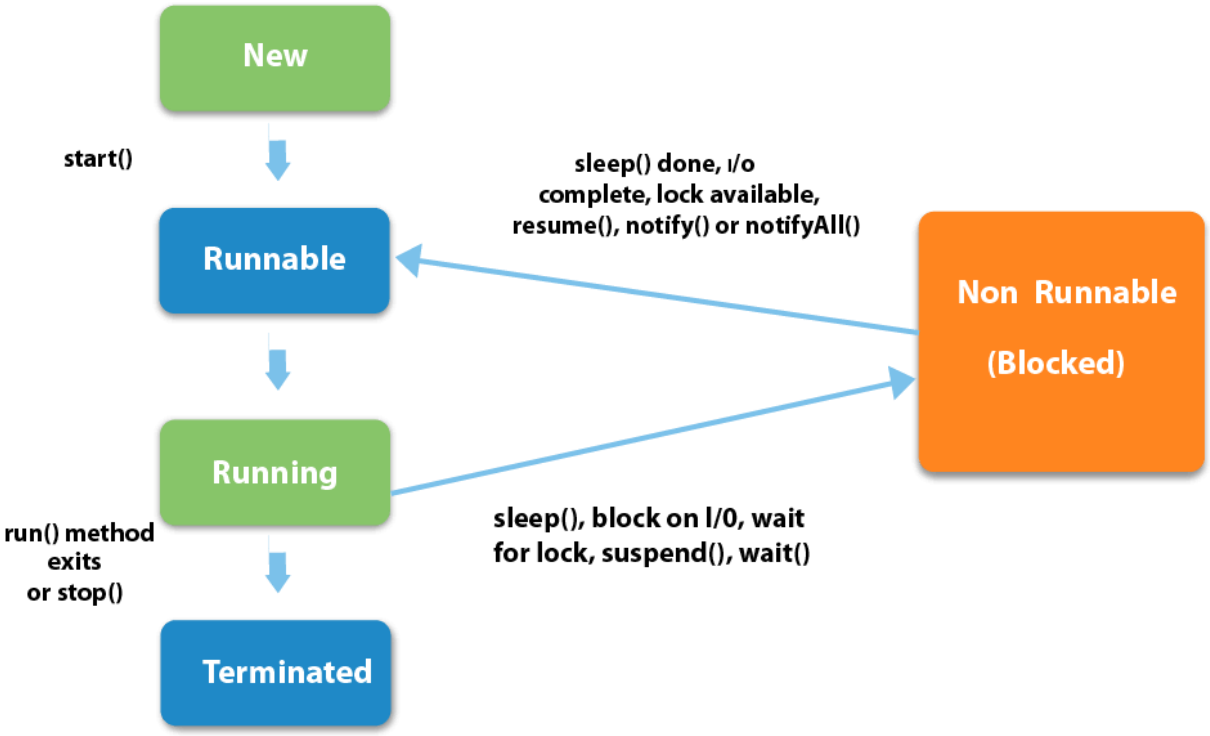
Multithreading



How to create thread

1. 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();
         } }
```

2. 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();
 }
}
```

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses **preemptive** or **time slicing** scheduling to schedule the threads

---

Sleep()

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long miliseconds)throws **InterruptedException**
- public static void sleep(long miliseconds, int nanos)throws **InterruptedException**

Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an ***IllegalThreadStateException*** is thrown. In such case, thread will run once but for second time, it will throw exception.

What if we call run() method directly instead start() method?

Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

join()

- public void join()throws **InterruptedException**
- public void join(long milliseconds)throws **InterruptedException**

it causes the currently running threads to stop executing until the thread it joins with completes its task.

currentThread() , getName(),setName(String) and getId() method:

The currentThread() method returns a reference to the currently executing thread object.

Priority of a Thread (Thread Priority) using setPriority(int Priority)

thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

1. **public static int MIN_PRIORITY = 1**
2. **public static int NORM_PRIORITY = 5**
3. **public static int MAX_PRIORITY = 10**

Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.
- JVM terminates the daemon thread if there is no user thread.
- Daemon threads Examples: gc, finalizer etc

1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

```
class TestDaemonThread2 extends Thread{
 public void run(){
  System.out.println("Name: "+Thread.currentThread().getName());
  System.out.println("Daemon: "+Thread.currentThread().isDaemon());
 }

 public static void main(String[] args){
  TestDaemonThread2 t1=new TestDaemonThread2();
  TestDaemonThread2 t2=new TestDaemonThread2();
        /*t1.start();
    t1.setDaemon(true);//will throw exception here  */
    /*t1.setDaemon(true);//should be demon before starting
    t1.start();*/
    t2.start();
 }
}
```

Java Thread Pool

Java Thread pool represents a group of worker threads that are waiting for the job and reuse many times.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

Better performance It saves time because there is no need to create new thread.

It is used in Servlet and JSP where container creates a thread pool to process the request.

java thread pool using ExecutorService and Executors

```
public class TestThreadPool {
    public static void main(String[] args) {
      ExecutorService executor = Executors.newFixedThreadPool(5); //creating a pool of 5
threads
      for (int i = 0; i < 10; i++) {
         Runnable worker = new WorkerThread("" + i);
         executor.execute(worker);//calling execute method of ExecutorService
       }
      executor.shutdown();
      while (!executor.isTerminated()) {   }

      System.out.println("Finished all threads");
   }
 }
```

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Note: Now suspend(), resume() and stop() methods are deprecated.

```
ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

      Thread t1 = new Thread(tg1, runnable,"one");
      t1.start();
      Thread t2 = new Thread(tg1, runnable,"two");
      t2.start();
      Thread t3 = new Thread(tg1, runnable,"three");
      t3.start();

      System.out.println("Thread Group Name: "+tg1.getName());
    tg1.list();  // java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
                                          Thread[one,5,Parent ThreadGroup]
                                          Thread[two,5,Parent ThreadGroup]
                                          Thread[three,5,Parent ThreadGroup]
```

Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. **Synchronized method.**
        1. Synchronized method is used to lock an object for any shared resource.
        2. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
synchronized returnType methodName() {

 //code block

}
```

    2. **Synchronized block**.
        1. Synchronized block can be used to perform synchronization on any specific resource of the method.
        2. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
        3. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

```
synchronized (object reference expression) {
 //code block
}
```

    3. **static synchronization**.
        1. If you make any static method as synchronized, the lock will be on the class not on object.

```
Synchronized static returnType methodName() {
                        //code block
}
```

        Suppose there are two objects of a shared class(e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

Deadlock : t1 is waiting for resource2 which is locked by t2 already, and it is waiting for resource1 locked by t1;

```
1.  public class TestDeadlockExample1 {
2.    public static void main(String[] args) {
3.      final String resource1 = "ratan jaiswal";
4.      final String resource2 = "vimal jaiswal";
5.      // t1 tries to lock resource1 then resource2
6.      Thread t1 = new Thread() {
7.        public void run() {
8.          synchronized (resource1) {
9.           System.out.println("Thread 1: locked resource 1");
10.
11.          try { Thread.sleep(100);} catch (Exception e) {}
12.
13.          synchronized (resource2) {
14.           System.out.println("Thread 1: locked resource 2");
15.          }
16.        }
17.      }
18.     };
19.
20.     // t2 tries to lock resource2 then resource1
21.     Thread t2 = new Thread() {
22.       public void run() {
23.         synchronized (resource2) {
24.           System.out.println("Thread 2: locked resource 2");
25.
26.          try { Thread.sleep(100);} catch (Exception e) {}
27.
28.          synchronized (resource1) {
29.            System.out.println("Thread 2: locked resource 1");
30.          }
31.        }
32.      }
33.     };
34.
35.
36.     t1.start();
37.     t2.start();  }}
```

Inter-thread communication

It is implemented by following methods of **Object class**, why in object class? It is because they are related to lock and object has a lock.

- wait()
  - Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
  - it must be called from the synchronized method , because it has to own the monitor.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

- notify()
  - Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

    public final void notify()

- notifyAll()
  - Wakes up all threads that are waiting on this object's monitor. Syntax:

    public final void notifyAll()

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Interrupting a Thread

- **public void interrupt()**
  - If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException
  - If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.
- **public static boolean interrupted(): returns flag and set it false, if true.**
- **public boolean isInterrupted() : return the interrupt flag**

Reentrant Monitor

- java thread can reuse the same monitor for different synchronized methods if method is called from the method.
- It eliminates the possibility of single thread deadlocking

```
class Reentrant {
    public synchronized void m() {
    n();  //lock n
          //out of N, still lock exists
    System.out.println("this is m() method");
    }
    public synchronized void n() {
    System.out.println("this is n() method");
    }
}
```

- ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.
- Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.