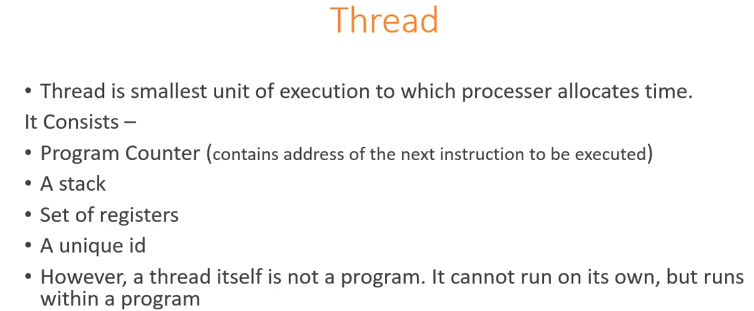
**Thread**

**Definition:** A thread is a single sequential flow of control within a program.

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



There are two ways to use a thread in java:

1. implement Runnable
2. extend Thread

*Runable* is better than extending *Thread*.

* **extends Thread:**

your thread creates unique object and associate with it

* **implements Runnable:**

it shares the *same* object to multiple threads

**Why 2 ways ?**

Extending Thread class will not give you an option to extend any other class. But if you implement Runnable interface you could extend other classes in your class..

So depending on your design requirement you could use either of the menthods.

**When to use what in thread implementation**

Both methods have different approaches. Implementing Runnable interface does not give any control over thread itself.And if we extends thread class then derived class can not extend any other base class.

So if user wants fully control over program then Extending of Thread class is better option and if user wants flexibility of extending other base classes then Implementing Runnable Interface is good option.

**Advantage of Runnable Thread**

Even if you implement Runnable interface you will need to create thread to let your task run as a thread. obvious advantages you get out implementing Runnable are

1. You have liberty to extend any other class
2. You can implement more interfaces
3. You can use you Runnable implementation in thread pools

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

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

**Run Method-** Entire functionality of Thread is written in Run function

|  |
| --- |
| 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 state

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

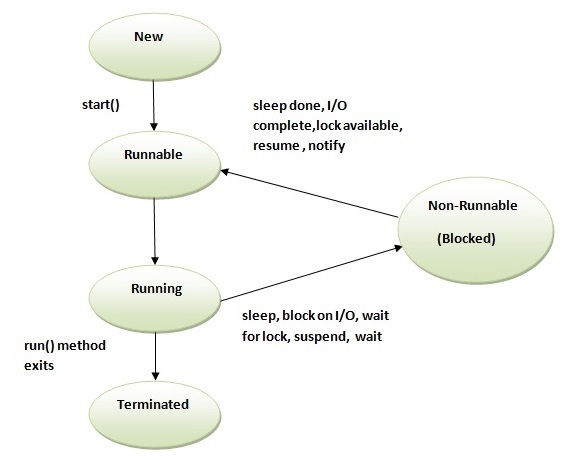
The thread is in running state if the thread scheduler has selected it.

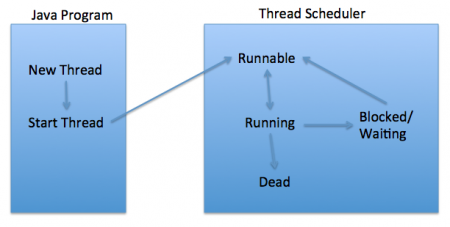
### 4) Non-Runnable (Blocked)

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.





When we create a thread it will be in ready or new state then moves to running state

When moved from new to running state it invokes start() method, start method internally called run method then it goes to running state after running state thread can go in 2 states either **Sleep or Death** (terminated) state Thread can go in sleep state in 3 ways

1. Sleep (Millisecond’s) method

2 suspend () method deprecated now

3. Wait () method

Sleep (1000)—running by given no of time ms then after given againvl b in active state

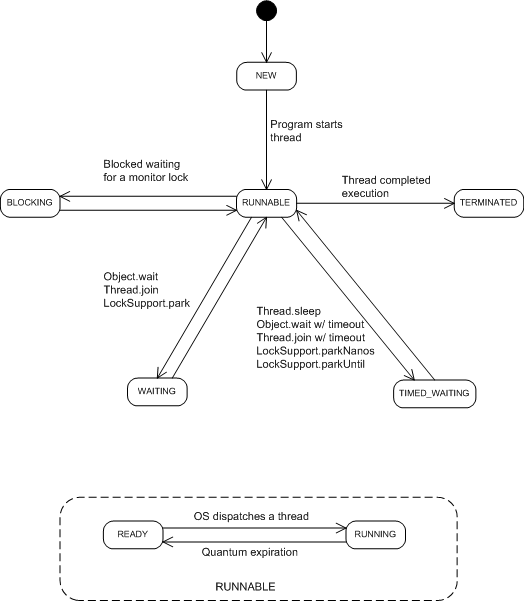
Suspend to active—done by resume () method**Deprecated**

Wait to active --Done by notifyall () or notify ();

Thread can go in Death or Stop State in 2 ways

1. Out of run method

2. Stop (); **Deprecated now**



**3 things used in thread**

1 thread class

2 thread group

3 runnable interface

Thread group-used if we want to manipulate n no of threads , if we want to put no of threads in sleep state , we can manage set of threads in one go, we can provide ceratain no of functionality we can do in one go

Join method()

Interrupt method ()

T1.

-

-

-

-

-

-T2.join() – Now t1 waits till T2 Not completed

-

-

-

-When t2 completed(Join method will not interrupt until t2 thread stops) or dies

- t1 continues to execution

-

-

-

But if we want to excute t1 again without stopping t2 then we cal Interrupt method in T2 Thread

T2

-

-

-

-

-

-T1.intrrupt() – Now t1 waits till T2 Not completed

-

T1 contuniues

-

-

## Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).Priority 10 is higher priority than priority 1, higher the priority it get CPU time first after that rest

1. join() If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.

Join : t1 and t2 are two threads , t2.join() is called then t1 enters into wait state until t2 completes execution. Then t1 will into runnable state then our specialist JVM thread scheduler will pick t1 based on criteria's.

1. **Yield() Method** pauses the currently executing thread temporarily for giving a chance to the remaining **waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution.** The yielded thread when it will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent.

Sleep: It blocks the execution of that particular thread for a given time.

isAlive()- to check thread is alive or dead

All threads th1,th2,th3 belongs to same Thread group tg1

th1.setpriorty(9); ---8

tg1.setmarpriorty(8);

th2.setpriorty(9); ----8 because max group is 8;

th3.setpriorty(7)----7;

**2 type’s thread we can create**

* 1. User Thread – when u create thread and start the run method that’s user thread
  2. Daemon thread – When you create Thread and before starting you call seDaemon(true) then you call Start it means you are creating Daemon thread. Child of Daemon is Daemon
  3. To check daemon or not call-- isDaemon() – it will return true or false

**Note:** Once you have started the thread you can not change user thread to daem

on and vice versa

**Difference in Daemon(like GC, clock) ad User Thread**

1.When all the user thread are died its JVM shuts down, JVM does not care about the Daemon threads

Daemon Process and threads are used for managing Resources(Like in JVM we have Garbage Collector and it’s a thread and its Daemon Thread which manages memory)

GC has lowest Priority 1 that’s why its executed in the end

Is it good practice to call Garbage Collector manually/explicitelly?

No, it is definitely not good practice.

You can use System.gc(). Note that this is not guaranteed to call the garbage collector - it only gives a hint to the system that it might be a good idea to do garbage collection. Because GS has lower priority always

It will not excute immediately because higher priority Thread will be executed firsth then will GC

JVM is Alive till User Threads are Alive

The daemon threads are typically used to perform services for user threads. The main() method of the application thread is a user thread. Threads created by a user thread are user thread. JVM doesn't terminates unless all the user thread terminate.

User can not kill Daemon thread it will Throw the Exception its done by JVM when all user thread has been killed then JVM kills

## 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)

### 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 process 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.

# Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.

**Process vs Thread**

**Threads** are used for small tasks, whereas **processes** are used for more 'heavyweight' tasks – basically the execution of applications. Another difference between a **thread** and a **process** is that **threads** within the same **process** share the same address space, whereas different **processes**

The thread is a subset of Process, in other words, one process can contain multiple threads. Two process runs on different memory space, but all threads share same memory space. Don't confuse this with stack memory, which is different for the different thread and used to store local data to that thread. For more detail see the answer

**11) What is thread-safety? is Vector a thread-safe class?** (Yes, see [details](http://javarevisited.blogspot.sg/2011/09/difference-vector-vs-arraylist-in-java.html))  
Thread-safety is a property of an object or code which guarantees that if executed or used by multiple threads in any manner e.g. read vs write it will behave as expected. For example, a thread-safe counter object will not miss any count if same instance of that counter is shared among multiple threads. Apparently, you can also divide collection classes in two category, thread-safe and non-thread-safe. Vector is indeed a thread-safe class and it achieves thread-safety by synchronizing methods which modify state of Vector, on the other hand, its counterpart ArrayList is not thread-safe.

### 9)Can we make the user thread as daemon thread if thread is started?

No, if you do so, it will throw IllegalThreadStateException

### What is the Java volatile keyword?

Essentially, volatile is used to indicate that a **variable's value will be modified by different**[**threads**](http://www.javamex.com/tutorials/threads/).

Declaring a volatile Java variable means:

* The value of this variable will **never be cached thread-locally**: all reads and writes will go straight to "main memory";
* Access to the variable **acts as though it is enclosed in a**[**synchronized block**](http://www.javamex.com/tutorials/synchronization_concurrency_synchronized1.shtml), synchronized on itself.