Threading

Multithreading in java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking

Advantages

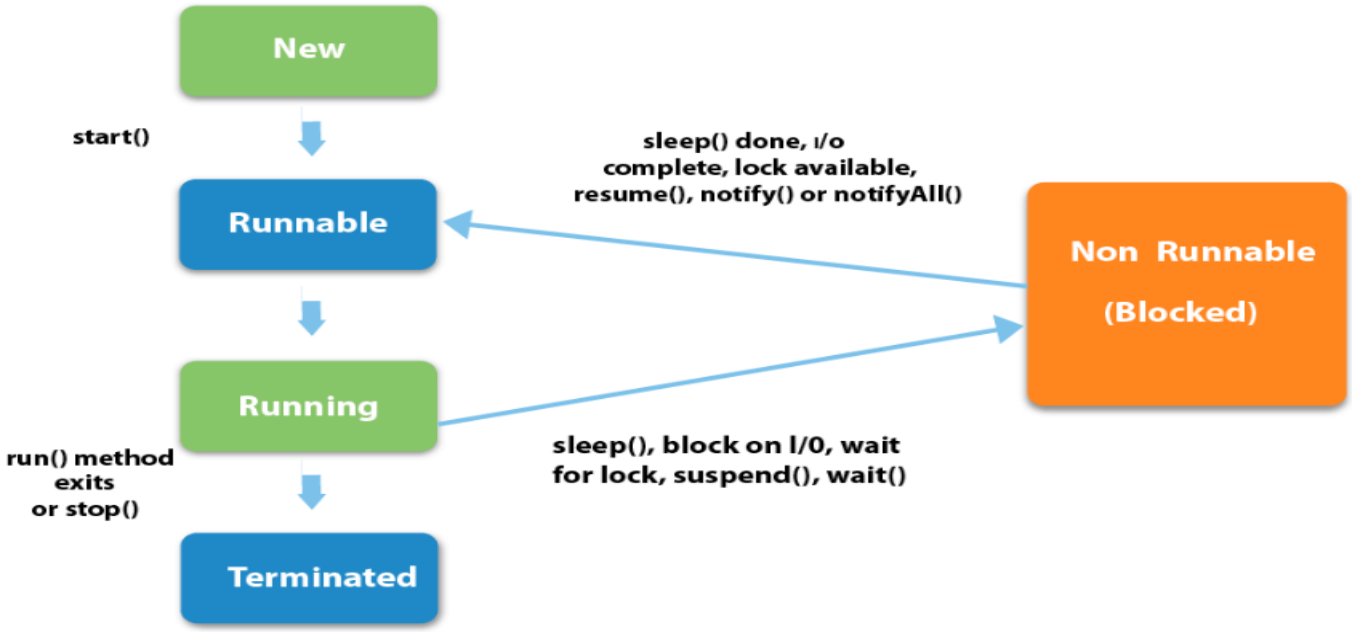
1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

A thread can have one of the following states during its lifetime:

1. **New:** In this state, a Thread class object is created using a new operator, but the thread is not alive. Thread doesn't start until we call the start() method.
2. **Runnable:** In this state, the thread is ready to run after calling the start() method. However, the thread is not yet selected by the thread scheduler.
3. **Running:** In this state, the thread scheduler picks the thread from the ready state, and the thread is running.
4. **Waiting/Blocked:** In this state, a thread is not running but still alive, or it is waiting for the other thread to finish.
5. **Dead/Terminated:** A thread is in terminated or dead state when the run() method exits.



How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

By Extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

By Implementing Runnable Interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Commonly used methods of Thread class:

public void run(): is used to perform action for a thread.

public void start(): starts the execution of the thread.JVM calls the run() method on the thread.

public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily

cease execution) for the specified number of milliseconds.

public final void join(): This java thread join method puts the current thread on wait until the thread on which it's called dead. If the thread is interrupted, it throws InterruptedException.

```
1.  class TestJoinMethod2 extends Thread{
2.   public void run(){
3.    for(int i=1;i<=5;i++){
4.     try{
5.      Thread.sleep(500);
6.     }catch(Exception e){System.out.println(e);}
7.    System.out.println(i);
8.    }
9.   }
10. public static void main(String args[]){
11. TestJoinMethod2 t1=new TestJoinMethod2();
12. TestJoinMethod2 t2=new TestJoinMethod2();
13. TestJoinMethod2 t3=new TestJoinMethod2();
14. t1.start();
15. try{
16.  t1.join();
17. }catch(Exception e){System.out.println(e);}
18.
19. t2.start();
20. t3.start();
21. }
22. }
```

public int getPriority(): returns the priority of the thread.

public int setPriority(int priority): changes the priority of the thread.

public String getName(): returns the name of the thread.

public void setName(String name): changes the name of the thread.

public Thread currentThread(): returns the reference of currently executing thread.

Java Concurrency issues and Thread Synchronization

Consider the following `Counter` class which contains an `increment()` method that increments the count by one, each time it is invoked -

```
class Counter {

    int count = 0;

    public void increment() {

        count = count + 1;

    }

    public int getCount() {

        return count;

    }

}
```

Now, Let's assume that several threads try to increment the count by calling the `increment()` method simultaneously –

```
class MyThread extends Thread
{
    // Reference variable of type counter.
    Counter counter;

    Train(Counter counter)
    {
        this.counter = counter;
    }


    @Override
    public void run()
    {
       For(int i=0;i<100;i++)
      {
        Counter.increment();
      }
    }
}
```

```
class GFG
{
    public static void main(String[] args)
    {
        Counter obj = new Counter();

        // we are creating two threads which share
        // same Object.
        MyThread t1 = new MyThread(obj);
        MyThread t2 = new MyThread(obj);

        // both threads start executing .
        t1.start();
        t2.start();
    }
}
```

What do you think the result of the above program will be? Will the final count be 100 because we're calling increment 100 times?

When multiple threads try to read and write a shared variable concurrently, and these read and write operations overlap in execution, then the final outcome depends on the order in which the reads and writes take place, which is unpredictable. This phenomenon is called Race condition.

Thread interference errors can be avoided by synchronizing access to shared variables. We'll learn about synchronization in the next section.

Synchronized Methods

```
class SynchronizedCounter {

    private int count = 0;


    // Synchronized Method

    public synchronized void increment() {

        count = count + 1;

    }


    public int getCount() {

        return count;

    }
```

```
}
```

We use Java's `synchronized` keyword on `increment()` method to prevent multiple threads from accessing it concurrently.

The *wait()* Method

Simply put, when we call *wait()* – this forces the current thread to wait until some other thread invokes *notify()* or *notifyAll()* on the same object.

notify() and *notifyAll()*

The *notify()* method is used for waking up threads that are waiting for an access to this object's monitor.

There are two ways of notifying waiting threads.

notify()

For all threads waiting on this object's monitor (by using any one of the *wait()* method), the method *notify()* notifies any one of them to wake up arbitrarily. The choice of exactly which thread to wake is non-deterministic and depends upon the implementation.

notifyAll()

This method simply wakes all threads that are waiting on this object's monitor.

The awakened threads will complete in the usual manner – like any other thread.

```
public class Data {

   private String packet;



   // True if receiver should wait

   // False if sender should wait

   private boolean transfer = true;

   public synchronized void send(String packet) {

      while (!transfer) {

         try {
```

```
            wait();
        } catch (InterruptedException e)  {
            Thread.currentThread().interrupt();
            Log.error("Thread interrupted", e);
        }
      }
      transfer = false;
     this.packet = packet;
      notifyAll();
   }
   public synchronized String receive() {
      while (transfer) {
        try {
            wait();
        } catch (InterruptedException e)  {
            Thread.currentThread().interrupt();
            Log.error("Thread interrupted", e);
        }
      }
      transfer = true;
      notifyAll();
      return packet;
   }
}
```