Interrupting a Thread:

|  |
| --- |
| 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. Let's first see the methods provided by the Thread class for thread interruption. |

## **The 3 methods provided by the Thread class for interrupting a thread**

|  |
| --- |
| * **public void interrupt()** * **public static boolean interrupted()** * **public boolean isInterrupted()** |

## **Example of interrupting a thread that stops working**

|  |
| --- |
| In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception. |

**class** TestInterruptingThread1 **extends** Thread{

**public** **void** run(){

**try**{

Thread.sleep(1000);

System.out.println("task");

}**catch**(InterruptedException e){

**throw** **new** RuntimeException("Thread interrupted..."+e);

}

}

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

TestInterruptingThread1 t1=**new** TestInterruptingThread1();

t1.start();

**try**{

t1.interrupt();

}**catch**(Exception e){System.out.println("Exception handled "+e);}

}

}

Output:Exception in thread-0

java.lang.RuntimeException: Thread interrupted...

java.lang.InterruptedException: sleep interrupted

at A.run(A.java:7)

## **Example of interrupting a thread that doesn't stop working**

|  |
| --- |
| In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working. |

**class** TestInterruptingThread2 **extends** Thread{

**public** **void** run(){

**try**{

Thread.sleep(1000)

System.out.println("task");

}**catch**(InterruptedException e){

System.out.println("Exception handled "+e);

}

System.out.println("thread is running...");

}

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

TestInterruptingThread2 t1=**new** TestInterruptingThread2();

t1.start();

t1.interrupt();

}

}

Output:Exception handled

java.lang.InterruptedException: sleep interrupted

thread is running...

class Thread1 extends Thread{

public void run(){

System.out.println("Task in run");

if(isInterrupted()){

System.out.println("Interrupted");

}

try{

Thread.sleep(2000);

}

catch(InterruptedException e){

System.out.println(e);

}

System.out.println("Thread is Running");

}

}

class Interrupt1{

public static void main(String args[]){

Thread1 t1 = new Thread1();

t1.start();

t1.interrupt();

}

}

Output

Task in run

Interrupted

java.lang.InterruptedException: sleep interrupted

Thread is Running

# Reentrant Monitor in Java

**Java monitors are reentrant** means java thread can reuse the same monitor for different synchronized methods if method is called from the method.

#### **Advantage of Reentrant Monitor**

It eliminates the possibility of single thread deadlocking

Let's understand the java reentrant monitor by the example given below:

In this class, m and n are the synchronized methods. The m() method internally calls the n() method.

Now let's call the m() method on a thread. In the class given below, we are creating thread using annonymous class.

class Reentrant {

public synchronized void m() {

n();

System.out.println("this is m() method");

}

public synchronized void n() {

System.out.println("this is n() method");

}

}

public class ReentrantExample{

public static void main(String args[]){

final Reentrant re=new Reentrant();

Thread t1=new Thread(){

public void run(){

re.m();//calling method of Reentrant class

}

};

t1.start();

}}

Output: this is n() method

this is m() method