***Java MultiThreading***

* **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.

### 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.
  + **Note: At least one process is required for each thread.**
  + **Note: At a time one thread is executed only.**

|  |
| --- |
| 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.   1. [Life cycle of a thread](http://www.javatpoint.com/life-cycle-of-a-thread)    1. [New](http://www.javatpoint.com/life-cycle-of-a-thread#threadstatenew)    2. [Runnable](http://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunnable)    3. [Running](http://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunning)    4. [Non-Runnable (Blocked)](http://www.javatpoint.com/life-cycle-of-a-thread#threadstateblocked)    5. [Terminated](http://www.javatpoint.com/life-cycle-of-a-thread#threadstateterminated)   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:   1. New 2. Runnable 3. Running 4. Non-Runnable (Blocked) 5. Terminated   thread life cycle in java 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**.

### 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:

|  |
| --- |
| 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. |

### 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**. |

* NOTE : - If you **are not extending the Thread class**,your class object would **not be treated as a thread** object.**So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute**.
* ***Thread Scheduler in Java***
* Thread scheduler in java is the **part of the JVM** that decides **which thread should run**.
* There is **no guarantee that which runnable thread will be chosen to run by the thread scheduler**.
* Only one thread at a time can run in a single process.
* The **thread scheduler** mainly uses **preemptive** or **time slicing scheduling** to schedule the threads.
* **Difference between preemptive scheduling and time slicing**
  + Under **preemptive** scheduling, the **highest priority task executes** until it enters the **waiting** or **dead states** or a **higher priority task** comes into existence. Under **time slicing**, a task **executes for a predefined slice of time** and then reenters the pool of ready tasks. **The scheduler then determines which task should execute next**, based on priority and other factors.
* ***NOTE : - 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.
* ***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.
* ***Join method in Thread Class***
  + ***The join() method waits for a thread to die.***
  + It will **holds the Starting of** all the **other threads** till it **executes completely**.

**public** **class** OverLo **extends** Thread{

**public** **void** run() {

**int** i=0;

**while**(i<10){

// logic to print 1 to 5 on hold of 0.5 sec

}

**public** **static** **void** main(String[] args) {

OverLo o1 = **new** OverLo();

OverLo o2 = **new** OverLo();

OverLo o3 = **new** OverLo();

o1.start();

**try** {

***==> ==> o1.join();***

} **catch** (InterruptedException e) {}

o2.start();

o3.start();

}}

output :-

**1 2 3 4 5** 1 1 2 2 3 3 4 4 5 5

* *Thread Priority*

**3 constants defined in Thread class:**

|  |
| --- |
| 1. public static int MIN\_PRIORITY 2. public static int NORM\_PRIORITY 3. public static int MAX\_PRIORITY |

* To set the priority of Thread

TestMultiPriority1 m1=new TestMultiPriority1();

TestMultiPriority1 m2=new TestMultiPriority1();

**m1.setPriority(Thread.MIN\_PRIORITY);**

**m2.setPriority(Thread.MAX\_PRIORITY);**

m1.start();

m2.start();

Daemon Thread in Java

* Daemon thread in java is a **service provider thread** that **provides services to the user thread**. Its life **depends on the mercy of user threads** i.e. when all the user threads dies, **JVM terminates this thread** automatically.
* There are many java daemon threads running automatically e.g. **gc, finalizer** etc.

**IMP\_Notes\_DeamonThreads**

* It provides **services to user threads for background supporting tasks**. It has no role in life than to serve user threads.
* Its **life depends on user threads**.
* It is a **low priority** thread.
* JVM **terminates the daemon thread** if there is **no user thread**.
* If you want to **make a user thread as Daemon**, it **must not be started** otherwise it will **throw IllegalThreadStateException**.

### Methods for Java Daemon thread by Thread class

* The java.lang.Thread class provides two methods for java daemon thread.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public void setDaemon(boolean status) | is used to mark the current thread  as daemon thread or user thread. |
| 2) | public boolean isDaemon() | is used to check that current is daemon. |

Java Thread Pool

* **Java Thread pool** represents a **group of worker threads** that **are waiting for the job and reuse many times**.
* In case of thread pool, **a group of fixed size threads are created**. A thread from the thread pool is pulled out and **assigned a job by the service provider**. After **completion of the job, thread is contained in the thread pool again**.
* **Advantage of Java Thread Pool**
  + Better performance It **saves time because there is no need to create new thread**.
* **Real time usage**
  + It is **used in Servlet and JSP where container creates a thread pool to process the request**.