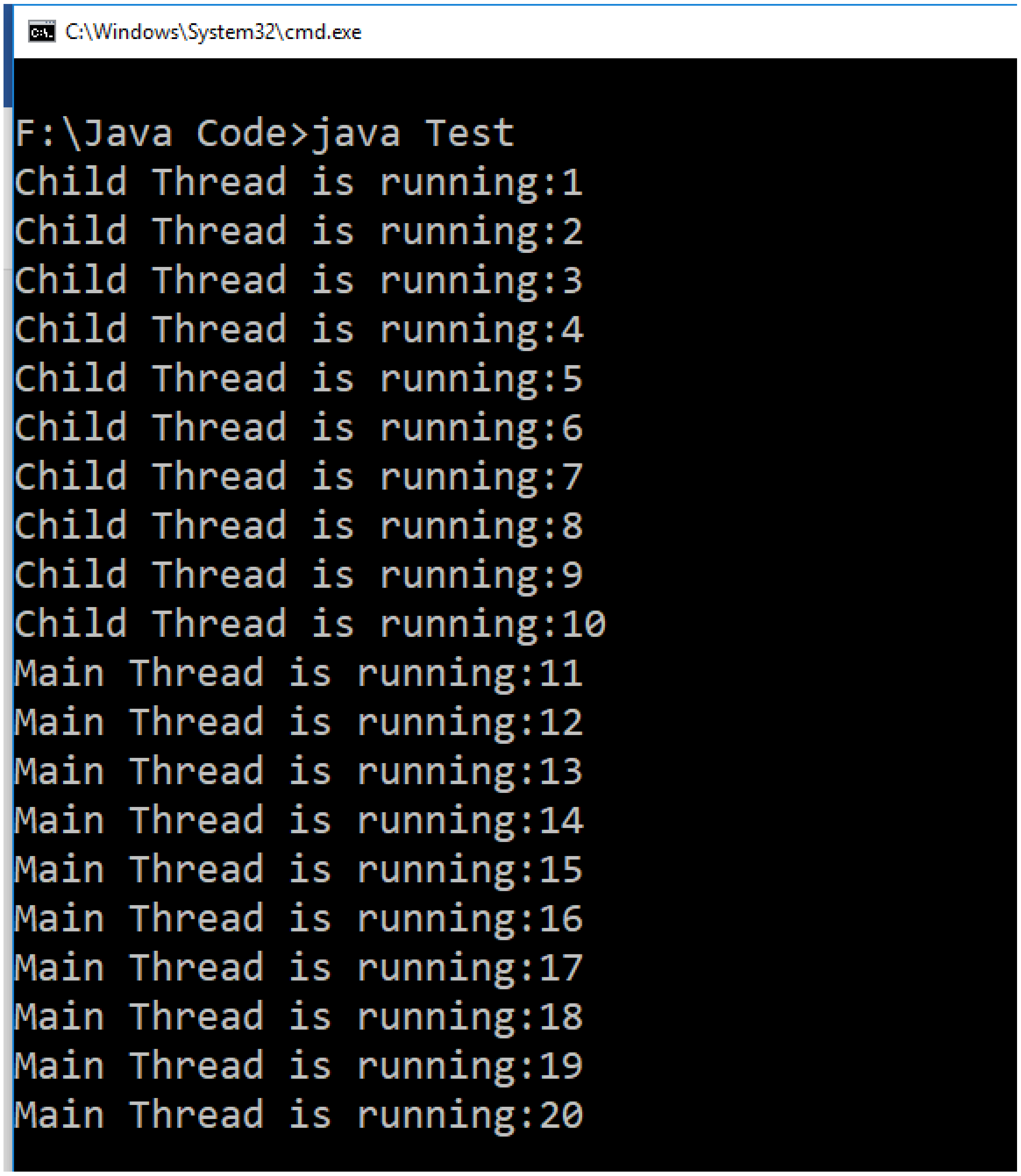
```
 class Test
                {
                public static void main(String args[])//main thread
                {
                MyThread mt=new MyThread();//thread instance creation ,main thread
               create a child thread object
                  mt.start();//main thread start child thread, child thread is responsible to
               execute run()method line(5-8)after start method
                  //two thread are there
                 for(int j=11;j<=20;j++)
                  {
                  System.out.println("Main Thread is running:"+j);
//main thread is responsible to run remaining code
                  }
                  }
                }
// both jobs are going be to be executed simultaneously, we will get mixed output.
```


C:\Windows\System32\cmd.exe

```
F:\Java Code>javac Test.java

F:\Java Code>java Test
Main Thread is running:11
Child Thread is running:1
Main Thread is running:12
Child Thread is running:2
Main Thread is running:13
Main Thread is running:14
Main Thread is running:15
Child Thread is running:3
Child Thread is running:4
Main Thread is running:16
Child Thread is running:5
Child Thread is running:6
Child Thread is running:7
Child Thread is running:8
Main Thread is running:17
Child Thread is running:9
Child Thread is running:10
Main Thread is running:18
Main Thread is running:19
Main Thread is running:20
```

Example 2:Child Thread, main Thread using run() method.

```
class MyThread extends Thread
{
public void run()//overridden method
  {
     for(int i=1;i<=10;i++)
     {
     System.out.println("Child Thread is running:"+i);
    }
  }
}
 class Test
 {
 public static void main(String args[])//main thread
 {
   MyThread mt=new MyThread();
mt.run();
   for(int j=11;j<=20;j++)
   {System.out.println("Main Thread is running:"+j);
           }
   }
 }
```


C:\Windows\System32\cmd.exe

```
F:\Java Code>java Test
Child Thread is running:1
Child Thread is running:2
Child Thread is running:3
Child Thread is running:4
Child Thread is running:5
Child Thread is running:6
Child Thread is running:7
Child Thread is running:8
Child Thread is running:9
Child Thread is running:10
Main Thread is running:11
Main Thread is running:12
Main Thread is running:13
Main Thread is running:14
Main Thread is running:15
Main Thread is running:16
Main Thread is running:17
Main Thread is running:18
Main Thread is running:19
Main Thread is running:20
```

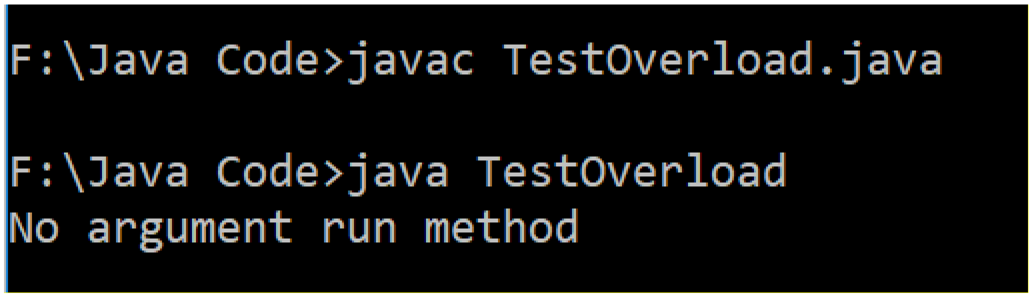
Thread Scheduler in Java

It is a part of JVM
It is responsible to schedule threads i. e. if multiple threads are waiting to get the chance of execution then in which order thread will be Thread Scheduler decides executed.it.
We can't expect exact algorithm followed by thread scheduler, it is varied from JVM to JVM. Hence we can't expect thread execution order and exact output.
Hence whenever situation comes to multithreading there is no guarantee for exact output but we can provide several possible output.

Overloading of run() method:

Overloading of run method is possible but Thread class start() method can invoke no argument run method,the other overloaded method we have to call explicitly like a normal method call.
Similarly if we overload main() method, but JVM always call String args[] method.
Example:

```
class MyThreadEx extends Thread
{
        public void run()
        {
                        System.out.println("No argument run method:);

        }
        public void run(int i)
        {
                        System.out.println("int argument run method:);

        }

}

class TestOverload
{
        public static void main(String arg[])
        {
         MyThreadEx t1=new MyThreadEx();

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



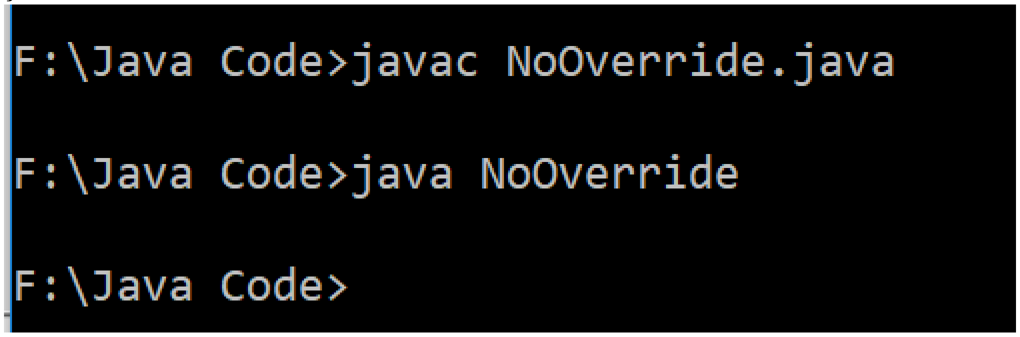
```
F:\Java Code>javac TestOverload.java

F:\Java Code>java TestOverload
No argument run method
```

If we are not overriding run() method:

If u are not overriding run method then Thread class run method will be executed which has empty implementation, hence we will not get any output.
It's a highly recommended to override run method. Otherwise, do not go for multithreading concept.

Example:

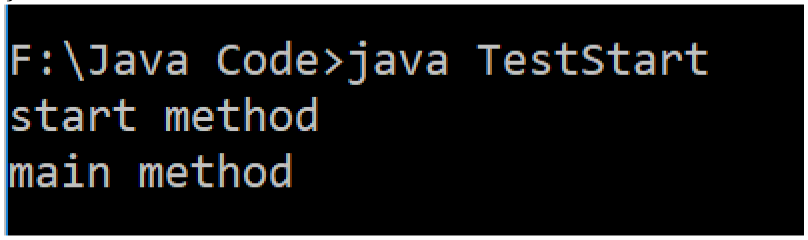
```
class MyThreadEx extends Thread
{
//not override run() method.

}
class NoOverride
{
	public static void main(String arg[])
	{
	 MyThreadEx t1=new MyThreadEx();

	t1.start();
	}
}
```



```
F:\Java Code>javac NoOverride.java

F:\Java Code>java NoOverride

F:\Java Code>
```

Overriding of start() method:

```
class MyThreadEx1 extends Thread
{
public void start()
{
System.out.println("start method");
}
public void run()
{
System.out.println("run method");
}
}

class TestStart
{
public static void main(String args[])
{
MyThreadEx1 t=new MyThreadEx1();
t.start();
System.out.println("main method");
}
}
```



```
F:\Java Code>java TestStart
start method
main method
```

Always get similar output if we run 1000 times.

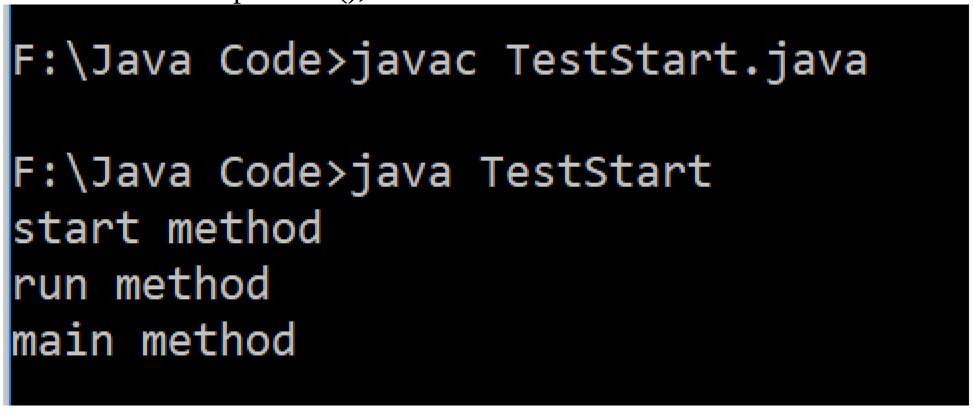
Reason:

If we override start method then our start method will be executed just like a normal method call and new thread will not be created. Total output will be produced by main thread. So it is recommended never override start method otherwise don't go for multithreading concept.

Let's do small change in the previous program.

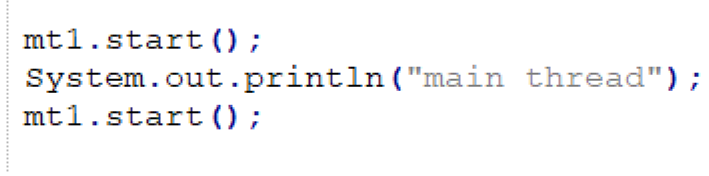
```
class MyThreadEx1 extends Thread
{
public void start()
{
        super start();
System.out.println("start method");
}
public void run()
{
System.out.println("run method");
}
}

class TestStart
{
public static void main(String args[])
{
MyThreadEx1 t=new MyThreadEx1();
t.start();
System.out.println("main method");
t.start();

}
}
```

We will get Mixed Output
Just because of super.start(),

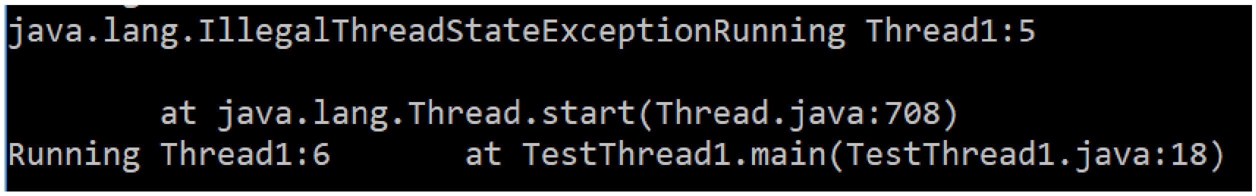


```
F:\Java Code>javac TestStart.java

F:\Java Code>java TestStart
start method
run method
main method
```

Restart the same thread.
After starting the thread if we are trying to restart the same thread then we will get run time exception: Illegal Thread state exception.



```
mt1.start();
System.out.println("main thread");
mt1.start();
```



```
java.lang.IllegalThreadStateExceptionRunning Thread1:5

        at java.lang.Thread.start(Thread.java:708)
Running Thread1:6       at TestThread1.main(TestThread1.java:18)
```