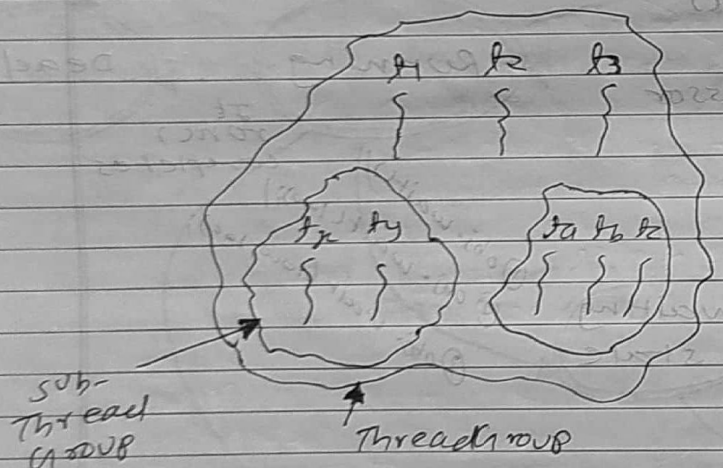
Multi-Thread Enhancements

⊛ ThreadGroup

↝ Based on functionality, we can group threads into a single unit which is nothing but thread group i.e thread group containing a group of threads.

↝ In Addition to threads, threadGroup can also contain sub-ThreadGroup



↝ The main Advantage of maintaining thread in the form of thread-group is, we can perform common operations very easily.

↝ Every thread in java belongs to some group, main thread belong to main thread-group.

↝ Every thread-group in java is the child group of system-group either directly (or) indirectly.

↝ Hence system-group act as root for all thread-group in java.

↝ system-group contains several system-level-threads :-

1) Finalizer / Garbage Collector
2) ReferenceHandler
3) Signal Dispatcher
4) Attach Listener

⋮

etc.

Ex-

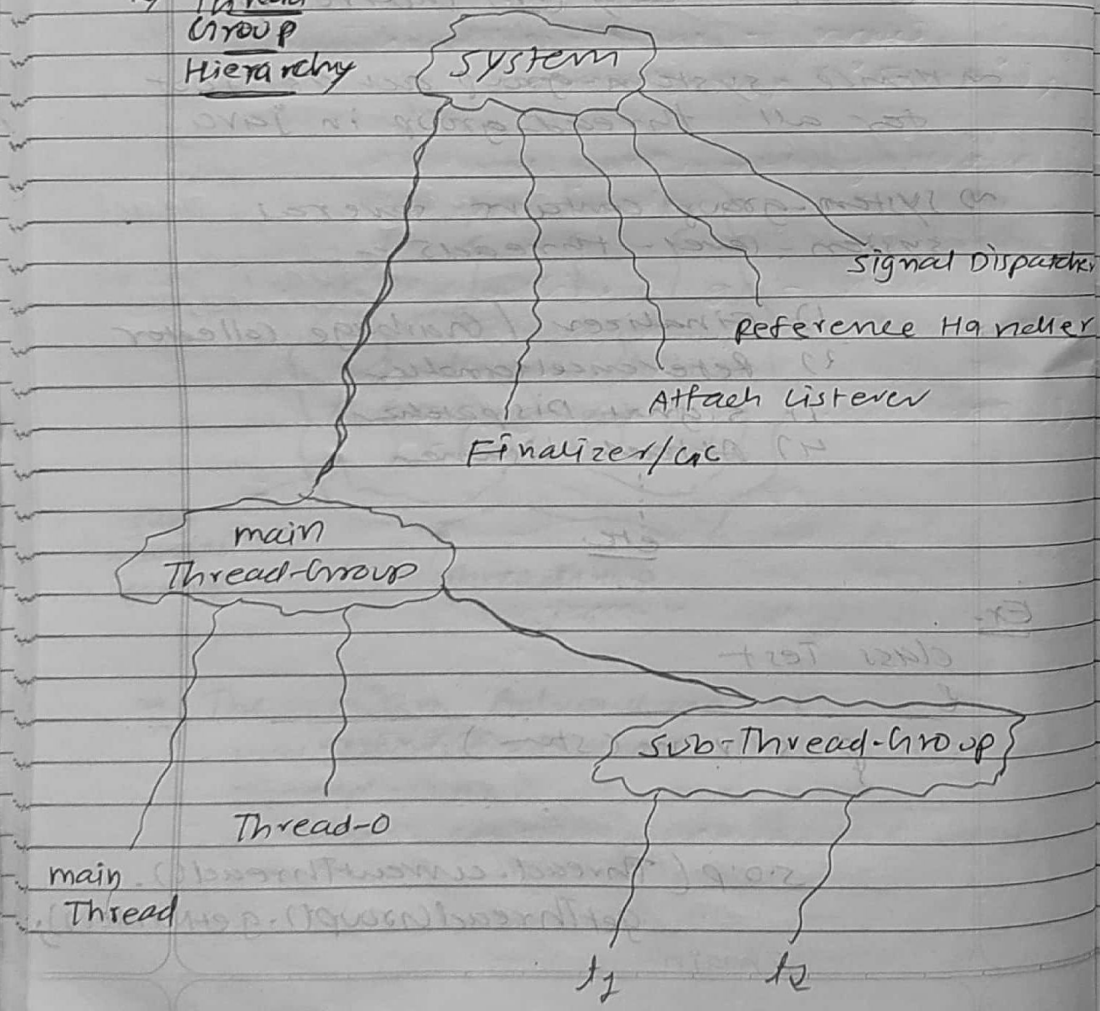
```
class Test
{
    p s v m(Str-)
    {

        S.O.P(Thread.currentThread().
              getThreadGroup().getName())
    //main
```

s.o.p(Thread.~~get~~currentThread().
getThreadGroup().getParent().
getName());

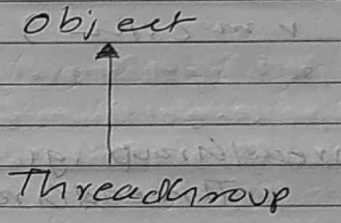
// system

}
}

⇒ Thread Group Hierarchy



⇝ ThreadGroup is a java-class, present in java-lang package and it is direct child class of object



Constructors :-

① ThreadGroup g = new ThreadGroup (String gname);

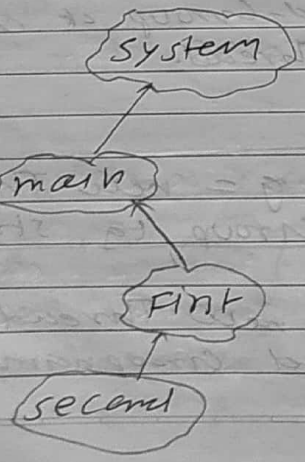
- creates a new ThreadGroup with specified group name.
- The parent of this new Group is The Thread-Group of currently executing Thread.

② ThreadGroup g = new ThreadGroup (ThreadGroup tg, String gname);

- creates a new ThreadGroup with the specified Groupname.

- The parent of the new ThreadGroup is specified parent group.

Ex.
```
class Test
{
    p s v m(str[])
    {
        ThreadGroup g1 = new
            ThreadGroup("First");
        s.o.p(g1.getParent().getName());
        // main
        ThreadGroup g2 = new
            ThreadGroup(g1, "second");
        s.o.p(g2.getParent().getName());
    } // First
}
```



⇒ Important methods of ThreadGroup class :-

① String getName()

- returns name of the ThreadGroup

② int getMaxPriority()

- return max priority of ThreadGroup

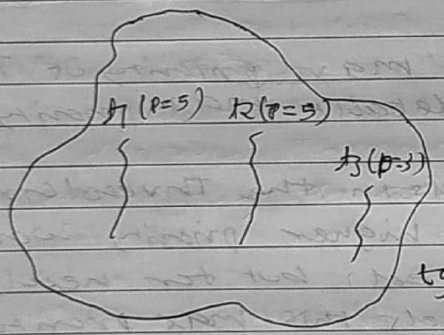
③ void setMaxPriority(int p)

- To set max priority of ThreadGroup
- The default max priority is 10.

- Threads in the ThreadGroup that already have higher priority won't be affected but for newly added Threads this max priority is applicable.

Ex.
```
class Test
{
    p s v m(str[])
    {
        ThreadGroup g1 = new
            ThreadGroup("tg");
```

```
Thread t1 = new Thread(g1, "t1");
Thread t2 = new Thread(g1, "t2");

g1.setMaxPriority(3);

Thread t3 = new Thread(g1, "t3");

s.o.p(t1.getPriority()); //5
s.o.p(t2.getPriority()); //5
s.o.p(t3.getPriority()); //3

}
}
```



max priority < ~~10~~ 3

④ **ThreadGroup getParent()**

- returns Parent Group of current Thread.

⑤ **void list()**

- print Info" about Thread-Group to the console

⑥ **int activeCount()**

- returns number of active threads present in thread-group.

⑦ **int activeGroupCount()**

- returns number of active groups present in current thread group.

⑧ **int enumerate(Thread[] t)**

- To copy all active threads of this Thread-Group into provided thread-array.
- In this case sub-ThreadGroup threads also considered.

⑨ **int enumerate(ThreadGroup[] g)**

- To copy all active thread-groups into Thread-Group Array.

(10) **boolean isDaemon()**

To check whether the ThreadGroup is Daemon or not.

(11) **void setDaemon(boolean b)**

(12) **void interrupt()**

- To interrupt all waiting / sleeping threads present in ThreadGroup.

(13) **void destroy()**

- To destroy ThreadGroup & its all sub-threadgroups.

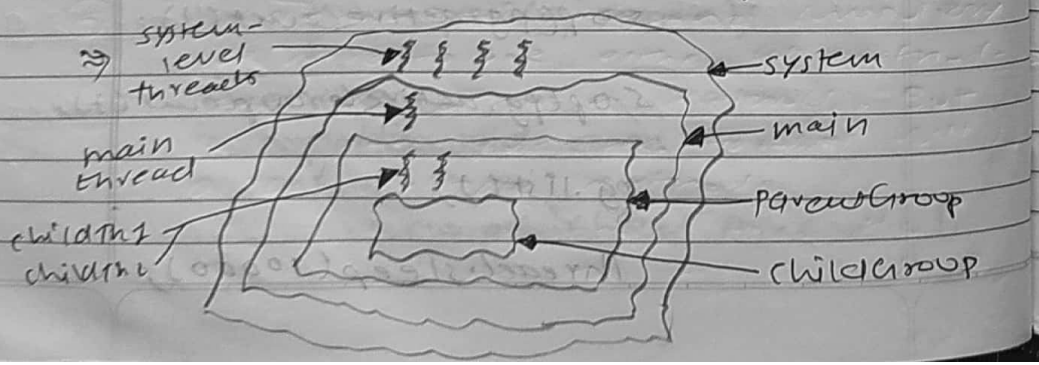
Ex.

```
class MyTh extends Thread
{
    MyTh(ThreadGroup g, String name)
    {
        super(g, name);
    }

    public void run()
    {
        s.o.p("Child Thread");
        try
        {
            Thread.sleep(5000);
        }
        catch(IE e)
        { }
    }
}
class Test
{
    p s v m(Sm) throws Exception
    {
        ThreadGroup pg = new
            ThreadGroup("ParentGroup");

        ThreadGroup cg = new
            ThreadGroup("ChildGroup");

        MyTh t1 = new MyTh(pg,
                           "ChildTh1");
        MyTh t2 = new MyTh(Pg,
                           "ChildTh2");

        t1.start();
        t2.start();

        s.o.p(pg.activeCount());

        s.o.p(pg.activeGroupCount());

        pg.list();

        Thread.sleep(10000);
```

```
        s.o.p(pg.activeCount());
        s.o.p(pg.activeGroupCount());

        pg.list();
    }
}
```

O/P :-

```
childThread
childThread
2
1
java.lang.ThreadGroup
[name = ParentGroup, maxpri = 10]
    Thread[childTh1, 5, ParentGroup]
    Thread[childTh2, 5, ParentGroup]
    java.lang.ThreadGroup
    [name = ChildGroup, maxpri = 10]

0
1

java.lang.ThreadGroup
[name = ParentGroup, maxpri = 10]
    java.lang.ThreadGroup
    [name = ChildGroup, maxpri = 10]
```



⇒ WAP to display all active thread names belongs to system groups & its child groups.

Ans:-

```
class Test
{
    p s v m(Str[])
    {
        ThreadGroup system =
            Thread.currentThread().
            getThreadGroup().getParent();

        Thread[] t = new
            Thread[system.activeCount()];

        system.enumerate(t);

        for(Thread tl:t)
        {
            s.o.p(tl.getName() + "---" +
                tl.isDaemon());
        }
    }
}
```

O/P :-

```
Reference Handler ---- true
Finalizer ---- true
Signal Dispatcher ---- true
Attach Listener ---- true
main ---- false
```

⊛ java.util.concurrent Package

⇒ Problems with the traditional synchronized keyword :-

① We are not having any flexibility to try for a lock without waiting.

② There is no way to specify maximum waiting time for a thread to get lock so that thread will wait until getting a lock, which may creates performance problem & may cause Deadlock.

③ If a thread releases lock then which waiting thread will get that lock, we are not having any control on this.

④ There is no API to list out all waiting threads for a lock.

⑤ synchronized keyword compulsory we have to use either at method-level (or) within a method. But it is not possible to use across multiple methods.

⇒ To overcome this problems SUN people introduced java.util.concurrent.locks package in 1.5V.

⇒ It also provides several Enhancements to the programmer to provide more control on concurrency.

⇒ Lock (I)

⇒ Lock object is similar to implicit lock acquired by a thread to execute synchronized method (or) synchronized block.

⇒ Lock implementation provide more extensive operations than traditional implicit locks.

⇒ Important methods of Lock (I) :-

① `void lock()`

- We can use this method to acquired a lock. If the lock is already available then current thread immediately get that lock. If the lock is not already

available then it will wait until getting the lock. It is exactly same behavior as traditional synchronized keyword.

(2) `boolean tryLock()`

- To acquire the lock without waiting. If lock is available then the thread acquire the lock & returns true. If lock is not available then this method return false & can continue its execution without waiting. In this case thread never be enter into waiting state.

ex.
```
if(l.tryLock())
{
    //perform safe operations
}
else
{
    //perform Alternative
      operations
}
```

(3) `boolean tryLock(long time, TimeUnit unit)`

- If lock is available then the thread will get the lock & continue its execution. If the lock is not available then the thread will wait until specified amount of time. Still if the lock is not available then thread can continue its execution.

⇒ TimeUnit :-

- TimeUnit is an Enum present in java.util.concurrent package.

```
enum TimeUnit
{
    NANOSECONDS,
    MICROSECONDS,
    MILLISECONDS,
    SECONDS,
    MINUTES,
    HOURS,
    DAYS;
}
```

ex.
```
if(l.tryLock(1000, TimeUnit.
                  MILLISECONDS))
{
}
```

(4) `void lockInterruptibly()`

- Acquires a lock if it is available & returns immediately. If the lock is not available then it will wait while waiting if the thread is interrupted then thread won't get the lock.

(5) **void unlock()**

- To unlock a lock & To call this method compulsory current thread should be ~~owner~~ owner of this lock otherwise we will get R.E.

  IllegalMonitorStateException