ThreadGroup

Based on functionality we can group thread into a single unit which is nothing but ThreadGroup.

ThreadGroup contains a group of thread. In addition to thread threadgroup can also contain subthread groups.

Advantage:
The main advantage of maintaining thread in the form of threadgroup is we can perform common operations very easily.

Every thread in Java belongs to ThreadGroup.

S.o.p(Thread.currentThread().getThreadGroup().getName())

group name => main

S.o.p(Thread.currentThread().getThreadGroup().getParent().getName)
main main threadgroup System threadgroup system

Every thread in Java belongs to some group.
Main thread belongs to maingroup.
Every threadgroup in Java is a childgroup of Systemgroup either directly or indirectly. Hence Systemgroup is a root for all threadgroups in Java.

Systemgroup contain several system level threads like
Finalizer(Garbage collector)
Reference Handler
Signal Dispatcher
Attach Listener
...

All system level threads are daemon thread

System
Finalizer AttachLisener Reference Handler Signal Dispatcer mainthreadgroup

```
        mainthreadgroup
mainthread    Thread 0    subthreadgroup
```

ThreadGroup is a Java class present in java.lang package and it is direct child class of Object.

constructor:
1)
ThreadGroup g =new Thread(String groupname);
create a new threadgroup with the specified group name.
The parent of this new group is threadgroup of currently executing thread.

Ex.
```
ThreadGroup g1=new ThreadGroup("FirstGroup");
S.o.p(g1.getParent().getName());
```

2)ThreadGroup g = new ThreadGroup(ThreadGroup parent,String groupname);
create a new threadgroup with the specified group name. The parent of this new threadgroup is specified parent group.

Ex.
```
ThreadGroup g1=new ThreadGroup("FirstGroup");
S.o.p(g1.getParent().getName());//main
ThreadGroup g2=new ThreadGroup(g1,"secondgroup");
S.o.p(g2.getParent().getName());//FirstGroup
```

As per above progrom following hirarchy:

```
            system
       main
          firstgroup
        secondgroup
```

System -> main ->Firsgroup ->Secondgroup

Important methods of ThreadGroup class :
String getName()
int getMaxPriority()
void setMaxPriority(int p)
ThreadGroup getParent()
void list()
int activeCount()
int activeGroupCount()
int enumerate(Thread[] t)
int enumerate(ThreadGroup[] g)
boolean isDaemon()
void setDaemon(boolean b)
void interrupt()
void destroy()

String getName()
-> return name of the threadgroup

int getMaxPriority()
-> return max priority of threadgroup

void setMaxPriority(int p)
-> to set maximum priority of threadgroup
default max priority is 10
ThreadGroup getParent()
void list()
int activeCount()
int activeGroupCount()
int enumerate(Thread[] t)
int enumerate(ThreadGroup[] g)
boolean isDaemon()
void setDaemon(boolean b)
void interrupt()
void destroy()

Ex.

```
ThreadGroup g1 = new ThreadGroup("tg");
Thread t1=new Thread(g1,"Thread1");
Thread t2=new Thread(g1,"Thread2");
```

```
g1.setMaxPriority(3);
Thread t3=new Thread(g1,"Thread2");
g1.setMaxPriority(3);
Thread t3 = new Thread(g1,"Thread3");
S.o.p(t1.getPriority()); //5
S.o.p(t2.getPriority()); //5
S.o.p(t3.getPriority()); //3
```

default thread max priority - 5

threads in the threadgroup that already have higher priority won't be affected but for newly added thread this max priority is applicable

ThreadGroup getParent()
return parent group of current thread.

void list()
It prints information about threadgroup to the console.

int activeCount()
returns no of active threads present in the threadgroup.

int activeGroupCount()
returns no of active groups present in the currentthread group
int enumerate(Thread[] t)
To copy all active thread of this threadgroup into provided thread array.subgroup threadgroup is also considered.

```
Thread[] t=new Thread[10];
g.enumerate(t);


for(Thread t1:t)
{
   So.p(t1.getName());
}
```

int enumerate(ThreadGroup[] g)

to copy all active subthread groups into threadgroup array

boolean isDaemon()
to check whether the threadgroup is daemon or not

void setDaemon(boolean b)

void interrupt()
to interrupt all waiting/sleeping thread present in the threadgroup

void destroy()
to destroy threadgroup and its subthreadgroups

Ex.

```
class MyThread extends Thread
{
   MyThread(ThreadGroup g,String name)
   {
      super(g,name);
   }
  public void run()
  {
     System.out.println("child thread");
     try
     {
        Thread.sleep(5000);
     }
     catch(InterruptedException e){}
  }
}
class ThreadGroupDemo3
{
   psvm(String[] args) throws Exception
   {
      ThreadGroup pg=new ThreadGroup("ParentGroup");
      ThreadGroup cg=new ThreadGroup(pg,"ChildGroup");
      MyThread t1=new MyThread(pg,"childThread1");
      MyThread t2=new MyThread(pg,"childThread2");
      t1.start();
```

```
        t2.start();
        S.o.p(pg.activeCount());//2
        S.o.p(pg.activeGroupCount());//1 childgroup
        pg.list();
        Thread.sleep(10000);
        S.o.p(pg.activeCount());//0
        S.o.p(pg.activeGroupCount());//1
        pg.list();



             System
         main
    ParentGroup
 childgroup     childthread1    childthread2
```

write a program to display all active thread names belongs to system group and its child groups.

```
ThreadGroup system = Thread.currentThread().getThreadGroup().getParent();

Thread[] t=new Thread[System.activeCount()];
System.enumerate(t);
for(Thread t1:t)
{
   S.o.p(t1.getName()+" "+t1.isDaemon());
}
```

output ->

```
Finalizer -----true
SignalDispatcher------true
Reference Handler-----true
Attach Listener
main ---------false
```

---

java.util.concurrent package

Problem with synchronized:
performance
deadlock

to overcome these problems java.util.concurrent.Lock is introduced

```
if(l.tryLock())
{
   safe operation
}
else
{
  alternative
}
```

The 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 the lock which may create performance problem which 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 thread for a lock.
5)The synchronized keyword compusory we have to use either at method level or within a method and it is not possible to use across multiple methods.
To over come these problems sun people introduced java.util.concurrent.locks package in 1.5 version
It also provides several enhancement to the programmer to provide more control on concurrency.

Lock(I) ---implementation - ReentrantLock

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 operation than traditional implicit locks.
Important methods of Lock interface:

void lock()
-> we can use this method to acquire a lock. If lock is already available then immediately current thread will get that lock. If the lock is not available then it will wait until getting the lock. It is exactly same behavior of traditional synchronized keyword.

boolean tryLock()
->

To acquire the lock without waiting. If the lock is available then the thread acquires that lock and returns true. If the lock is not available then this method returns false and can continue its execution without waiting. In this case thread never be entered into waiting state.

```
if(l.tryLock())
 {
     perform safe operations
 }
 else
 {
  perform alternative operations
 }
```

boolean tryLock(long time,TimeUnit unit)
->
l.tryLock(1,TimeUnit.HOUR)
if lock is available then thread will get the lock and continue its execuetion. If the lock is not available then the thread will wait until spcified amount of time. still if the lock is not available then thread can continue its execution.

>javap java.util.concurrent.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))
 {
  ----
 }
```

void lockInterruptibly()
->

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

void unlock()
->
To releases lock.
To call this method compulsory current thread should be owner of the lock otherwise we will get runtime exception - IllegalMonitorStateException

ReentrantLock :

It is implementation class of Lock interface and direct child class of Object.

Reentrant means a thread can acquire same lock multiple times without any issue.
Internally Reentrant lock increment threads personal count whenever we call lock method and decrement count value whenever thread calls unlock method and lock will be released whenever count reaches zero.

constructor:
1)ReentrantLock l = new ReentrantLock();
creates an instance of ReentrantLock
2)ReentrantLock l =new ReentrantLock(boolean fairness);
creates ReentrantLock with the given fairness policy. If the fairness is true then longest waiting thread can acquire the lock if it is available i.e. it follows first come first serve policy. If fairness is false then which waiting thread will get the chance we can't expect. The default value for fairness is false.

Which of the following declarations are equal :
ReentrantLock l = new ReentrantLock();
ReentrantLock l = new ReentrantLock(true);
ReentrantLock l = new ReentrantLock(false);
All the above

1 and 3 are equal.

Important methods of ReentrantLock:
from Lock:
void lock()
boolean tryLock()
boolean tryLock(long l,TimeUnit t)
void lockInterruptibly()
void unlock()

int getHoldCount()
->returns number of holds on this lock by current thread

boolean isHeldByCurrentThread()
->returns true if and only if lock is hold by current thread

int getQueueLength()
->returns no of threads waiting for the lock

Collection getQueuedTherads()
-> it returns a collection of thread which are waiting to get the lock.

boolean hasQueuedThreads()
->returns true if any thread waiting to get the lock

boolean isLocked()
->returns true if the lock is acquired by some thread.

boolean isFair()
->returns true if the fairness policy is set with the true value

Thread getOwner()
->returns the thread which acquires the lock

Ex.

```
import java.util.concurrent.locks.*;


class ReentrantLockDemo
{
  psvm(String[] args)
  {
     ReentrantLock l= new ReentrantLock();
     l.lock();
     l.lock();
     S.o.p(l.isLocked()); //true
     S.o.p(l.isHeldByCurrentThread());//true
     S.o.p(l.getQueueLength());//0
```

```
      l.unlock();
      S.o.p(l.getHoldCount());//1
      S.o.p(l.isLocked());//true
      l.unlock();
      S.o.p(l.isLocked());//false
      S.o.p(l.isFair());//false
   }
}
```

Ex.

```
class Display
{
    public void wish(String name)
    {
      for(int i=0;i<10;i++)
      {
        System.out.println("Good morning");
        try
        {
           Thread.sleep(1000);
        }catch(InterruptedException ie)
         {}
        System.out.println(name);
      }
    }
}

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 MainApp
  {
    public static void main(String args)
    {
      Display d =new Display();
      MyThread t1=new MyThread(d,"Sachin");
      MyThread t2=new MyThread(d,"Virat");
      t1.start();
      t2.start();
    }
  }
```

Second Ex.

```
  class Display
  {
    public synchronized void wish(String name)
        // public void wish(String name)
     {
      for(int i=0;i<10;i++)
      {
        System.out.println("Good morning");
        try
        {
           Thread.sleep(1000);
        }catch(InterruptedException ie)
         {ie.printStackTrace();}
        System.out.println(name);
      }
     }
  }

  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 MainApp
  {
    public static void main(String[] args)
    {
      Display d =new Display();
      MyThread t1=new MyThread(d,"Sachin");
      MyThread t2=new MyThread(d,"Virat");
      t1.start();
      t2.start();
    }
  }
```

ThirdEx.

```
--------
class Display
{
     ReentrantLock l=new ReentrantLock();
    public void wish(String name)
    {
      l.lock();  //line 1
      for(int i=0;i<10;i++)
      {
        System.out.println("Good morning");
```

```
        try
        {
           Thread.sleep(1000);
        }catch(InterruptedException ie)
         {ie.printStackTrace();}
        System.out.println(name);
      }
      l.unlock(); //line 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 MainApp
{
   public static void main(String[] args)
   {
     Display d =new Display();
     MyThread t1=new MyThread(d,"Sachin");
     MyThread t2=new MyThread(d,"Virat");
     t1.start();
     t2.start();
   }
}
```

if we comment line1 and line 2 then the threads will be executed concurrently and we will get irregular output.

if we are not commenting line1 and 2 then the threads will be executed one by one and we will get regular output.

Demo program for tryLock()

```
-------------------------
import java.util.concurrent.locks.*;

class MyThread extends Thread
{
  static ReentrantLock l = new ReentrantLock();

  MyThread(String name)
  {
   super(name);
  }
  public void run()
  {
    if(l.tryLock())
     {
      System.out.println(Thread.currentThread().getName()+" ..get lock and performing operation");
     try
      {
      Thread.sleep(2000);
      }catch(InterruptedException ie){}
     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();
   }
```

```
}


=> output
First Thread ..get lock and performing operation
Second Thread...unable to get lock and hence performing alternative operations

package demo2;
import java.util.concurrent.locks.*;
import java.util.concurrent.*;

class MyThread extends Thread
{
 static ReentrantLock l= new ReentrantLock();
  MyThread(String name)
  {
    super(name);
  }
  public void run()
  {
     do
     {
       try{
        if(l.tryLock(5000,TimeUnit.MILLISECONDS))
        {
           System.out.println(Thread.currentThread().getName()+"...get lock");
           Thread.sleep(30000);
           System.out.println(Thread.currentThread().getName()+"...releases lock");
           l.unlock();
           break;
        }
        else
        {
           System.out.println(Thread.currentThread().getName()+"..unable to get lock and will try again");
        }
       }catch(Exception 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();
  }
}
```

ThreadPools(Executor Framework)

connection pool - to improve performance

Creating a new thread for every job may create performance and memory problem.
To overcome this we should go for ThreadPool.
ThreadPool is pool of already created threads ready to do our job. Java1.5 version introduces ThreadPool Framework to implement Thread pools.

ThreadPool framework also known as executor framework.

We can create a thread pool as follows:
ExecutorService service =Executors.new FixedThreadPool(3);

We can submit a runnable job by using submit method
service.submit(job);

We can shutdown executor service by using shutdown method
service.shutdown();

Ex.

```
import java.util.concurrent.*;

class PrintJob implements Runnable
{
  String name;
  PrintJob(String name)
  {
```

```
      this.name = name;
    }


    public void run()
    {
      System.out.println(name+"...Job started by thread: "+Thread.currentThread().getName());

      try
      {
        Thread.sleep(5000);
      }catch(InterruptedException e)
       {}
      System.out.println(name+"...Job completed by thread "+Thread.currentThread().getName());
     }
  }
  public class ExecutorDemo
  {
    public static void main(String[] args)
    {
      PrintJob[] jobs={new PrintJob("Sachin"),
                       new PrintJob("Virat"),
                       new PrintJob("Rohit"),
                       new PrintJob("Rahul"),
                       new PrintJob("Kedar"),
                       new PrintJob("Shikhar")};
   ExecutorService service = Executors.newFixedThreadPool(3);

   for(PrintJob job:jobs)
   {
     service.submit(job);
   }
    service.shutdown();
  }
  }
```

In the above example 3 threads are responsible to execute 6 jobs so that a single thread can be reused for multiple jobs.

While designing/developing web server and application server we can use threadpool concept.

Callable and Future:

In the case of Runnable job thread won't return anything after completing the job.
If a thread is required to return some result after execution then we should go for Callable.
Callable interface contains only one method - call()
 public Object call() throws Exception
If we submit Callable object to executor then after completing the job thread returns Future.
Future object can be used to retrieve the result from Callable job

```
import java.util.concurrent.*;

class MyCallable implements Callable
{
  int num;
  MyCallable(int num)
  {
    this.num=num;
  }
  public object call()throws Exception
  {
    System.out.println(Thread.currentThread.getName()+" is responsible to find sum of first "+num+"numbers");
    int sum=0;
    for(int i=1;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();
  }
}
```

vedik math = n*n+1/2

Differences between Runnable and Callable

Runnable:
run()
return type void
if there is possiblity of checked exception compulsory we should use try catch
introducted in 1.0

Callable:
call()
return type Object
can use throws Exception in method signature
introducted in 1.5

If a thread is not required to return anything after completing the job then we should go for Runnable.
If a thread require to return something after completing the job then we should go for Callable.
Runnable interface contains only one method - run()
Callable interface contains only one method - call()
Runnable job not require to return anything and hence return type of run method is void.
Callable job is required to return something and hence return type of call method is Object.

Within the run method if there is any chance of raising checked exception compulsory we should handle by using try catch because we can't use throws keyword for run method.
Within call method if there is any chance of raising checked exception we are not required to handle by using try catch because call method already throws

exception.

Runnable interface present in java.lang package
Callable interface present in java.util.concurrent package.

Runnable introduced in 1.0 version
Callable introduced in 1.5 version

ThreadLocal

ThreadLocal class provides thread local variable.
ThreadLocal calss maintains values per 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 manipulate 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 variable.
Ex.
Consider a servlet which invokes some business methods. We have requirement to generate a unique transaction id for each and every request and we have to pass this transaction id to the business methods, for this requirement we can use ThreadLocal to maintain separate transaction id for every request i.e. for every thread.

Note : ThreadLocal class introduced in 1.2 version and enhanced in 1.5 version.
ThreadLocal can be associated with thread scope. Total code which is executed by the thread has access to the corresponding ThreadLocal variable. A thread can access its own local variable and can't access other threads local variable.
Once thread enter into dead state all its localvariable are by default eligible for garbage collection.

constructor:
ThreadLocal tl=new ThreadLocal();
creates a thread local variable

Methods:
Object get()
-> returns the value of ThreadLocal variable associated with current thread.

Object initialValue()
-> returns initial value of ThreadLocal variable associated with current thread.

default implementation of this method returns null. To customize our own initial value we have to override this method.

void set(Object newValue)
-> To set new value

void remove()
-> to remove the value of ThreadLocal variable associated with current thread.
It is newly added method in 1.5 version.
After removal if we are trying to access it will be reinitialized once again by invoking its initialValue() method

Ex1.

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

Ex2.

Overriding of initialValue() method

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

```
    System.out.println(tl.get());//abc
  }
}
```

Ex.

```
---
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()
  {
   System.out.println(Thread.currentThread().getName()+"executing with customer id: "+tl.get());
 }
}

class ThreadLocalDemo2
{
  public static void main(String[] args)
  {

    CustomerThread c1=new CustomerThread("customer Thread-1");
    CustomerThread c2=new CustomerThread("customer Thread-2");
    CustomerThread c3=new CustomerThread("customer Thread-3");
    CustomerThread c4=new CustomerThread("customer Thread-4");

    c1.start();
    c2.start();
```

```
      c3.start();
      c4.start();

    }
}
```

In the above program for every customer thread a separate customerid will be maintained by ThreadLocal object.

ThreadLocal vs Inheritance

Parent thread ThreadLocal variable by default not available to the child thread. If we want to make parent thread's thread local value available to the child thread then we should go for InheritableThreadLocal class.
By default child thread value is exactly same as ParentThread's value but we can provide customized value for child thread by overriding child value method.

Constructor:
InheritableThreadLocal tl=new InheritableThreadLocal();

Methods:
4 methods from ThreadLocal class
InheritableThreadLocal is child class of ThreadLocal and hence all methods present in ThreadLocal by default available to InheritableThreadLocal. In addition to these methods it contains only one method:
public Object childValue(Object parentValue)

Ex.

```
class ParentThread extends Thread
{
  public static InheritableThreadLocal t1=new InheritableThreadLocal()
  {
    public Object childValue(Object p)
    {
      return "cc";
    }
  };
```

```
  public void run()
  {
    tl.set("PP");
    System.out.println("Parent thread value : "+tl.get());
    ChildThread ct=new ChildThread();
    ct.start();
  }
}

class ChildThread extends Thread
{
  public void run()
  {
    System.out.println("Child thread value : "+tl.get());
  }
}

class ThreadLocalDemo3
{
   public static void main(String[] args)
   {
     ParentThread pt=new ParentThread();
     pt.start();
   }
}
```

=>output

Parent Thread value PP

Child thread value CC

In the above program if we replace InheritableThreadLocal with ThreadLocal and if we are not overriding child value method then output is

Parent thread value : pp

Child thread value : null

In the above program if we are maintaining InheritableThreadLocal and if we are not overriding childValue method then output is

Parent thread value : PP

Child Thread value : PP