Multithreading

Multithreading is a Java feature that allows **concurrent execution of two or more parts of a program** for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

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

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

```
class A extends Thread {
   public void run()
   {
     int sum=0;
      try {
         for(int i=1;i<=1000;i++){
          sum = sum+i;
         System.out.println(sum);
Thread.sleep(1000);
         }
      }
      catch (Exception e) {
         System.out.println("Exception is caught");
      }
   }
}

class B extends Thread {
```

```
   public void run()
   {

      try {
         for(int i=1000;i>=1;i=i-5){
         System.out.println(i);
Thread.sleep(1000);
         }
      }
      catch (Exception e) {
         System.out.println("Exception is caught");
      }
   }
}

public class MultiThreadDemo {
   public static void main(String[] args)
   {

         A obj1 = new A();
         B obj2 = new B();

         obj1.start();
        obj2.start();


   }
}
```

Thread creation by implementing the Runnable Interface

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

```
class ThreadA implements Runnable{
    public void run()
    {
       int sum=0;
```

```
        try {
            for(int i=1;i<=1000;i++){
           sum = sum+i;
            System.out.println(sum);
Thread.sleep(1000);
            }
        }
        catch (Exception e) {
            System.out.println("Exception is caught");
        }
    }
}

class ThreadB implements Runnable{
    public void run()
    {

        try {
            for(int i=1000;i>=1;i=i-5){
            System.out.println(i);
Thread.sleep(1000);
            }
        }
        catch (Exception e) {
            System.out.println("Exception is caught");
        }
    }
}


public class MultiThreadDemo1 {
    public static void main(String[] args)
    {

          ThreadA obj1 = new ThreadA();
                    ThreadB obj2 = new ThreadB();

                    obj1.run();
                    obj2.run();



    }
}
```

Life cycle of a Thread

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

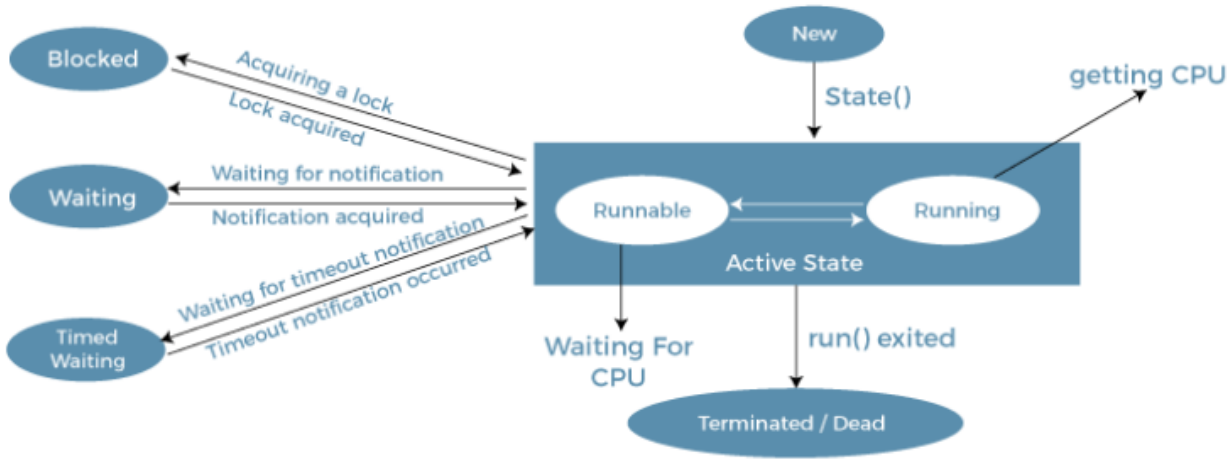
Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting

state.

Timed Waiting: Sometimes, waiting for leads to starvation.A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.



Life Cycle of a Thread

Synchronization

Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
class Table {
	 void printTable(int n) {
		for (int i = 1; i <= 20; i++) {
			System.out.println(n * i);
			try {
				Thread.sleep(400);
			} catch (Exception e) {
				System.out.println(e);
			}
		}

	}
}

class MyThread1 extends Thread {
	Table t;

	MyThread1(Table t) {
		this.t = t;
	}

	public void run() {
		t.printTable(2);
	}

}

class MyThread2 extends Thread {
	Table t;
```

```
	MyThread2(Table t) {
		this.t = t;
	}

	public void run() {
		t.printTable(5);
	}
}

public class SynchronizationExample {
	public static void main(String args[]) {
		Table obj = new Table();
		MyThread1 t1 = new MyThread1(obj);
		MyThread2 t2 = new MyThread2(obj);
		t1.start();
		t2.start();
	}
}
```