CORE JAVA

With

SCJP / OCJP

Study Material

Chapter 8: Multi Threading Enhancements

DURGA M.Tech

(Sun certified & Realtime Expert)

DURGA SOFTWARE SOLUTIONS

www.durgasoft.com Ph: 9246212143 8096969696

Multi Threading Enhancements

8.1) ThreadGroup

8.2) ThreadLocal

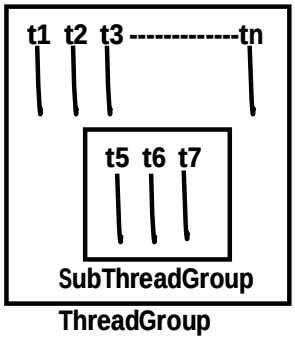
8.3) java.util.concurrent.locks package
-> Lock(I)
-> ReentrantLock(C)

8.4) Thread Pools

8.5) Callable and Future

ThreadGroup:

- **Based on the Functionality we can Group Threads into a Single Unit which is Nothing but ThreadGroup i.e. ThreadGroup Represents a Set of Threads.**
- **In Addition a ThreadGroup can Also contains Other SubThreadGroups.**



- **ThreadGroup Class Present in java.lang Package and it is the Direct Child Class of Object.**
- **ThreadGroup provides a Convenient Way to Perform Common Operation for all Threads belongs to a Particular Group.**

Eg:Stop All Consumer Threads.
Suspend All Producer Threads.

Constructors:

1) ThreadGroup g = new ThreadGroup(String gname);

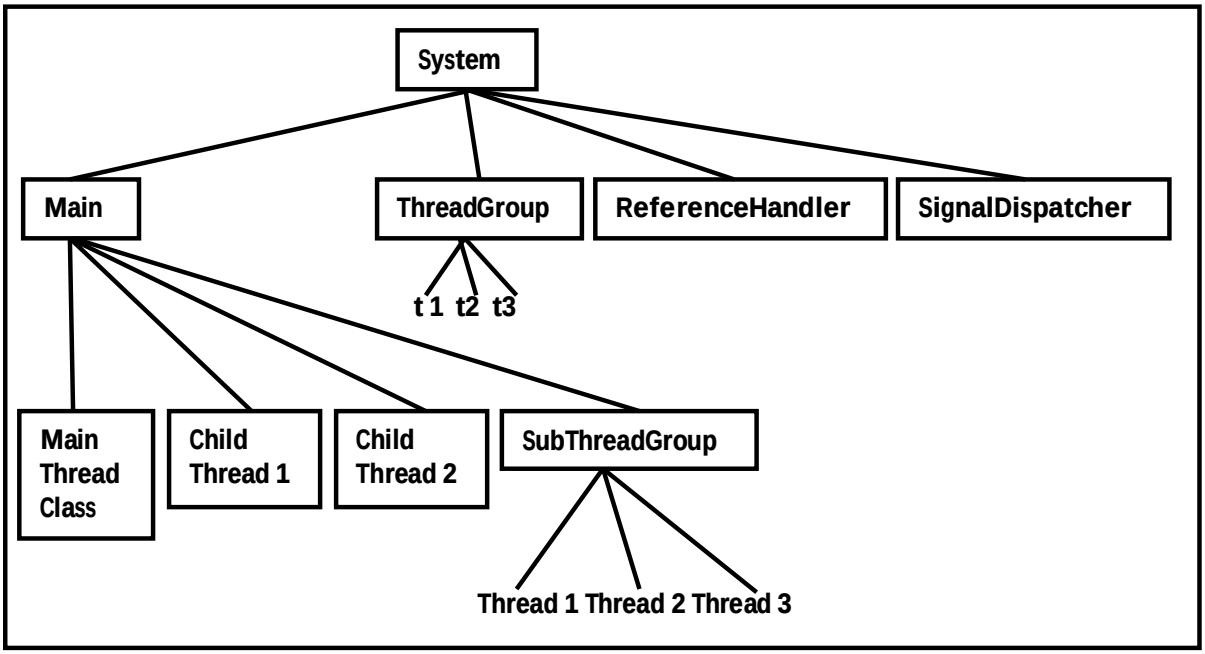
- **Createsa New ThreadGroup.**
- **The Parent of this New Group is the ThreadGroup of Currently Running Thread.**

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

- **Creates a New ThreadGroup.**
- **The Parent of this ThreadGroup is the specified ThreadGroup.**

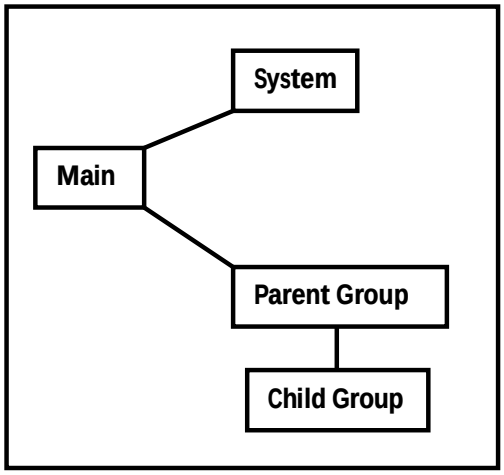
Note:

- **In Java Every Thread belongs to Some Group.**
- **Every ThreadGroup is the Child Group of *System Group* either Directly OR Indirectly. Hence SystemGroup Acts as Root for all ThreadGroup's in Java.**
- **System ThreadGroup Represents System Level Threads Like ReferenceHandler, SignalDispatcher, Finalizer, AttachListener Etc.**



```
class ThreadGroupDemo {
public static void main(String[] args) {
        System.out.println(Thread.currentThread().getThreadGroup().getName());
        System.out.println(Thread.currentThread().getThreadGroup().getParent().getName());
        ThreadGroup pg = new ThreadGroup("Parent Group");
        System.out.println(pg.getParent().getName());
        ThreadGroup cg = new ThreadGroup(pg, "Child Group");
        System.out.println(cg.getParent().getName());
}
}
```

```
main
system
main
Parent Group
```



Important Methods of ThreadGroup Class:

1) **String getName();** Returns Name of the ThreadGroup.

2) **int getMaxPriority();** Returns the Maximum Priority of ThreadGroup.

3) **void setMaxPriority();**
   - To Set Maximum Priority of ThreadGroup.
   - The Default Maximum Priority is 10.
   - Threads in the ThreadGroup that Already have Higher Priority, Not effected but Newly Added Threads this MaxPriority is Applicable.

```
class ThreadGroupDemo {
        public static void main(String[] args) {
                ThreadGroup g1 = new ThreadGroup("tg");
                Thread t1 = new Thread(g1, "Thread 1");
                Thread t2 = new Thread(g1, "Thread 2");
                g1.setMaxPriority(3);
                Thread t3 = new Thread(g1, "Thread 3");
                System.out.println(t1.getPriority());→ 5
                System.out.println(t2.getPriority());→ 5
                System.out.println(t3.getPriority());→ 3
        }
}
```

4) **ThreadGroup getParent():** Returns Parent Group of Current ThreadGroup.

5) **void list():** It Prints Information about ThreadGroup to the Console.

6) **int activeCount():** Returns Number of Active Threads Present in the ThreadGroup.

7) **int activeGroupCount():** It Returns Number of Active ThreadGroups Present in the Current ThreadGroup.

8) **int enumerate(Thread[] t):** To Copy All Active Threads of this Group into provided Thread Array. In this Case SubThreadGroup Threads also will be Considered.

9) **int enumerate(ThreadGroup[] g):** To Copy All Active SubThreadGroups into ThreadGroupArray.

10) **boolean isDaemon():**

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

12) **void interrupt():** To Interrupt All Threads Present in the ThreadGroup.

13) **void destroy():** To Destroy ThreadGroup and its SubThreadGroups.

```
class MyThread extends Thread {
        MyThread(ThreadGroup g, String name) {
                super(g, name);
        }
        public void run() {
          System.out.println("Child Thread");
          try {
                  Thread.sleep(2000);
          }
          catch (InterruptedException e) {}
        }
}
class ThreadGroupDemo {
        public static void main(String[] args) throws InterruptedException {
                ThreadGroup pg = new ThreadGroup("Parent Group");
                ThreadGroup cg = new ThreadGroup(pg, "Child Group");
                MyThread t1 = new MyThread(pg, "Child Thread 1");
                MyThread t2 = new MyThread(pg, "Child Thread 2");
                t1.start();
                t2.start();
                System.out.println(pg.activeCount());
                System.out.println(pg.activeGroupCount());
                pg.list();
                Thread.sleep(5000);
                System.out.println(pg.activeCount());
                pg.list();
        }
}
```

```
2
1
java.lang.ThreadGroup[name=Parent Group,maxpri=10]
   Thread[Child Thread 1,5,Parent Group]
   Thread[Child Thread 2,5,Parent Group]
   java.lang.ThreadGroup[name=Child Group,maxpri=10]
Child Thread
Child Thread
0
java.lang.ThreadGroup[name=Parent Group,maxpri=10]
   java.lang.ThreadGroup[name=Child Group,maxpri=10]
```

Write a Program to Display All Thread Names belongs to System Group

```
class ThreadGroupDemo {
public static void main(String[] args) {
        ThreadGroup system = Thread.currentThread().getThreadGroup().getParent();
        Thread[] t = new Thread[system.activeCount()];
        system.enumerate(t);
        for (Thread t1: t) {
                System.out.println(t1.getName()+"-------"+t1.isDaemon());
        }
}
}
```

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

ThreadLocal:

- **ThreadLocal Provides ThreadLocal Variables.**
- **ThreadLocal Class Maintains Values for Thread Basis.**
- **Each ThreadLocal Object Maintains a Separate Value Like userID, transactionID Etc for Each Thread that Accesses that Object.**
- **Thread can Access its Local Value, can Manipulates its Value and Even can Remove its Value.**
- **In Every Part of the Code which is executed by the Thread we can Access its Local Variables.**

Eg:

- **Consider a Servlet which Calls Some Business Methods. we have a Requirement to generate a Unique transactionID for Each and Every Request and we have to Pass this transactionID to the Business Methods for Logging Purpose.**
- **For this Requirement we can Use ThreadLocal to Maintain a Separate transactionID for Every Request and for Every Thread.**

Note:

- **ThreadLocal Class introduced in 1.2 Version.**
- **ThreadLocal can be associated with Thread Scope.**
- **All the Code which is executed by the Thread has Access to Corresponding ThreadLocal Variables.**
- **A Thread can Access its Own Local Variables and can't Access Other Threads Local Variables.**
- **Once Thread Entered into Dead State All Local Variables are by Default Eligible for Garbage Collection.**

Constructor:ThreadLocal tl = new ThreadLocal();
Creates a ThreadLocal Variable.

Methods:

1) **Object get(); Returns the Value of ThreadLocal Variable associated with Current Thread.**

2) **Object initialValue();**
   - **Returns the initialValue of ThreadLocal Variable of Current Thread.**
   - **The Default Implementation of initialValue() Returns null.**
   - **To Customize Our initialValue we have to Override initialValue().**

3) **void set(Object newValue);To Set a New Value.**

4) **void remove();**
   - **To Remove the Current Threads Local Variable Value.**
   - **After Remove if we are trying to Access it will be reinitialized Once Again by invoking its initialValue().**
   - **This Method Newly Added in 1.5 Version.**

```
class ThreadLocalDemo {
public static void main(String[] args) {
        ThreadLocal tl = new ThreadLocal();
        System.out.println(tl.get()); //null
        tl.set("Durga");
        System.out.println(tl.get()); //Durga
        tl.remove();
        System.out.println(tl.get()); //null
}
}
```

```
//Overriding of intialValue()
class ThreadLocalDemo {
public static void main(String[] args) {
        ThreadLocal tl = new ThreadLocal() {
            protected Object initialValue() {
                return "abc";
}
        };
        System.out.println(tl.get()); //abc
        tl.set("Durga");
        System.out.println(tl.get()); //Durga
        tl.remove();
        System.out.println(tl.get()); //abc
}
}
```

```
class CustomerThread extends Thread {
        static Integer custID = 0;
        private static ThreadLocal tl = new ThreadLocal() {
            protected Integer initialValue() {
                return ++custID;
            }
        };
        CustomerThread(String name) {
            super(name);
        }
        public void run() {
            for (int i=0; i<5; i++) {
SOP(Thread.currentThread().getName()+" Executing with Customer ID:"+tl.get());
            }
        }
}
class ThreadLocalDemo {
        public static void main(String[] args) {
                CustomerThread c1 = new CustomerThread("CustomerThread - 1");
                CustomerThread c2 = new CustomerThread("CustomerThread - 2");
                CustomerThread c3 = new CustomerThread("CustomerThread - 3");
                CustomerThread c4 = new CustomerThread("CustomerThread - 4");
                c1.start();
                c2.start();
                c3.start();
                c4.start();
        }
}
```

```
CustomerThread - 1 Executing with Customer ID:1
CustomerThread - 1 Executing with Customer ID:1
CustomerThread - 1 Executing with Customer ID:1
CustomerThread - 1 Executing with Customer ID:1
CustomerThread - 1 Executing with Customer ID:1

CustomerThread - 2 Executing with Customer ID:2
CustomerThread - 2 Executing with Customer ID:2
CustomerThread - 2 Executing with Customer ID:2
CustomerThread - 2 Executing with Customer ID:2
CustomerThread - 2 Executing with Customer ID:2

CustomerThread - 3 Executing with Customer ID:3
CustomerThread - 3 Executing with Customer ID:3
CustomerThread - 3 Executing with Customer ID:3
CustomerThread - 3 Executing with Customer ID:3
CustomerThread - 3 Executing with Customer ID:3

CustomerThread - 4 Executing with Customer ID:4
CustomerThread - 4 Executing with Customer ID:4
CustomerThread - 4 Executing with Customer ID:4
CustomerThread - 4 Executing with Customer ID:4
CustomerThread - 4 Executing with Customer ID:4
CustomerThread - 4 Executing with Customer ID:4
```

In the Above Program for Every Customer Thread a Separate customerID will be maintained by ThreadLocal Object.

ThreadLocal Vs Inheritance:

- **Parent Threads ThreadLocal Variables are by Default Not Available to the Child Threads.**
- **If we want to Make Parent Threads Local Variables Available to Child Threads we should go for InheritableThreadLocal Class.**
- **It is the Child Class of ThreadLocal Class.**
- **By Default Child Thread Values are Same as Parent Thread Values but we can Provide Customized Values for Child Threads by Overriding childValue().**

Constructor:InheritableThreadLocal itl = new InheritableThreadLocal();
InheritableThreadLocal is the Child Class of ThreadLocal and Hence All Methods Present in ThreadLocal by Default Available to the InheritableThreadLocal.

Method: public Object childValue(Object pvalue);

```
class ParentThread extends Thread {
public static InheritableThreadLocal itl = new InheritableThreadLocal() {
        public Object childValue(Object p) {
    return "cc";
    }
};
public void run() {
        itl.set("pp");
    System.out.println("Parent Thread --"+itl.get());
    ChildThread ct = new ChildThread();
    ct.start();
}
class ChildThread extends Thread {
        public void run() {
                System.out.println("Child Thread --"+ParentThread.itl.get());
    }
}
}
class ThreadLocalDemo {
    public static void main(String[] args) {
            ParentThread pt = new ParentThread();
            pt.start();
    }
}
```

```
Parent Thread --pp
Child Thread --cc
```

Java.util.concurrent.locks package:

Problems with Traditional synchronized Key Word

- If a Thread Releases the Lock then which waiting Thread will get that Lock we are Not having any Control on this.
- We can't Specify Maximum waiting Time for a Thread to get Lock so that it will Wait until getting Lock, which May Effect Performance of the System and Causes Dead Lock.
- We are Not having any Flexibility to Try for Lock without waiting.
- There is No API to List All Waiting Threads for a Lock.
- The synchronized Key Word Compulsory we have to Define within a Method and it is Not Possible to Declare Over Multiple Methods.
- To Overcome Above Problems SUN People introduced *java.util.concurrent.locks* Package in 1.5 Version.
- It Also Provides Several Enhancements to the Programmer to Provide More Control on Concurrency.

Lock(I):

- A Lock Object is Similar to Implicit Lock acquired by a Thread to Execute synchronized Method OR synchronized Block
- Lock Implementations Provide More Extensive Operations than Traditional Implicit Locks.

Important Methods of Lock Interface

1) void lock();

- It Locks the Lock Object.
- If Lock Object is Already Locked by Other Thread then it will wait until it is Unlocked.

2) boolean tryLock();

- To Acquire the Lock if it is Available.
- If the Lock is Available then Thread Acquires the Lock and Returns true.
- If the Lock Unavailable then this Method Returns false and Continue its Execution.
- In this Case Thread is Never Blocked.

```
if (l.tryLock()) {
Perform Safe Operations
}
else {
   Perform Alternative Operations
}
```

3) **boolen tryLock(long time, TimeUnit unit);**
   - **To Acquire the Lock if it is Available.**
   - **If the Lock is Unavailable then Thread can Wait until specified Amount of Time.**
   - **Still if the Lock is Unavailable then Thread can Continue its Execution.**

   Eg: if (l.tryLock(1000, TimeUnit.SECONDS)) {}

TimeUnit: TimeUnit is an *enum* Present in *java.util.concurrent* Package.

```
enum TimeUnit {
        DAYS, HOURS, MINUTES, SECONDS, MILLI SECONDS, MICRO SECONDS, NANO SECONDS;
}
```

4) **void lockInterruptedly();**

   Acquired the Lock Unless the Current Thread is Interrupted.
   Acquires the Lock if it is Available and Returns Immediately.
   If it is Unavailable then the Thread will wait while waiting if it is Interrupted then it won't get the Lock.

5) **void unlock(); To Release the Lock.**

ReentrantLock

- **It implements Lock Interface and it is the Direct Child Class of an Object.**
- **Reentrant Means a Thread can acquires Same Lock Multiple Times without any Issue.**
- **Internally ReentrantLock Increments Threads Personal Count whenever we Call lock() and Decrements Counter whenever we Call unlock() and Lock will be Released whenever Count Reaches '0'.**

Constructors:

1) **ReentrantLock rl = new ReentrantLock();**
   Creates an Instance of ReentrantLock.

2) **ReentrantLock rl = new ReentrantLock(boolean fairness);**
   - **Creates an Instance of ReentrantLock with the Given Fairness Policy.**
   - **If Fairness is true then Longest Waiting Thread can acquired Lock Once it is Avaiable i.e. if follows First - In First – Out.**
   - **If Fairness is false then we can't give any Guarantee which Thread will get the Lock Once it is Available.**

   Note: If we are Not specifying any Fairness Property then by Default it is Not Fair.

Which of the following 2 Lines are Equal?

```
ReentrantLock rl = new ReentrantLock(); √

ReentrantLock rl = new ReentrantLock(true);

ReentrantLock rl = new ReentrantLock(false); √
```

Important Methods of ReentrantLock

1) **void lock();**

2) **boolean tryLock();**

3) **boolean tryLock(long l, TimeUnit t);**

4) **void lockInterruptedly();**

5) **void unlock();**
   - **To Realease the Lock.**
   - **If the Current Thread is Not Owner of the Lock then we will get Runtime Exception Saying IllegalMonitorStateException.**

6) **int getHoldCount(); Returns Number of Holds on this Lock by Current Thread.**

7) **boolean isHeldByCurrentThread(); Returns true if and Only if Lock is Hold by Current Thread.**

8) **int getQueueLength(); Returns the Number of Threads waiting for the Lock.**

9) **Collection getQueuedThreads(); Returns a Collection containing Thread Objects which are waiting to get the Lock.**

10) **boolean hasQueuedThreads();** Returns true if any Thread waiting to get the Lock.

11) **boolean isLocked();** Returns true if the Lock acquired by any Thread.

12) **boolean isFair();**Returns true if the Lock's Fairness Set to true.

13) **Thread getOwner();**Returns the Thread which acquires the Lock.

```
import java.util.concurrent.locks.ReentrantLock;
class Test {
        public static void main(String[] args) {
                ReentrantLock l = new ReentrantLock();
                l.lock();

                l.lock();
                System.out.println(l.isLocked()); //true
                System.out.println(l.isHeldByCurrentThread()); //true
                System.out.println(l.getQueueLength()); //0

                l.unlock();
                System.out.println(l.getHoldCount()); //1
                System.out.println(l.isLocked()); //true

                l.unlock();
                System.out.println(l.isLocked()); //false
                System.out.println(l.isFair()); //false
        }
}
```

```
import java.util.concurrent.locks.ReentrantLock;
class Display {
ReentrantLock l = new ReentrantLock(true);
public void wish(String name) {
        l.lock();→ 1
                for(int i=0; i<5; i++) {
                System.out.println("Good Morning");
                try {
                        Thread.sleep(2000);
                }
                catch(InterruptedException e) {}
                System.out.println(name);
        }
        l.unlock();→ 2
}
}
class MyThread extends Thread {
        Display d;
        String name;
        MyThread(Display d, String name) {
                this.d = d;
                this.name = name;
        }
        public void run() {
                d.wish(name);
        }
}
class ReentrantLockDemo {
   public static void main(String[] args) {
                Display d = new Display();
                MyThread t1 = new MyThread(d, "Dhoni");
                MyThread t2 = new MyThread(d, "Yuva Raj");
                MyThread t3 = new MyThread(d, "Virat Kohli");
                t1.start();
                t2.start();
                t3.start();
        }
}
```

```
Good Morning
Dhoni
Good Morning
Dhoni
Good Morning
Dhoni
Good Morning
Dhoni
Good Morning
Dhoni
Good Morning
Yuva Raj
Good Morning
Yuva Raj
Good Morning
Yuva Raj
Good Morning
Yuva Raj
Good Morning
Yuva Raj
Good Morning
Virat Kohli
Good Morning
Virat Kohli
Good Morning
Virat Kohli
Good Morning
Virat Kohli
Good Morning
Virat Kohli
```

If we Comment Both Lines 1 and 2 then All Threads will be executed Simultaneously and Hence we will get Irregular Output.

If we are Not Commenting then the Threads will be executed One by One and Hence we will get Regular Output

Demo Program To Demonstrate tryLock();

```
import java.util.concurrent.locks.ReentrantLock;
class MyThread extends Thread {
static ReentrantLock l = new ReentrantLock();
MyThread(String name) {
        super(name);
}
public void run() {
        if(l.tryLock()) {
SOP(Thread.currentThread().getName()+" Got Lock and Performing Safe Operations");
        try {
                        Thread.sleep(2000);
        }
        catch(InterruptedException e) {}
        l.unlock();
        }
        else {
                        System.out.println(Thread.currentThread().getName()+" Unable To Get Lock
and Hence Performing Alternative Operations");
                }
        }
}
class ReentrantLockDemo {
        public static void main(String args[]) {
                MyThread t1 = new MyThread("First Thread");
                MyThread t2 = new MyThread("Second Thread");
                t1.start();
                t2.start();
        }
}
```

```
First Thread Got Lock and Performing Safe Operations
Second Thread Unable To Get Lock and Hence Performing Alternative Operations
```

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;
class MyThread extends Thread {
	static ReentrantLock l = new ReentrantLock();
	MyThread(String name) {
		super(name);
	}
	public void run() {
		do {
			try {
				if(l.tryLock(1000, TimeUnit.MILLISECONDS)) {
					SOP(Thread.currentThread().getName()+"------- Got Lock");
Thread.sleep(5000);
l.unlock();
SOP(Thread.currentThread().getName()+"------- Releases Lock");
break;
				}
				else {
SOP(Thread.currentThread().getName()+"------- Unable To Get Lock And Will Try Again");
				}
			}
			catch(InterruptedException e) {}
		}
		while(true);
	}
}
class ReentrantLockDemo {
	public static void main(String args[]) {
		MyThread t1 = new MyThread("First Thread");
		MyThread t2 = new MyThread("Second Thread");
		t1.start();
		t2.start();
	}
}
```

```
First Thread------- Got Lock
Second Thread------- Unable To Get Lock And Will Try Again
Second Thread------- Unable To Get Lock And Will Try Again
Second Thread------- Unable To Get Lock And Will Try Again
Second Thread------- Unable To Get Lock And Will Try Again
Second Thread------- Got Lock
First Thread------- Releases Lock
Second Thread------- Releases Lock
```

Thread Pools:

- **Creating a New Thread for Every Job May Create Performance and Memory Problems.**
- **To Overcome this we should go for Thread Pool Concept.**
- **Thread Pool is a Pool of Already Created Threads Ready to do Our Job.**
- **Java 1.5 Version Introduces Thread Pool Framework to Implement Thread Pools.**
- **Thread Pool Framework is Also Known as Executor Framework.**
- **We can Create a Thread Pool as follows**
  ExecutorService service = Executors.newFixedThreadPool(3);//Our Choice
- **We can Submit a Runnable Job by using *submit()*.**
  service.submit(job);
- **We can Shutdown *ExecutiorService* by using *shutdown()*.**
  service.shutdown();

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class PrintJob implements Runnable {
        String name;
        PrintJob(String name) {
           this.name = name;
        }
        public void run() {
        SOP(name+"....Job Started By Thread:" +Thread.currentThread().getName());
                try {
                        Thread.sleep(10000);
                }
                catch (InterruptedException e) {}
                SOP(name+"....Job Completed By Thread:" +Thread.currentThread().getName());
        }
}
class  ExecutorDemo {
        public static void main(String[] args) {
                PrintJob[] jobs = {
new PrintJob("Durga"),
new PrintJob("Ravi"),
new PrintJob("Nagendra"),
new PrintJob("Pavan"),
new PrintJob("Bhaskar"),
new PrintJob("Varma")
};
                ExecutorService service = Executors.newFixedThreadPool(3);
                for (PrintJob job : jobs) {
                        service.submit(job);
                }
                service.shutdown();
        }
}
```

Output

```
Durga....Job Started By Thread:pool-1-thread-1
Ravi....Job Started By Thread:pool-1-thread-2
Nagendra....Job Started By Thread:pool-1-thread-3
Ravi....Job Completed By Thread:pool-1-thread-2
Pavan....Job Started By Thread:pool-1-thread-2
Durga....Job Completed By Thread:pool-1-thread-1
Bhaskar....Job Started By Thread:pool-1-thread-1
Nagendra....Job Completed By Thread:pool-1-thread-3
Varma....Job Started By Thread:pool-1-thread-3
Pavan....Job Completed By Thread:pool-1-thread-2
Bhaskar....Job Completed By Thread:pool-1-thread-1
Varma....Job Completed By Thread:pool-1-thread-3
```

On the Above Program 3 Threads are Responsible to Execute 6 Jobs. So that a Single Thread can be reused for Multiple Jobs.

Note: Usually we can Use ThreadPool Concept to Implement Servers (Web Servers And Application Servers).

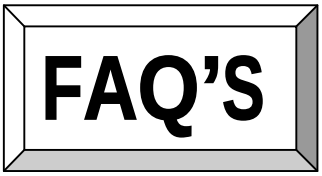
Callable and Future:

- **In the Case of Runnable Job Thread won't Return anything.**
- **If a Thread is required to Return Some Result after Execution then we should go for Callable.**
- **Callable Interface contains Only One Method *public Object call() throws Exception.***
- **If we Submit a Callable Object to Executor then the Framework Returns an Object of Type *java.util.concurrent.Future***
- **The Future Object can be Used to Retrieve the Result from Callable Job.**

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class MyCallable implements Callable {
        int num;
        MyCallable(int num) {
                this.num = num;
        }
        public Object call() throws Exception {
                int sum = 0;
                for(int i=0; i<num; i++) {
                        sum = sum+i;
                }
                return sum;
        }
}
class CallableFutureDemo {
        public static void main(String args[]) throws Exception {
                MyCallable[] jobs = {
new MyCallable(10),
new MyCallable(20),
new MyCallable(30),
new MyCallable(40),
new MyCallable(50),
new MyCallable(60)
                                        };
                ExecutorService service = Executors.newFixedThreadPool(3);
                for(MyCallable job : jobs) {
                        Future f = service.submit(job);
                        System.out.println(f.get());
                }
                service.shutdown();
        }
}
```

```
45
190
435
780
1225
1770
```



1) **What Is Multi Tasking?**

2) **What Is Multi Threading And Explain Its Application Areas?**

3) **What Is Advantage Of Multi Threading?**

4) **When Compared With C++ What Is The Advantage In Java With Respect To Multi Threading?**

5) **In How Many Ways We Can Define A Thread?**

6) **Among Extending Thread And Implementing Runnable Which Approach Is Recommended?**

7) **Difference Between t.start() And t.run()?**

8) **Explain About Thread Scheduler?**

9) **If We Are Not Overriding run() What Will Happen?**

10) **Is It Possible Overloading Of run()?**

11) **Is It Possible To Override a start() And What Will Happen?**

12) **Explain Life Cycle Of A Thread?**

13) **What Is The Importance Of Thread Class start()?**

14) **After Starting A Thread If We Try To Restart The Same Thread Once Again What Will Happen?**

15) **Explain Thread Class Constructors?**

16) **How To Get And Set Name Of A Thread?**

17) **Who Uses Thread Priorities?**

18) **Default Priority For Main Thread?**

19) **Once We Create A New Thread What Is Its Priority?**

20) How To Get Priority From Thread And Set Priority To A Thread?

21) If We Are Trying To Set Priority Of Thread As 100, What Will Happen?

22) If 2 Threads Having Different Priority Then Which Thread Will Get Chance First For Execution?

23) If 2 Threads Having Same Priority Then Which Thread Will Get Chance First For Execution?

24) How We Can Prevent Thread From Execution?

25) What Is yield() And Explain Its Purpose?

26) Is Join Is Overloaded?

27) Purpose Of sleep()?

28) What Is synchronized Key Word? Explain Its Advantages And Disadvantages?

29) What Is Object Lock And When It Is Required?

30) What Is A Class Level Lock When It Is Required?

31) While A Thread Executing Any Synchronized Method On The Given Object Is It Possible To Execute Remaining Synchronized Methods On The Same Object Simultaneously By Other Thread?

32) Difference Between Synchronized Method And Static Synchronized Method?

33) Advantages Of Synchronized Block Over Synchronized Method?

34) What Is Synchronized Statement?

35) How 2 Threads Will Communicate With Each Other?

36) wait(), notify(), notifyAll() Are Available In Which Class?

37) Why wait(), notify(), notifyAll() Methods Are Defined In Object Instead Of Thread Class?

38) Without Having The Lock Is It Possible To Call wait()?

39) If A Waiting Thread Gets Notification Then It Will Enter Into Which State?

40) In Which Methods Thread Can Release Lock?

41) Explain wait(), notify() and notifyAll()?

42) Difference Between notify() and notifyAll()?

43) Once A Thread Gives Notification Then Which Waiting Thread Will Get A Chance?

44) How A Thread Can Interrupt Another Thread?

45) What Is Deadlock? Is It Possible To Resolve Deadlock Situation?

46) Which Keyword Causes Deadlock Situation?

47) How We Can Stop A Thread Explicitly?

48) Explain About suspend() And resume()?

49) What Is Starvation And Explain Difference Between Deadlock and Starvation?

50) What Is Race Condition?

51) What Is Daemon Thread? Give An Example Purpose Of Daemon Thread?

52) How We Can Check Daemon Nature Of A Thread? Is It Possible To Change Daemon Nature Of A Thread? Is Main Thread Daemon OR Non-Daemon?

53) Explain About ThreadGroup?

54) What Is ThreadLocal?