What are Java Threads?

A thread is a:

- Facility to allow multiple activities within a single process
- Referred as lightweight process
- A thread is a series of executed statements
- Each thread has its own program counter, stack and local variables
- A thread is a nested sequence of method calls
- Its shares memory, files and per-process state

Whats the need of a thread or why we use Threads?

- To perform asynchronous or background processing
- Increases the responsiveness of GUI applications
- Take advantage of multiprocessor systems
- Simplify program logic when there are multiple independent entities

What happens when a thread is invoked?

When a thread is invoked, there will be two paths of execution. One path will execute the thread and the other path will follow the statement after the thread invocation. There will be a separate stack and memory space for each thread.

Risk Factor

- Proper co-ordination is required between threads accessing common variables [use of synchronized and volatile] for consistence view of data
- overuse of java threads can be hazardous to program's performance and its maintainability.

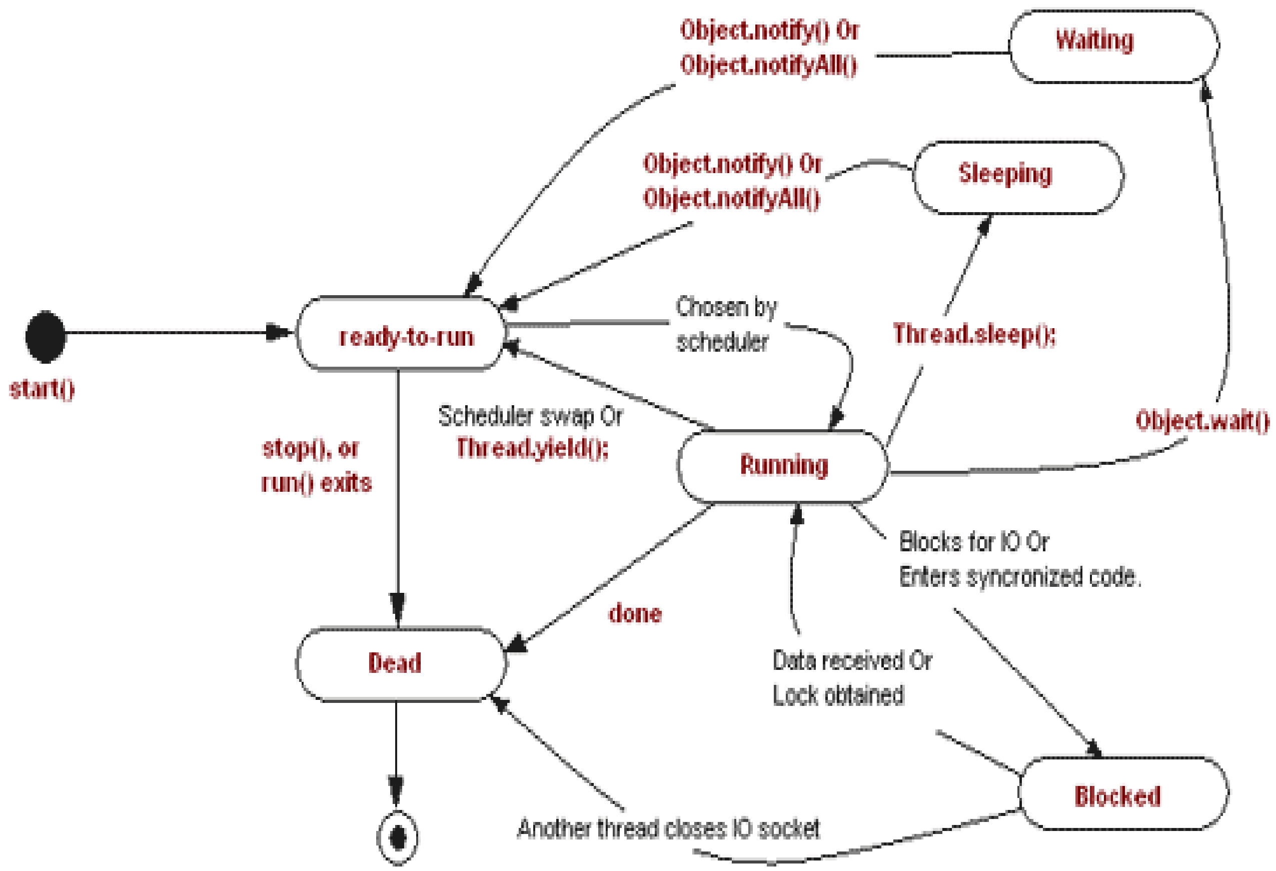
1. Thread Life cycle in Java

- The start method creates the system resources, necessary to run the thread, schedules the thread to run, and calls the thread's run method.

- A thread becomes "Not Runnable" when one of these events occurs:
    - If sleep method is invoked.
    - The thread calls the wait method.
    - The thread is blocking on I/O.

- A thread dies naturally when the run method exits.

Below diagram clearly depicts the various phases of thread life cycle in java.



2. Thread Scheduling

- Execution of multiple threads on a single CPU, in some order, is called scheduling.
- In general, the runnable thread with the highest priority is active (running)
- Java is priority-preemptive
  - If a high-priority thread wakes up, and a low-priority thread is running
  - Then the high-priority thread gets to run immediately
- Allows on-demand processing
- Efficient use of CPU

2.1 Types of scheduling

- Waiting and Notifying
  - Waiting [wait()] and notifying [notify(), notifyAll()] provides means of communication between threads that synchronize on the same object.
- wait(): when wait() method is invoked on an object, the thread executing that code gives up its lock on the object immediately and moves the thread to the wait state.

- notify(): This wakes up threads that called wait() on the same object and moves the thread to ready state.
- notifyAll(): This wakes up all the threads that called wait() on the same object.
- Running and Yielding
    - Yield() is used to give the other threads of the same priority a chance to execute i.e. causes current running thread to move to runnable state.
- Sleeping and Waking up
    - nSleep() is used to pause a thread for a specified period of time i.e. moves the current running thread to Sleep state for a specified amount of time, before moving it to runnable state. Thread.sleep(no. of milliseconds);

2.2 Thread Priority

- When a Java thread is created, it inherits its priority from the thread that created it.
- You can modify a thread's priority at any time after its creation using the setPriority method.
- Thread priorities are integers ranging between MIN_PRIORITY (1) and MAX_PRIORITY (10) . The higher the integer, the higher the priority.Normally the thread priority will be 5.

2.3 isAlive() and join() methods

- isAlive() method is used to determine if a thread is still alive. It is the best way to determine if a thread has been started but has not yet completed its run() method. **final boolean isAlive();**
- The nonstatic join() method of class Thread lets one thread "join onto the end" of another thread. This method waits until the thread on which it is called terminates. **final void join();**

3. Blocking Threads

- When reading from a stream, if input is not available, the thread will block
- Thread is suspended ("blocked") until I/O is available
- Allows other threads to automatically activate
- When I/O available, thread wakes back up again
    - Becomes "runnable" i.e. gets into ready state

4. Grouping of threads

- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- To put a new thread in a thread group the group must
- be explicitly specified when the thread is created
    - - public Thread(ThreadGroup group, Runnable runnable)
    - - public Thread(ThreadGroup group, String name)
    - - public Thread(ThreadGroup group, Runnable runnable, String name)
- A thread can not be moved to a new group after the thread has been created.
- When a Java application first starts up, the Java runtime system creates a ThreadGroup named main.
- Java thread groups are implemented by the java.lang.ThreadGroup class.

Threads in Java

Java threads facility and API is deceptively simple:
Every java program creates at least one thread [main() thread]. Additional threads are created through the Thread constructor or by instantiating classes that extend the Thread class.

Thread creation in Java

Thread implementation in java can be achieved in two ways:

1. Extending the java.lang.Thread class
2. Implementing the java.lang.Runnable Interface

Note: The Thread and Runnable are available in the   java.lang.* package

1) By extending thread class

- The class should extend Java Thread class.
- The class should override the run() method.
- The functionality that is expected by the Thread to be executed is written in the run() method.

void start(): Creates a new thread and makes it runnable.
void run(): The new thread begins its life inside this method.

Example:

```
public class MyThread extends Thread {

   public void run(){
    System.out.println("thread is running...");
  }

   public static void main(String[] args) {
     MyThread obj = new MyThread();
     obj.start();
}
```

2) By Implementing Runnable interface

- The class should implement the Runnable interface
- The class should implement the run() method in the Runnable interface
- The functionality that is expected by the Thread to be executed is put in the run() method

Example:

```
public class MyThread implements Runnable {

   public void run(){
     System.out.println("thread is running..");
   }

   public static void main(String[] args) {
     Thread t = new Thread(new MyThread());
     t.start();
}
```

Extends Thread class vs Implements Runnable Interface?

- Extending the Thread class will make your class unable to extend other classes, because of the single inheritance feature in JAVA. However, this will give you a simpler code structure. If you implement Runnable, you can gain better object-oriented design and consistency and also avoid the single inheritance problems.
- If you just want to achieve basic functionality of a thread you can simply implement Runnable interface and override run() method. But if you want to do something serious with thread object as it has other methods like suspend(), resume(), ..etc which are not available in Runnable interface then you may prefer to extend the Thread class.