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 in two ways:
- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)
- Each process has its own address in memory i.e., each process allocates a separate memory area.
- Process is heavyweight.
- Cost of communication between the processes is high.
- Switching from one process to another requires time to save and load registers, memory maps, and updated lists.

2) Thread-based Multitasking (Multithreading)
- Threads share the same address space. **Example MSWORD typing-t1, autosave-t2, spellcheck-t3,page number -t4………….2048(RAM)**
- Thread is lightweight.
- Cost of communication between the threads 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, the smallest unit of processing. Multiprocessing and multithreading are both used to achieve multitasking.
But we use multithreading then multiprocessing because threads share a common memory area. They don't allocate separate memory areas, save memory, and context-switching between the threads takes less time than the process.

Advantages of Threading
- Better utilization of system resources, including the CPU.
- Reduces the computation time.
- Improves the performance of an application.

- Threads share the same address space, so it saves memory.
- Context switching between threads is usually less expensive than 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 the same time.
- You **can perform many operations together, so it saves time**.
- Threads are **independent,** so it doesn't affect other threads if exceptions occur in a single thread.

Thread States or Life cycle of a Thread
JVM controls the life cycle of the thread in java. 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 a new state; in this state, thread will not be executed and not share time with the processor.
2) Runnable/Ready: When the start() method is called on the thread object, the thread is in a runnable state. In this state, the thread executes and shares time with the processor.
3) Running: If the thread scheduler allocates a processor to a thread, it will go into the Running state OR A thread currently being executed by the CPU is in the running state.
**4) Non-Runnable (Blocked):** This is the state in which the thread is still alive but is currently not eligible to run. A running thread may go to a blocked state due to the following conditions.
•   wait() or sleep() method is called
•   The thread performs I/O operation.
When a blocked thread is unblocked, it goes to a runnable state and not to a running state.
5) Terminated: A thread is in the 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.)
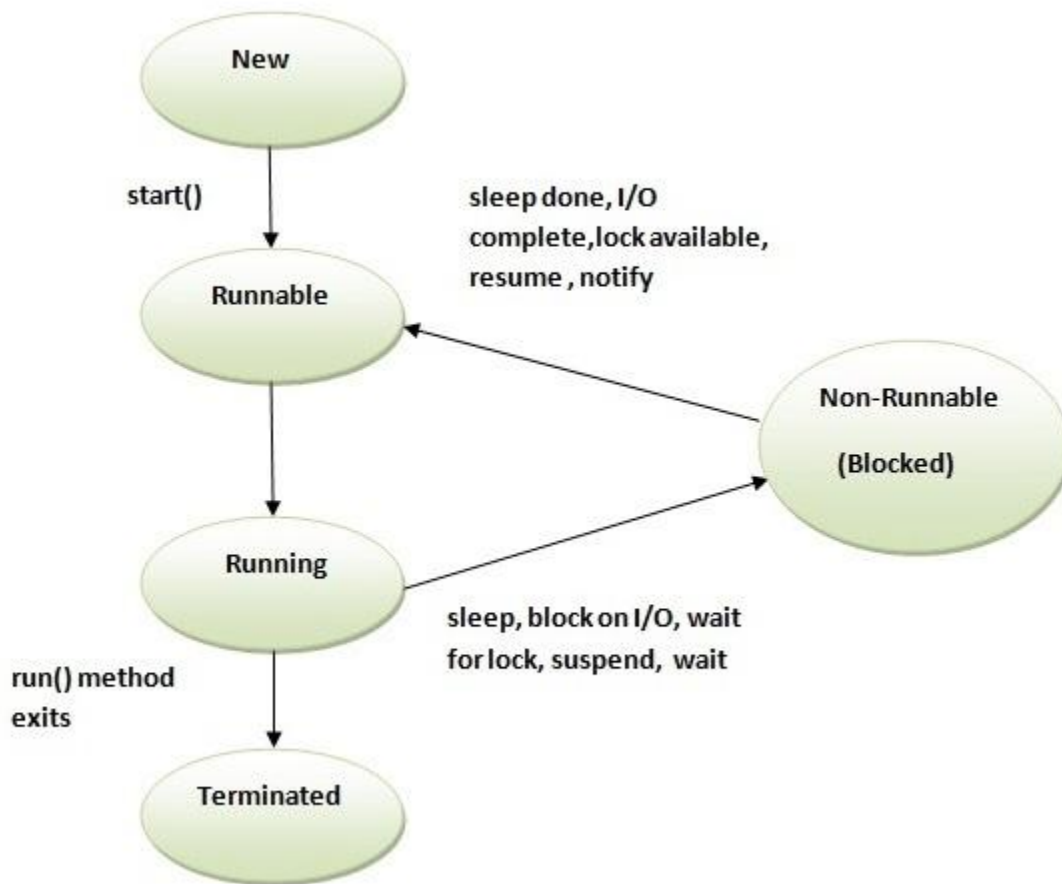
Note: According to sun microsystem, there are only 4 states in the thread life cycle in java new, runnable, non-runnable, and terminated. There is no running state.

CREATION OF THREADS
There are two ways to create a thread:
1. By extending the Thread class
2. By implementing a Runnable interface.

1. By extending the Thread class:

The Thread class provides 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 an 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 milliseconds): 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 milliseconds): waits for a thread to die for the specified milliseconds.

public int getPriority(): returns the priority of the thread.

public int setPriority(int priority): changes the thread's priority.

public String getName(): returns the name of the thread.

public void setName(String name): changes the thread's name.

public Thread currentThread(): returns the reference of the 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(deprecated).

public void resume(): is used to resume the suspended thread(deprecated).

public void stop(): is used to stop the thread(deprecated).

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

```java
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();
   }
   }
```

C:\Windows\System32\cmd.exe

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

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

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

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

```java
class TestThread
{
public static void main(String arg[])
        {

          MyThread1 mt1=new MyThread1();
          mt1.run();  //1-10
          MyThread2 mt2=new MyThread2();
          mt2.run();//11-20

        } }
```

1) run()← CPU –cpu exit

2) run()← CPU

```
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
```

```
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();  --run()
MyThread2 mt2=new MyThread2();
      mt2.start();  --run()
      } }
```

//will get mixed output.
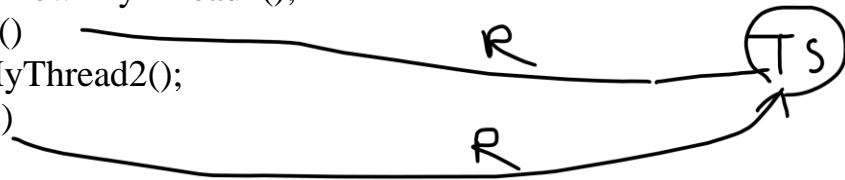
```
C:\Windows\System32\cmd.exe

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 a 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 of getting a similar output. It varies according to the system.

run() vs start()
t.run()
If we use the run() method instead of the start() method, then the run() method will be executed just like a normal method (called by the main method).

t.start()

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

Internal Definition of start()

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