Feel free to contact santhosh_arena@yahoo.co.in





# Chapter 7 Threads

- Java is fundamentally multi-threaded.
- Every thread corresponds to an instance of java.lang.Thread class or a sub-class.
- A thread becomes **eligible to run**, when its start() method is called. Thread scheduler co-ordinates between the threads and allows them to run.
- When a thread begins execution, the scheduler calls its run method.

  Signature of run method - **public void run ()**

- When a thread returns from its run method (or stop method is called - deprecated in 1.2), its dead. It cannot be restarted, but its methods can be called. (it's just an object no more in a running state)
- If start is called again on a dead thread, **IllegalThreadStateException** is thrown.
- When a thread is in running state, it may move out of that state for various reasons. When it becomes eligible for execution again, thread scheduler allows it to run.
- There are two ways to implement threads.

  1. Extend Thread class

     - Create a new class, extending the Thread class.
     - Provide a public void run method, otherwise empty run in Thread class will be executed.
     - Create an instance of the new class.

- Call start method on the instance (don't call run - it will be executed on the same thread)

2. Implement Runnable interface

- Create a new class implementing the Runnable interface.
- Provide a public void run method.
- Create an instance of this class.
- Create a Thread, passing the instance as a target - new Thread(object)
- Target should implement Runnable, Thread class implements it, so it can be a target itself.
- Call the start method on the Thread.

- JVM creates one user thread for running a program. This thread is called main thread. The main method of the class is called from the main thread. It dies when the main method ends. If other user threads have been spawned from the main thread, program keeps running even if main thread dies. Basically a program runs until all the user threads (non-daemon threads) are dead.
- A thread can be designated as a daemon thread by calling **setDaemon(boolean)** method. This method should be called before the thread is started, otherwise IllegalThreadStateException will be thrown.
- A thread spawned by a daemon thread is a daemon thread.
- Threads have priorities. Thread class have constants MAX_PRIORITY (10), MIN_PRIORITY (1), NORM_PRIORITY (5)
- A newly created thread gets its priority from the creating thread. Normally it'll be NORM_PRIORITY.
- **getPriority** and **setPriority** are the methods to deal with priority of threads.
- Java leaves the implementation of thread scheduling to JVM developers. Two types of scheduling can be done.

1. Pre-emptive Scheduling.

Ways for a thread to leave running state -

- It can cease to be ready to execute ( by calling a blocking i/o method)
- It can get pre-empted by a high-priority thread, which becomes ready to execute.

- It can explicitly call a thread-scheduling method such as wait or suspend.
- Solaris JVM's are pre-emptive.
- Windows JVM's were pre-emptive until Java 1.0.2

2. Time-sliced or Round Robin Scheduling

- A thread is only allowed to execute for a certain amount of time. After that, it has to contend for the CPU (virtual CPU, JVM) time with other threads.
- This prevents a high-priority thread mono-policing the CPU.
- The drawback with this scheduling is - it creates a non-deterministic system - at any point in time, you cannot tell which thread is running and how long it may continue to run.
- Macintosh JVM's
- Windows JVM's after Java 1.0.2

- Different states of a thread:

1. Yielding

- Yield is a static method. Operates on current thread.
- Moves the thread from running to ready state.
- If there are no threads in ready state, the yielded thread may continue execution, otherwise it may have to compete with the other threads to run.
- Run the threads that are doing time-consuming operations with a low priority and call yield periodically from those threads to avoid those threads locking up the CPU.

2. Sleeping

- Sleep is also a static method.
- Sleeps for a certain amount of time. (passing time without doing anything and w/o using CPU)
- Two overloaded versions - one with milliseconds, one with milliseconds and nanoseconds.
- Throws an **InterruptedException**. (must be caught)

- After the time expires, the sleeping thread goes to ready state. It may not execute immediately after the time expires. If there are other threads in ready state, it may have to compete with those threads to run. The correct statement is the sleeping thread would execute *some time after* the specified time period has elapsed.

- If interrupt method is invoked on a sleeping thread, the thread moves to ready state. The next time it begins running, it executes the InterruptedException handler.

### 3. Suspending

- **Suspend** and **resume** are instance methods and are **deprecated** in 1.2

- A thread that receives a suspend call, goes to suspended state and stays there until it receives a resume call on it.

- A thread can suspend it itself, or another thread can suspend it.

- But, a thread can be resumed only by another thread.

- Calling resume on a thread that is not suspended has no effect.

- Compiler won't warn you if suspend and resume are successive statements, although the thread may not be able to be restarted.

### 4. Blocking

- Methods that are performing I/O have to wait for some occurrence in the outside world to happen before they can proceed. This behavior is blocking.

- If a method needs to wait an indeterminable amount of time until some I/O takes place, then the thread should graciously step out of the CPU. All Java I/O methods behave this way.

- A thread can also become blocked, if it failed to acquire the lock of a monitor.

### 5. Waiting

- **wait**, **notify** and **notifyAll** methods are not called on Thread, they're **called on Object.** Because the object is the one which controls the threads in this case. It asks the threads to wait and then notifies when its state changes. It's called a **monitor**.

- Wait puts an executing thread into waiting state.(to the monitor's waiting pool)

- Notify moves one thread in the monitor's waiting pool to ready state. We cannot control which thread is being notified. notifyAll is recommended.

- NotifyAll moves all threads in the monitor's waiting pool to ready.

- These methods can only be called from synchronized code, or an **IllegalMonitorStateException** will be thrown. In other words, only the threads that obtained the object's lock can call these methods.