Threads

Thread Life Cycle

* **New** - When we create a new Thread object using new operator, thread state is New Thread. At this point, thread is not alive.
* **Runnable** - When we call start() function on Thread object, it’s state is changed to Runnable. The control is given to Thread scheduler to finish it’s execution.
* **Running** - When thread is executing, it’s state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it’s state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.
* **Blocked/Waiting** - A thread can be waiting for other thread to finish using thread join or it can be waiting for some resources to available. Once the thread wait state is over, it’s state is changed to Runnable and it’s moved back to runnable thread pool.
* **Dead** - Once the thread finished executing, it’s state is changed to Dead and it’s considered to be not alive.

How to avoid deadlock in java

* **Avoid Nested Locks**: This is the most common reason for deadlocks, avoid locking another resource if you already hold one. It’s almost impossible to get deadlock situation if you are working with only one object lock.
* **Lock Only What is Required**: You should acquire lock only on the resources you have to work on, for example in above program I am locking the complete Object resource but if we are only interested in one of it’s fields, then we should lock only that specific field not complete object.
* **Avoid waiting indefinitely**: You can get deadlock if two threads are waiting for each other to finish indefinitely using thread join. If your thread has to wait for another thread to finish, it’s always best to use join with maximum time you want to wait for thread to finish.

Java Callable

* Java Callable interface use Generic to define the return type of Object. ***Executors*** class provide useful methods to execute Java Callable in a thread pool. Since callable tasks run in parallel, we have to wait for the returned Object.

Java Future

* Java Callable tasks return **java.util.concurrent.Future** object. Using Java Future object, we can find out the status of the Callable task and get the returned Object. It provides ***get()*** method that can wait for the Callable to finish and then return the result. Java Future provides ***cancel()*** method to cancel the associated Callable task. There is an overloaded version of ***get()*** method where we can specify the time to wait for the result, it’s useful to avoid current thread getting blocked for longer time. There are ***isDone()*** and ***isCancelled()*** methods to find out the current status of associated Callable task.