**Q1. Create and Run a Thread using Runnable Interface and Thread class.**

**Solution**

**package com.ttn.threads;**

**class RunnableThread implements Runnable{**

**@Override**

**public void run() {**

**System.*out*.println("Runnable Thread"+Thread.*currentThread*().getName());**

**}**

**}**

**class SimpleThreadDemo extends Thread**

**{**

**public void run()**

**{**

**System.*out*.println("Thread using Thread class"+Thread.*currentThread*().getName());**

**}**

**}**

**public class ThreadDemo {**

**public static void main(String[] args) {**

**System.*out*.println("In main");**

***//When we implement runnable, we can start it by passing it's object to thread class constructor***

**Thread thread1=new Thread(new RunnableThread());**

**thread1.start();**

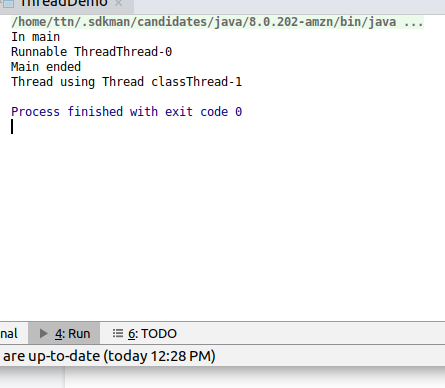
**Thread thread2=new SimpleThreadDemo();**

**thread2.start();**

**System.*out*.println("Main ended");**

**}**

**Output**



**Q2. Use sleep and join methods with thread.**

**Solution**

**package com.ttn.threads;**

**public class SleepAndJoinDemo {**

**public static void main(String[] args) throws InterruptedException {**

**System.*out*.println("Main started");**

**Thread thread1=new Thread(new RunnableThread(){**

**@Override**

**public void run()**

**{**

**for(int i=0;i<5;i++)**

**{**

**try {**

**Thread.*sleep*(1000);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**System.*out*.println("Thread-1----"+i);**

**}**

**}**

**});**

**Thread thread2=new Thread(new Runnable() {**

**@Override**

**public void run() {**

**for(int i=5;i<10;i++)**

**{**

**System.*out*.println("Thread2---"+i);**

**}**

**}**

**});**

**thread1.start();**

**thread2.start();**

**thread1.join();**

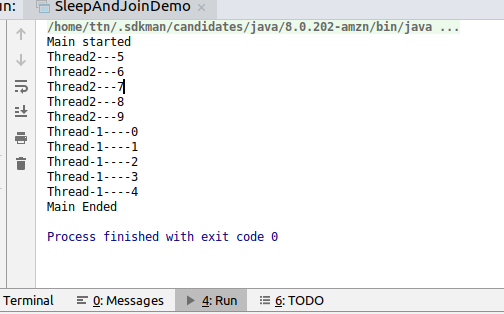
**thread2.join();**

**System.*out*.println("Main Ended");**

**}**

**}**

**Output**



**Q3. Use a singleThreadExecutor to submit multiple threads.**

**Solution**

**package com.ttn.threads;**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.concurrent.Executor;**

**import java.util.concurrent.ExecutorService;**

**import java.util.concurrent.Executors;**

**public class SingleThreadExecutorDemo {**

**public static void main(String[] args) {**

**ExecutorService executorService= Executors.*newSingleThreadExecutor*();**

**List<Runnable> threadList=new ArrayList<Runnable>();**

**threadList.add(new Runnable() {**

**@Override**

**public void run() {**

**System.*out*.println("Thread1");**

**}**

**});**

**threadList.add(new Runnable() {**

**@Override**

**public void run() {**

**System.*out*.println("Thread2");**

**}**

**});**

**threadList.add(new Runnable() {**

**@Override**

**public void run() {**

**System.*out*.println("Thread3");**

**}**

**});**

**try {**

**for(Runnable cur:threadList)**

**{**

**executorService.submit(cur);**

**}**

**} finally {**

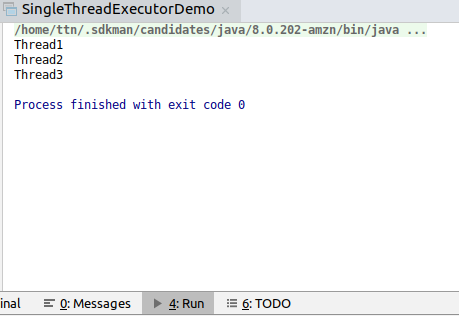
**executorService.shutdown();**

**}**

**}**

**}**

**Output**



**Q4. Try shutdown() and shutdownNow() and observe the difference.**

**Solution**

**ShutDownDemo**

**package com.ttn.threads;**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.concurrent.Executor;**

**import java.util.concurrent.ExecutorService;**

**import java.util.concurrent.Executors;**

**public class ShutDownDemo {**

**public static void main(String[] args) {**

**ExecutorService executorService= Executors.*newSingleThreadExecutor*();**

**List<Runnable> threadList=new ArrayList<Runnable>();**

**threadList.add(new Runnable() {**

**@Override**

**public void run() {**

**System.*out*.println("Thread1");**

**}**

**});**

**threadList.add(new Runnable() {**

**@Override**

**public void run() {**

**System.*out*.println("Thread2");**

**}**

**});**

**threadList.add(new Runnable() {**

**@Override**

**public void run() {**

**System.*out*.println("Thread3");**

**}**

**});**

**try {**

**for(Runnable cur:threadList)**

**{**

**executorService.submit(cur);**

**}**

**} finally {**

**executorService.shutdown();**

**}**

**System.*out*.println("Termination status"+executorService.isTerminated());**

**System.*out*.println("Shut down status"+executorService.isShutdown());**

**}**

**}**

**ShutDownNowDemo**

**package com.ttn.threads;**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.concurrent.ExecutorService;**

**import java.util.concurrent.Executors;**

**public class ShutDownNowDemo {**

**public static void main(String[] args) {**

**ExecutorService executorService= Executors.*newSingleThreadExecutor*();**

**List<Runnable> threadList=new ArrayList<Runnable>();**

**threadList.add(new Runnable() {**

**@Override**

**public void run() {**

**System.*out*.println("Thread1");**

**}**

**});**

**threadList.add(new Runnable() {**

**@Override**

**public void run() {**

**System.*out*.println("Thread2");**

**}**

**});**

**threadList.add(new Runnable() {**

**@Override**

**public void run() {**

**System.*out*.println("Thread3");**

**}**

**});**

**try {**

**for(Runnable cur:threadList)**

**{**

**executorService.submit(cur);**

**}**

**} finally {**

**executorService.shutdownNow();**

**}**

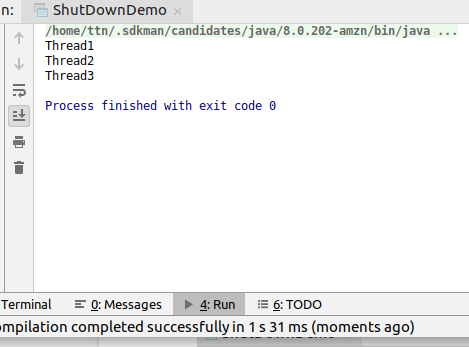
**System.*out*.println("Termination status"+executorService.isTerminated());**

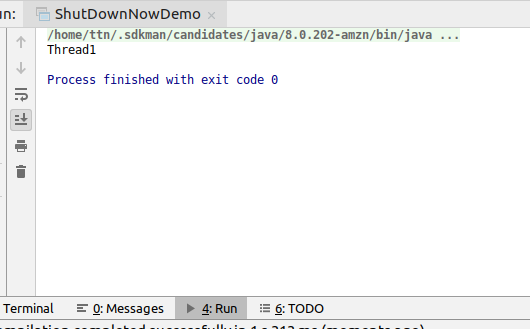
**System.*out*.println("Shut down status"+executorService.isShutdown());**

**}**

**}**

**Output**





**Difference**

**shutDownNow()--even submitted tasks are cancelled.**

**shutDown()---all submitted tasks gets executed.**

**Q5. Use isShutDown() and isTerminate() with ExecutorService**

**Solution**

**package** com.ttn.threads;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public class** ShutDownNowDemo {

**public static void** main(String[] args) {

ExecutorService executorService= Executors.*newSingleThreadExecutor*();

List<Runnable> threadList=**new** ArrayList<Runnable>();

threadList.add(**new** Runnable() {

@Override

**public void** run() {

System.***out***.println(**"Thread1"**);

}

});

threadList.add(**new** Runnable() {

@Override

**public void** run() {

System.***out***.println(**"Thread2"**);

}

});

threadList.add(**new** Runnable() {

@Override

**public void** run() {

System.***out***.println(**"Thread3"**);

}

});

**try** {

**for**(Runnable cur:threadList)

{

executorService.submit(cur);

}

} **finally** {

executorService.shutdownNow();

}

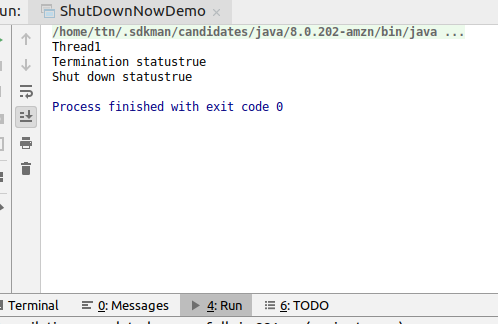
System.***out***.println(**"Termination status"**+executorService.isTerminated());

System.***out***.println(**"Shut down status"**+executorService.isShutdown());

}

}

**Output**



**Q6. Return a Future from ExecutorService by using callable and use get(), isDone(), isCancelled() with the Future object to know the status of task submitted.**

**Solution**

**package** com.ttn.threads;

**import** java.util.concurrent.\*;

**public class** FutureDemo {

**public static void** main(String[] args) **throws** InterruptedException, ExecutionException {

ExecutorService executorService= Executors.*newSingleThreadExecutor*();

Future<Float>floatFuture=executorService.submit(**new** Callable<Float>() {

@Override

**public** Float call() **throws** Exception {

**return** 4.5f;

}

});

executorService.shutdown();

**if**(floatFuture.isDone())

{

System.***out***.println(**"Result received from callable is "**+floatFuture.get());

}

**if**(floatFuture.isCancelled())

{

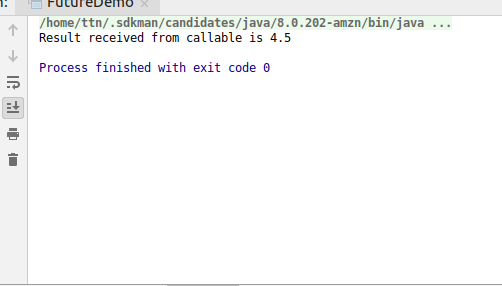
System.***out***.println(**"Task is cancelled"**);

}

}

}

**Output**



**Q7. Submit List of tasks to ExecutorService and wait for the completion of all the tasks.**

**Solution**

**package** com.ttn.threads;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.TimeUnit;

**public class** ExecutorServiceListAwaitTermination {

**public static void** main(String[] args) **throws** InterruptedException {

ExecutorService executorService= Executors.*newSingleThreadExecutor*();

List<Runnable> threadList=**new** ArrayList<Runnable>();

threadList.add(**new** Runnable() {

@Override

**public void** run() {

System.***out***.println(**"Thread1"**);

}

});

threadList.add(**new** Runnable() {

@Override

**public void** run() {

System.***out***.println(**"Thread2"**);

}

});

threadList.add(**new** Runnable() {

@Override

**public void** run() {

System.***out***.println(**"Thread3"**);

}

});

**for**(Runnable cur:threadList)

{

executorService.submit(cur);

}

executorService.shutdown();

executorService.awaitTermination(1000L, TimeUnit.***MILLISECONDS***);

**if**(executorService.isTerminated())

{

System.***out***.println(**"All tasks are terminated"**);

}

**else**

{

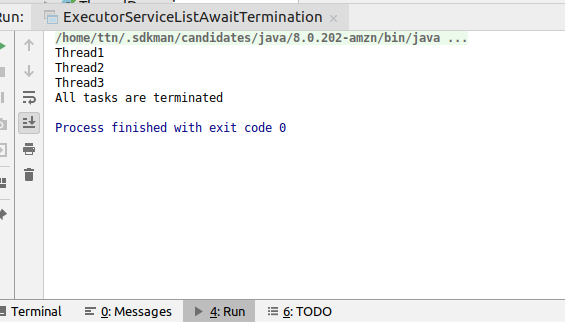
System.***out***.println(**"Some tasks are pending"**);

}

}

}

**Output**



**Q8. Schedule task using schedule(), scheduleAtFixedRate() and scheduleAtFixedDelay()**

**Solution**

package com.ttn.threads;import java.util.concurrent.\*;  
  
public class SchedulingDemo {  
 public static void main(String[] args) throws InterruptedException {  
 ScheduledExecutorService executorService= Executors.newSingleThreadScheduledExecutor();  
 executorService.schedule(new Runnable() {  
 @Override  
 public void run() {  
 System.out.println("Scheduled task after 2 second called");  
 }  
 },2,TimeUnit.SECONDS);  
 executorService.scheduleAtFixedRate(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.out.println("Scheduled at a fixed rate of 1 second");  
 }  
 },0,1,TimeUnit.SECONDS);  
 executorService.scheduleWithFixedDelay(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 Thread.sleep(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.out.println("Scheduled With Fix Delay");  
 }  
 },0,2,TimeUnit.SECONDS);  
 }  
}

**Q9. Increase concurrency with Thread pools using newCachedThreadPool() and newFixedThreadPool().**

**Solution**

**package com.ttn.threads;**

**import java.util.concurrent.ExecutorService;**

**import java.util.concurrent.Executors;**

**class Process implements Runnable**

**{**

**int processId;**

**public Process(int processId) {**

**this.processId = processId;**

**}**

**@Override**

**public void run() {**

**System.*out*.println("Runing process on thread: "+Thread.*currentThread*().getId()+"with id"+processId);**

**try {**

**Thread.*sleep*(2000);**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**System.*out*.println("Runing process on thread: "+Thread.*currentThread*().getId()+"End"+processId);**

**}**

**}**

**public class CachedThreadPoolDemo {**

**public static void main(String[] args) {**

***//using fixed pool***

**ExecutorService executorService= Executors.*newFixedThreadPool*(10);**

**for(int i=0;i<15;i++)**

**{**

**executorService.submit(new Thread(new Process(i)));**

**}**

**executorService.shutdown();**

***//using cached pool***

**ExecutorService cachedService=Executors.*newCachedThreadPool*();**

**for (int i=15;i<30;i++)**

**{**

**cachedService.submit(new Thread(new Process(i)));**

**}**

**cachedService.shutdown();**

**}**

**}**

**Difference**

When fixedThreadPool is used with a size 10; from process id 0-9 starts and when they end other process gets started. (Only 10 processes can run at a time, as thread size is fixed).

When cachedThreadPool is used, all 15 process starts at a time.(Threads are created as per request count).

**Q10. Use Synchronize method to enable synchronization between multiple threads trying to access method at same time.**

**Solution**

**package com.ttn.threads;**

**import java.util.concurrent.ExecutorService;**

**import java.util.concurrent.Executors;**

**public class SynchronizeMethodDemo {**

**int count=0;**

**synchronized void incrementCount()**

**{**

**count++;**

**}**

**synchronized void decrementCount(){count--;}**

**void worker1()**

**{**

**for(int i=0;i<5000;i++)**

**incrementCount();**

**for(int i=0;i<200;i++)**

**decrementCount();**

**}**

**void worker2()**

**{**

**for(int i=0;i<5000;i++)**

**decrementCount();**

**for(int i=0;i<200;i++)**

**incrementCount();**

**}**

**public static void main(String[] args) throws InterruptedException {**

**SynchronizeMethodDemo synchronizeMethodDemo=new SynchronizeMethodDemo();**

**ExecutorService executorService= Executors.*newSingleThreadExecutor*();**

**Thread thread1= new Thread(new Runnable() {**

**@Override**

**public void run() {**

**synchronizeMethodDemo.worker1();**

**}**

**});**

**Thread thread2=new Thread(new Runnable() {**

**@Override**

**public void run() {**

**synchronizeMethodDemo.worker2();**

**}**

**});**

**thread1.start();**

**thread2.start();**

**thread1.join();**

**thread2.join();**

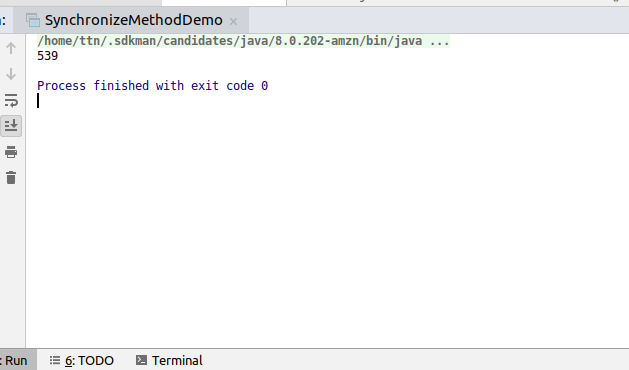
**System.*out*.println(synchronizeMethodDemo.count);**

**}**

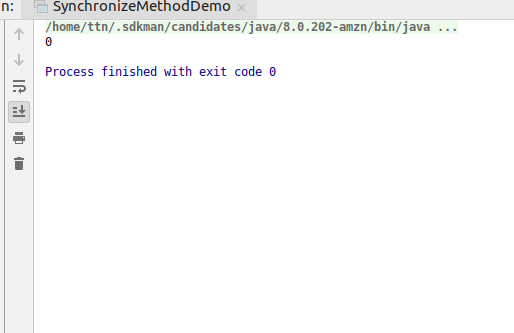
**}**

**Output**

**When increment and decrement were not synchronized(yielded incorrect reults)**



**When increment and decrement made synchronized**



**Q11. Use Synchronize block to enable synchronization between multiple threads trying to access method at same time.**

**Solution**

**package com.ttn.threads;**

**import java.util.concurrent.ExecutorService;**

**import java.util.concurrent.Executors;**

**public class SynchronizeBlockDemo {**

**int count = 0;**

**void incrementCount() {**

**synchronized (this) {**

**count++;**

**}**

**}**

**void decrementCount() {**

**synchronized (this) {**

**count--;**

**}**

**}**

**void worker1() {**

**for (int i = 0; i < 5000; i++)**

**incrementCount();**

**for (int i = 0; i < 200; i++)**

**decrementCount();**

**}**

**void worker2() {**

**for (int i = 0; i < 5000; i++)**

**decrementCount();**

**for (int i = 0; i < 200; i++)**

**incrementCount();**

**}**

**public static void main(String[] args) throws InterruptedException {**

**SynchronizeMethodDemo synchronizeMethodDemo = new SynchronizeMethodDemo();**

**ExecutorService executorService = Executors.*newSingleThreadExecutor*();**

**Thread thread1 = new Thread(new Runnable() {**

**@Override**

**public void run() {**

**synchronizeMethodDemo.worker1();**

**}**

**});**

**Thread thread2 = new Thread(new Runnable() {**

**@Override**

**public void run() {**

**synchronizeMethodDemo.worker2();**

**}**

**});**

**thread1.start();**

**thread2.start();**

**thread1.join();**

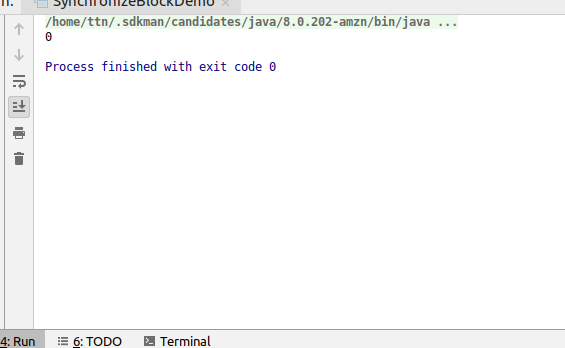
**thread2.join();**

**System.*out*.println(synchronizeMethodDemo.count);**

**}**

**}**

**Output**



**Q12. Use Atomic Classes instead of Synchronize method and blocks.**

**Solution**

**package** com.ttn.threads;

**import** java.util.concurrent.atomic.AtomicIntegerArray;

**public class** AtomicClassDemo {

AtomicIntegerArray **atomicIntegerArray**=**new** AtomicIntegerArray(10);

**void** worker1()

{

**for**(**int** i=0;i<9;i++)

{

**atomicIntegerArray**.set(i,i+1);

}

}

**void** worker2()

{

**for**(**int** i=0;i<9;i++)

{

**atomicIntegerArray**.set(i,i+2);

}

}

**public static void** main(String[] args) **throws** InterruptedException {

AtomicClassDemo atomicClassDemo=**new** AtomicClassDemo();

Thread thread1=**new** Thread(**new** Runnable() {

@Override

**public void** run() {

atomicClassDemo.worker1();

}

});

Thread thread2=**new** Thread(**new** Runnable() {

@Override

**public void** run() {

atomicClassDemo.worker2();

}

});

thread1.start();

thread2.start();

thread1.join();

thread2.join();

**for**(**int** i=0;i<atomicClassDemo.**atomicIntegerArray**.length();i++)

{

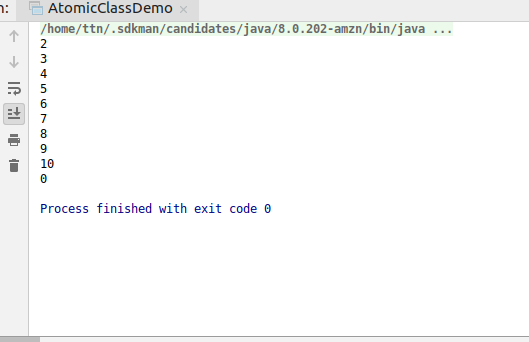
System.***out***.println(atomicClassDemo.**atomicIntegerArray**.get(i));

}

}

}

**Output**



**Q13. Coordinate 2 threads using wait() and notify().**

**Solution**

**package com.ttn.threads;**

**public class WaitAndNotifyDemo {**

**public void waitForNotify() throws InterruptedException {**

**System.*out*.println("Wait for notify started");**

**synchronized (this)**

**{**

**wait();**

**}**

**System.*out*.println("Got notification!I am happy to end");**

**}**

**public void notifyWaitingMethod()**

**{**

**System.*out*.println("I will perform my work and notify to continue");**

**System.*out*.println("Mywork done");**

**synchronized (this)**

**{**

**notify();**

**}**

**}**

**public static void main(String[] args) throws InterruptedException {**

**WaitAndNotifyDemo waitAndNotifyDemo=new WaitAndNotifyDemo();**

**Thread thread1=new Thread(new Runnable() {**

**@Override**

**public void run() {**

**try {**

**waitAndNotifyDemo.waitForNotify();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**});**

**Thread thread2=new Thread(new Runnable() {**

**@Override**

**public void run() {**

**waitAndNotifyDemo.notifyWaitingMethod();**

**}**

**});**

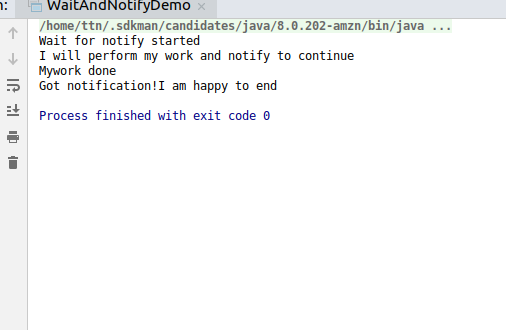
**thread1.start();**

**thread2.start();**

**}**

**}**

**Output**



**Q14. Coordinate multiple threads using wait() and notifyAll()**

**Solution**

**package com.ttn.threads;**

**public class WaitAndNotifyAllDemo {**

**public void waitForNotify1() throws InterruptedException {**

**System.*out*.println("Wait for notify started");**

**synchronized (this)**

**{**

**wait();**

**}**

**System.*out*.println("Got notification!I am happy to end");**

**}**

**public void waitForNotify2() throws InterruptedException {**

**System.*out*.println("I am also waiting for notification :(");**

**synchronized (this)**

**{**

**wait();**

**}**

**System.*out*.println(":)Got notification!I am happy to end");**

**}**

**public void notifyAllWaitingMethod()**

**{**

**System.*out*.println("I will perform my work and notify to continue");**

**System.*out*.println("Mywork done");**

**synchronized (this)**

**{**

**notifyAll();**

**}**

**}**

**public static void main(String[] args) throws InterruptedException {**

**WaitAndNotifyAllDemo waitAndNotifyAllDemo=new WaitAndNotifyAllDemo();**

**Thread thread1=new Thread(new Runnable() {**

**@Override**

**public void run() {**

**try {**

**waitAndNotifyAllDemo.waitForNotify1();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**});**

**Thread thread2=new Thread(new Runnable() {**

**@Override**

**public void run() {**

**try {**

**waitAndNotifyAllDemo.waitForNotify2();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**});**

**Thread thread3=new Thread(new Runnable() {**

**@Override**

**public void run() {**

**waitAndNotifyAllDemo.notifyAllWaitingMethod();**

**}**

**});**

**thread1.start();**

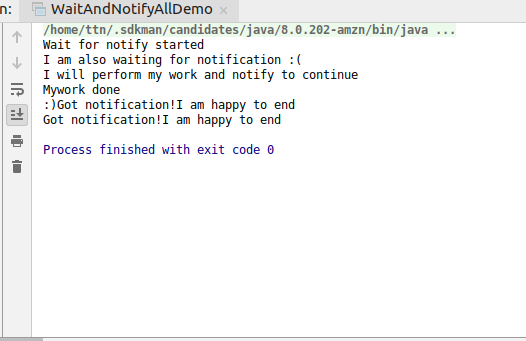
**thread2.start();**

**thread3.start();**

**}**

**}**

**Output**



**Q15. Use Reentract lock for coordinating 2 threads with signal(), signalAll() and wait().**

**Solution**

**package com.ttn.threads;**

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

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

**public class ReentractLockDemo {**

**Lock lock=new ReentrantLock(true);**

**int count=0;**

**void incrementCount()**

**{**

**lock.lock();**

**count++;**

**lock.unlock();**

**}**

**void decrementCount()**

**{**

**lock.lock();**

**count--;**

**lock.unlock();**

**}**

**void worker1()**

**{**

**for(int i=0;i<500;i++)**

**incrementCount();**

**}**

**void worker2()**

**{**

**for (int i=0;i<200;i++)**

**decrementCount();**

**}**

**public static void main(String[] args) throws InterruptedException {**

**ReentractLockDemo reentractLockDemo=new ReentractLockDemo();**

**Thread thread1=new Thread(new Runnable() {**

**@Override**

**public void run() {**

**reentractLockDemo.incrementCount();**

**}**

**});**

**Thread thread2=new Thread(new Runnable() {**

**@Override**

**public void run() {**

**reentractLockDemo.decrementCount();**

**}**

**});**

**thread1.start();**

**thread1.join();**

**thread2.start();**

**thread2.join();**

**System.*out*.println(reentractLockDemo.count);**

**}**

**}**

**Q16. Create a deadlock and Resolve it using tryLock().**

**Solution**

**package com.ttn.threads;**

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

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

**public class DeadlockSolutionDemo {**

**Lock lock1 = new ReentrantLock(true);**

**Lock lock2 = new ReentrantLock(true);**

**public void acquireLock(Lock lock1, Lock lock2) {**

**boolean acquireLock1 = lock1.tryLock();*//Checking if lock1 is able to obtain lock***

**boolean acquireLock2 = lock2.tryLock();*//Checking if lock2 is able to obtain lock***

***//both have obtained lock***

**if (acquireLock1 && acquireLock2) {**

**return;**

**}**

***//lock1 is acquired***

**if (acquireLock1) {**

**lock1.unlock();**

**}**

***//lock2 is acquired***

**if (acquireLock2) {**

**lock2.unlock();**

**}**

**}**

**public void worker1() {**

***//try acquiring lock in order lock1-->lock2***

**acquireLock(lock1, lock2);**

**System.*out*.println("lock 1 worker 1");**

**System.*out*.println("lock 2 worker 1");**

**lock2.unlock();**

**lock1.unlock();**

**}**

**public void worker2() {**

***//reversing the order of acquirelock lock2--->lock1***

***//if both locks will be granted at same time, it will be a deadlock, tryLock is preventing it***

**acquireLock(lock2, lock1);**

**System.*out*.println("lock 1 worker 2");**

**System.*out*.println("lock 2 worker 2");**

**lock1.unlock();**

**lock2.unlock();**

**}**

**public static void main(String[] args) throws InterruptedException {**

**DeadlockSolutionDemo deadlockSolutionDemo = new DeadlockSolutionDemo();**

**Thread thread1 = new Thread(new Runnable() {**

**@Override**

**public void run() {**

**deadlockSolutionDemo.worker1();**

**}**

**});**

**Thread thread2 = new Thread(new Runnable() {**

**@Override**

**public void run() {**

**deadlockSolutionDemo.worker2();**

**}**

**});**

**thread1.start();**

**thread2.start();**

**thread1.join();**

**thread2.join();**

**}**

**}**

**Output**

