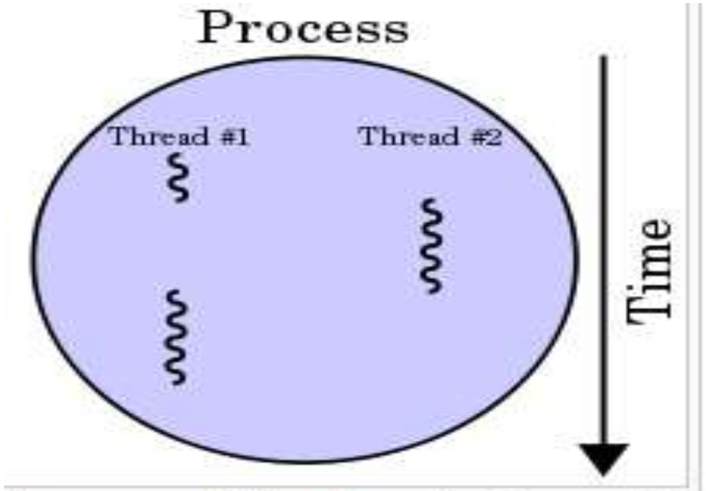
Thread Basics



A process with two threads of execution on a single processor

- In computer science, a **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler (typically as part of an operating system).

- *multithreaded programming language* which means we can develop multithreaded program using Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

- Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running in the computer. 235

Multi-threaded applications

- **Responsiveness**: Multi-threading has the ability for an application to remain responsive to input. In a single-threaded program, if the main execution thread blocks on a long-running task, the entire application can appear to freeze. By moving such long-running tasks to a *worker thread* that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background. On the other hand, in most cases multithreading is not the only way to keep a program responsive, with non-blocking I/O and/or Unix signals being available for gaining similar results

- **Faster Execution**: This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple or multi-core CPUs, or across a cluster of machines, because the threads of the program naturally lend themselves to truly concurrent execution.

- **Less Resource Intensive**: Using threads, an application can serve multiple clients concurrently using less resource than it would need when using multiple process copies of itself. For example, the Apache HTTP server, which uses a pool of listener and server threads for listening to incoming requests and processing these

- . **Better System Utilization**: Multi-threaded applications can also utilize the system better. For example, a file-system using multiple threads can achieve higher throughput and lower latency since data in faster mediums like the cache can be delivered earlier while waiting for a slower medium to retrieve the data.

- **Simplified Sharing and Communication**: Unlike processes, which require message passing or shared memory to perform inter-process communication, communication between threads is very simple. Threads automatically share the data, code and files and so, communication is vastly simplified.

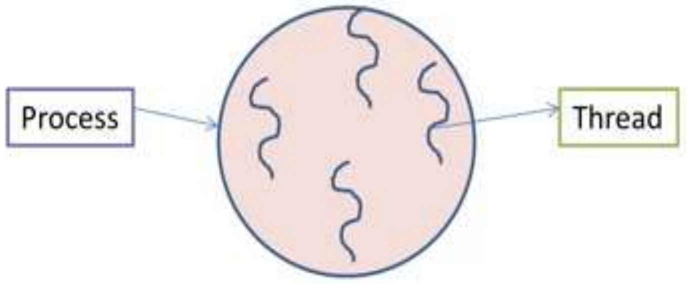
- **Parallelization**: Applications looking to utilize multi-core and multi-CPU systems can use multi-threading to split data and tasks into parallel sub-tasks and let the underlying architecture manage how the threads run, either concurrently on a single core or in parallel on multiple cores. GPU computing environments like CUDA and OpenCL use the multi-threading model where dozens to hundreds of threads run in parallel on a large number of cores.

- **Multi-threading has the following drawbacks:**
- **Synchronization**: Since threads share the same address space, the programmer must be careful to avoid race conditions and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require mutually exclusive operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified or read while in the process of being modified. Careless use of such primitives can lead to deadlocks.
- **Thread crashes Process**: An illegal operation performed by a thread crashes the entire process and so, one misbehaving thread can disrupt the processing of all the other threads in the application.

Process and Threads

A single application running in OS is a process.
A process can have multiple threads.

Spell checker in Word can be considered as a thread.

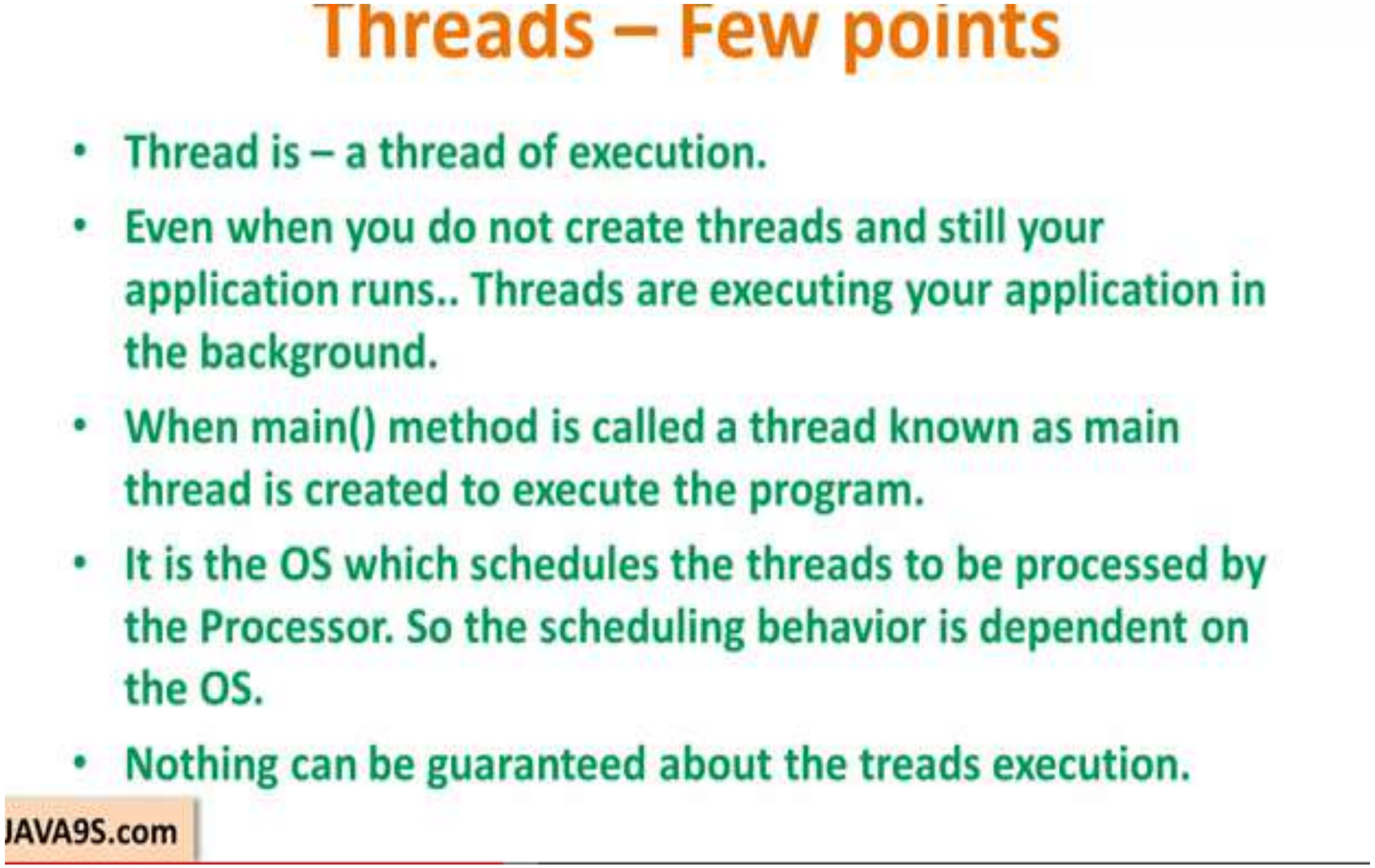


Java library for threading

- import java.util.Random;

Java Concurrency

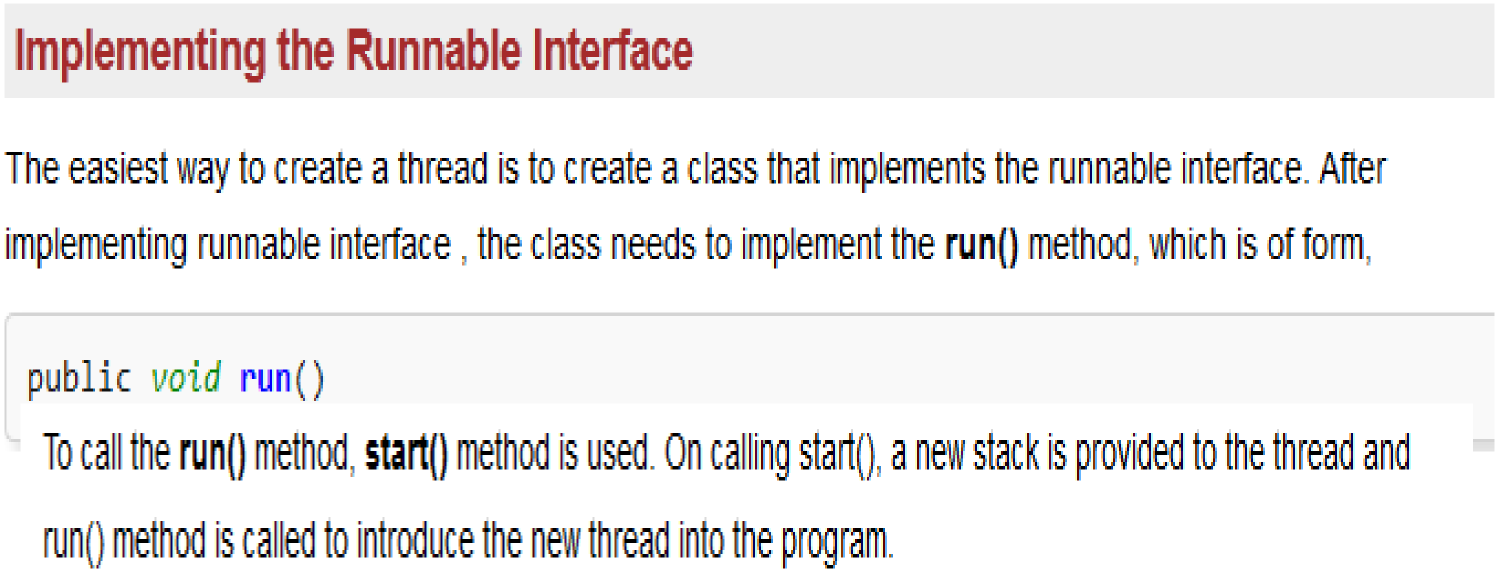
Java makes concurrency available to you through the language and APIs. Java programs can have multiple **threads of execution**, where each thread has its own method-call stack and program counter, allowing it to execute concurrently with other threads while sharing with them application-wide resources such as memory. This capability is called **multi-threading**.



Threads – Few points

- Thread is – a thread of execution.
- Even when you do not create threads and still your application runs.. Threads are executing your application in the background.
- When main() method is called a thread known as main thread is created to execute the program.
- It is the OS which schedules the threads to be processed by the Processor. So the scheduling behavior is dependent on the OS.
- Nothing can be guaranteed about the treads execution.

Two ways for Thread Creation



Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface , the class needs to implement the **run()** method, which is of form,

```
public void run()
```

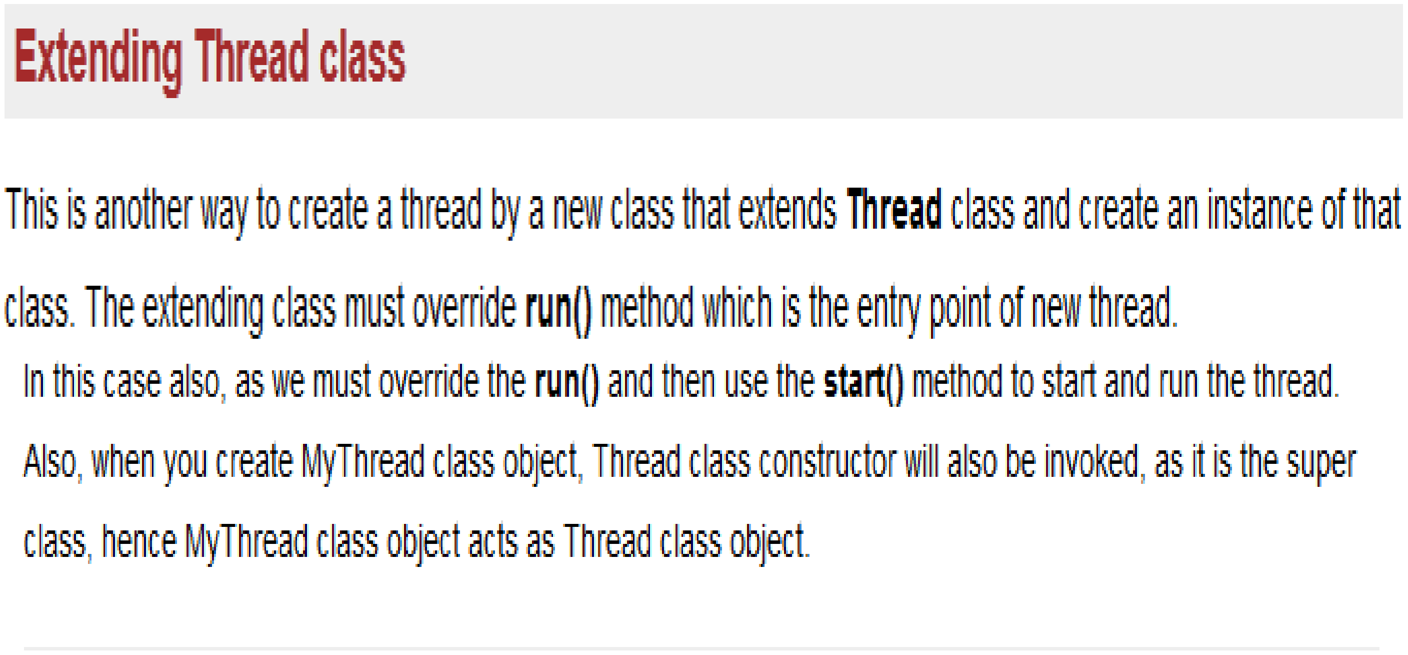
To call the **run()** method, **start()** method is used. On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.

Implementing thread using Runnable interface

```
public class thread1 implements Runnable{
    @Override
    public void run()
    {
        System.out.println("hello world");
    }
     public static void main(String args[]) {
        //(new Thread(new hello_thread())).start();
         Thread t=new Thread(new thread1());
        t.start();

    }
}
```

Extending Thread class



Extending Thread class

This is another way to create a thread by a new class that extends **Thread** class and create an instance of that class. The extending class must override **run()** method which is the entry point of new thread.

In this case also, as we must override the **run()** and then use the **start()** method to start and run the thread. Also, when you create MyThread class object, Thread class constructor will also be invoked, as it is the super class, hence MyThread class object acts as Thread class object.

Creating thread using Extending Thread class

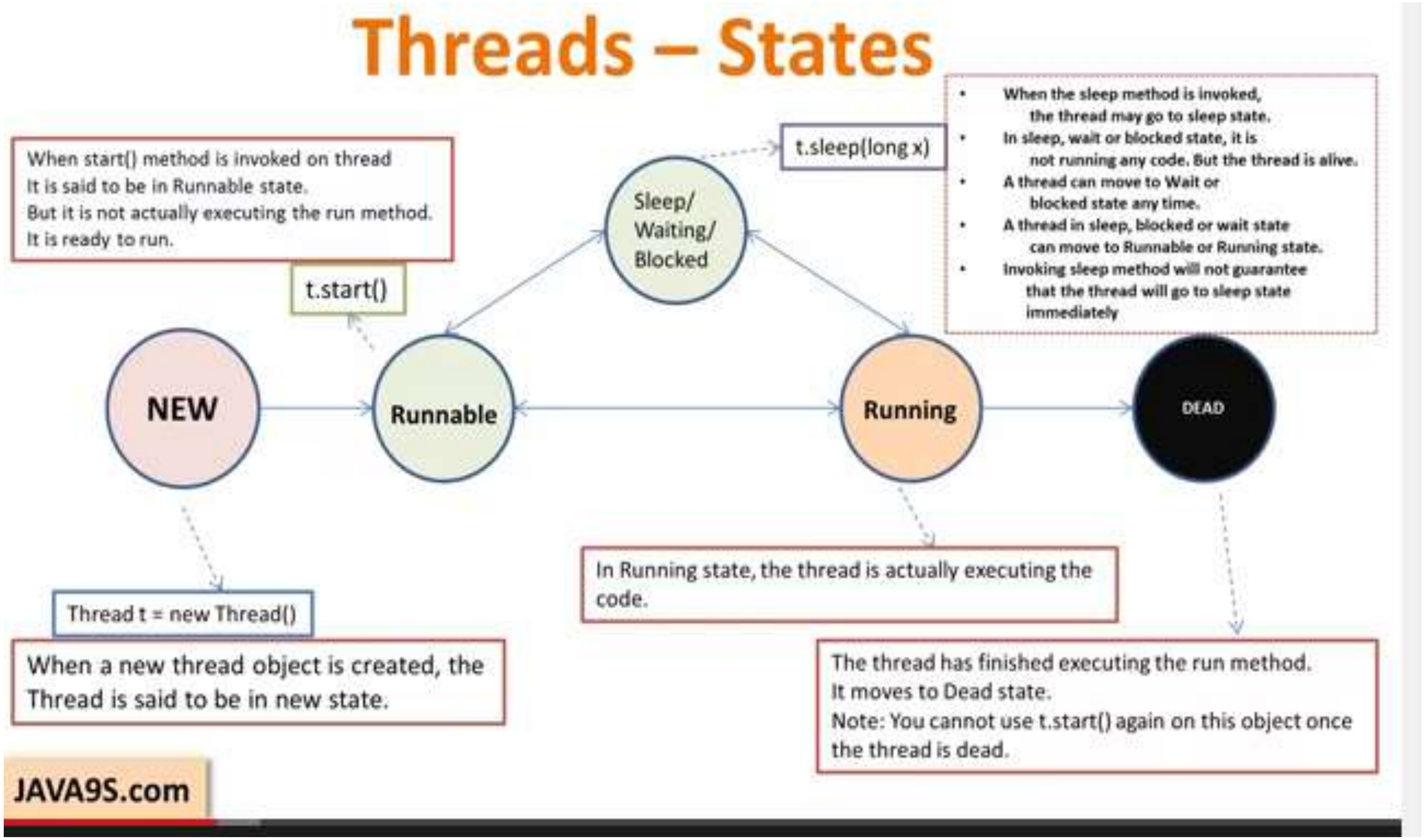
```
•   public class thread2 extends Thread{
•       public void run()
•       {
•           System.out.println("helloworld");
•       }
•        public static void main(String[] args)
•        {
•            thread2 t=new thread2();
•   t.start();
•        }
•
•   }
```

14.3.3 Thread Class versus Runnable Interface

It is a little confusing why there are two ways of doing the same thing in the threading API. It is important to understand the implication of using these two different approaches. By extending the thread class, the derived class itself is a thread object and it gains full control over the thread life cycle. Implementing the Runnable interface does not give developers any control over the thread itself, as it simply defines the unit of work that will be executed in a thread. Another important point is that when extending the Thread class, the derived class cannot extend any other base classes because Java only allows single inheritance. By implementing the Runnable interface, the class can still extend other base classes if necessary. To summarize, if the program needs a full control over the thread life cycle, extending the Thread class is a good choice, and if the program needs more flexibility of extending other base classes, implementing the Runnable interface would be preferable. If none of these is present, either of them is fine to use.

- **Can we start a thread twice?**
- No. After staring a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown.

Thread Life Cycle



When start() method is invoked on thread
It is said to be in Runnable state.
But it is not actually executing the run method.
It is ready to run.

- When the sleep method is invoked, the thread may go to sleep state.
- In sleep, wait or blocked state, it is not running any code. But the thread is alive.
- A thread can move to Wait or blocked state any time.
- A thread in sleep, blocked or wait state can move to Runnable or Running state.
- Invoking sleep method will not guarantee that the thread will go to sleep state immediately

When a new thread object is created, the Thread is said to be in new state.

In Running state, the thread is actually executing the code.

The thread has finished executing the run method.
It moves to Dead state.
Note: You cannot use t.start() again on this object once the thread is dead.

New *and* Runnable *States*

A new thread begins its life cycle in the *new* state. It remains in this state until the program starts the thread, which places it in the *runnable* state. A thread in the *runnable* state is considered to be executing its task.

Waiting *State*

Sometimes a *runnable* thread transitions to the *waiting* state while it waits for another thread to perform a task. A *waiting* thread transitions back to the *runnable* state only when another thread notifies it to continue executing.

Timed Waiting *State*

A *runnable* thread can enter the *timed waiting* state for a specified interval of time. It transitions back to the *runnable* state when that time interval expires or when the event it's waiting for occurs. *Timed waiting* and *waiting* threads cannot use a processor, even if one

Blocked *State*

A *runnable* thread transitions to the *blocked* state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes. For example, when a thread issues an input/output request, the operating system blocks the thread from executing until that I/O request completes—at that point, the *blocked* thread transitions to the *runnable* state, so it can resume execution. A *blocked* thread cannot use a processor, even if one is available.

Terminated *State*

A *runnable* thread enters the *terminated* state (sometimes called the *dead* state) when it successfully completes its task or otherwise terminates (perhaps due to an error). In the UML state diagram of Fig. 26.1, the *terminated* state is followed by the UML final state (the bull's-eye symbol) to indicate the end of the state transitions.

Thread - Priorities

- All the threads created in Java carry normal priority unless specified.
- Priorities can be specified from 1 to 10.
- The thread with highest priority will be given preference in execution. But no guarantee that it will be in the running state the moment it starts.
- The currently executing thread will have the highest priority when compared to the threads that are there in the pool.
- Thread scheduler decides on which threads have to be given chance.
- t.setPriority() can be used to set the priorities on a thread object.
- The Priority should be set before the threads start() method is invoked.
- Thread.MIN_PRIORITY, Thread.NORM_PRIORITY, Thread.MAX_PRIORITY can be used in the threads setPriority() method.

Major thread operations

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.

Multithreaded Program for calculator

```
public class calc_add extends Thread
{
   @Override
   public void run()
   {

      try
      {
         int sum;
      int a=2;
      int b=4;
      sum=a+b;
      System.out.println(sum);
         Thread.sleep(8000);
      }
      catch(Exception e)
            {
              System.out.println(e);
            }
   }
}
```

```
public class calc_mul extends Thread {
   public void run()
   {

      try
      {
         int mul;
      int a=2;
      int b=4;
      mul=a*b;
      System.out.print(mul);
         Thread.sleep(2000);
      }
      catch(Exception e)
            {
             System.out.println(e);
            }

}}
```

```
•    public class calc_min extends Thread{
•       public void run()
•       {
•
•          try
•          {
•             int minus;
•          int a=2;
•          int b=4;
•          minus=a-b;
•          System.out.print(minus);
•             Thread.sleep(2000);
•          }
•          catch(Exception e)
•                {
•                  System.out.println(e);
•                }
•
•    }}
•  public class calc_main {
•
•      public static void main(String[] args)
•      {
•          calc_add t1=new calc_add();
•          calc_mul t2=new calc_mul();
•          calc_min t3=new calc_min();
•          t1.start();
•          t2.start();
•          t3.start();
•      }
•  }
```