**Multithreading:**

Process of executing multiple threads simultaneously

A thread is a lightweight sub process, it is a path of execution. Threads are independent, If there occurs exception in one thread it doesn’t affect other threads.

Life cycle of Thread (Thread states)

New – whenever new thread is created it is always new state

Active – When a thread invokes the start() method, it moves from new state to active state. The active states contain two states within it , runnable and running

1. Runnable: A thread that is ready to run is then moved to runnable state.
2. Running: When thread gets the CPU, it moves from runnable to running state.

Blocked/Waiting – Whenever a thread is inactive for a sometime then either the thread is in blocked state or is in waiting state

Timed Waiting – waiting for a certain period of time

Terminated – Thread reaches termination state when a thread has finished its job or Abnormal termination (some unusual events such as unhandled exception or other)

Two ways to create thread – By extending Thread class, By implementing Runnable interface

Thread class – it provides constructors and methods to create and perform operations on thread.

Commonly used constructors

Thread()

Thread(String name)

Thread(Runnable r)

Thread(Runnable r, String name)

Commonly used methods

public void run() – to perform action of thread

public void start() – execution of thread

public void sleep(long milliseconds) – thread to sleep for specified time

public void join() – waits for thread to die

public void join(long milliseconds) – waits for thread to die for specified time

public int getPriority() – returns the priority of thread

public int setPriority(int priority) – change the priority of thread

public String getName() – returns the name of thread

public String setName(String name) – change the name of thread

public Thread currentThread() – returns the reference of current thread

public int getId() - gets the id of thread

public getState() – returns the state if thread

public Boolean isAlive() – tests if thread is alive

public void yield() – causes the currently executing thread object to temporarily pause and allow other threads to execute

public void suspend() – used to suspend the thread

public void resume() – resume the suspended thread

public void stop() – used to stop the thread

public boolean isDaemon() – tests if thread is daemon thread

public void setDaemon(boolean b) – marks the thread as daemon or user thread

public void interrupt() – interrupts the thread

public Boolean isInterrupted() – tests if thread has been interrupted

public static Boolean interrupted() – test it current thread has been interrupted

Runnable interface – should be implemented by any class whose instances are intended to be executed by Thread. It has only one method.

Public void run() – perform action of thread

Example: By extending Thread class

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

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

}

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

// **TODO** Auto-generated method stub

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.start();

}

}

Example: By implementing Runnable interface

**package** com.lokesh;

**public** **class** HelloWorld2 **implements** Runnable {

**public** **void** run() {

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

}

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

// **TODO** Auto-generated method stub

HelloWorld2 hw2 = **new** HelloWorld2();

Thread th1 = **new** Thread(hw2); //using constructor Thread(Runnable r)

th1.start();

}

}

Example: Using constructor Thread(String name)

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

// **TODO** Auto-generated method stub

Thread th1 = **new** Thread("This is thread ABC");

th1.start();

String str =th1.getName();

System.***out***.println(str);

}

}

Example: Using constructor Thread(Runnable ,String name)

**package** com.lokesh;

**public** **class** HelloWorld2 **implements** Runnable {

**public** **void** run() {

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

}

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

// **TODO** Auto-generated method stub

Runnable hw2 = **new** HelloWorld2();

Thread th1 = **new** Thread(hw2, "this is thread 123");

th1.start();

String str =th1.getName();

System.***out***.println(str);

}

}

Thread scheduler – A component of Java decides which thread to run or execute and which thread to wait. In Java, a thread is only chosen by a thread scheduler if it is in runnable state. If there are more than one thread in runnable state, it is up to thread scheduler to pick one of the threads and ignore the other ones. There are two factors for scheduling a thread i.e. Priority and Time of Arrival

Priority – Priority of each thread lies between 1 to 10. If a thread has higher priority, it means that thread has got better chance of getting picked up by thread scheduler

Time of Arrival – Suppose two threads of same priority enter the runnable state, then priority cannot be a factor to pick a thread from these two threads. In such case, arrival time of thread is considered by thread scheduler. A thread that arrived first gets the preference over other threads

The threads are basically first come first serve scheduling

Time slicing scheduling – Usually, the first come first serve is non-preemptive, which is bad as it may lead to infinite blocking. To avoid that some time-slices are provided to threads so that after some time, the running thread has to given up the CPU. Thus other waiting threads also get time to run their job.

For example time slicing of 2 seconds for each thread.

Thread.sleep() :

The method sleep() is used to halt the working of a thread for a given amount of time.

Two variants of sleep() method. One accepts one argument while other accepts two arguments

Syntax:

Public static void sleep(long millisecs) throws Interrupted Exception

Public static void sleep(long millisecs, int m) throws Interrupted Exception

millisecs – time in milliseconds is represented by parameter millsecs. Duration for which thread will sleep

The time cannot be negative

m: it shows the additional time up to which programmer wants the thread to be in sleep state. Range is 0 to 999999

Example: On custom Thread

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

**for**(**int** i=1;i<=5;i++) {

**try** {

Thread.*sleep*(500);

}**catch**(InterruptedException e) {

System.***out***.println(e);

}

System.***out***.println(i);

}

}

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

// **TODO** Auto-generated method stub

HelloWorld2 hw2 = **new** HelloWorld2();

HelloWorld2 hw3 = **new** HelloWorld2();

hw2.start();

hw3.start();

}

}

Example: on Main Thread

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

// **TODO** Auto-generated method stub

**for**(**int** i=1;i<=5;i++) {

**try** {

Thread.*sleep*(1000);

}**catch**(InterruptedException e) {

System.***out***.println(e);

}

System.***out***.println(i);

}

}

}

After starting a thread we cannot start it again. Thread will run for first time and for second time it throws an exception

Example:

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

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

}

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

// **TODO** Auto-generated method stub

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.start();

hw2.start();

}

}

If we start Java run() method directly instead of start() method, each thread starts in separate call stack

Invoking the run() method from main thread, the run() method goes onto current call stack rather than new call stack

Example:

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

**for**(**int** i=1;i<=5;i++) {

**try** {

Thread.*sleep*(500);

}**catch**(InterruptedException e) {

System.***out***.println(e);

}

System.***out***.println(i);

}

}

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

// **TODO** Auto-generated method stub

HelloWorld2 hw2 = **new** HelloWorld2();

HelloWorld2 hw3 = **new** HelloWorld2();

hw2.run();

hw3.run();

}

}

The output for above program is 1 2 3 4 5 1 2 3 4 5

So it means since run() method uses current call stack it is treating as normal object not thread object

Join() method: it permits one thread to wait until the other thread finish its execution.

Suppose hw2, hw3 and hw4 are objects of class thread. hw2 whose thread is doing execution currently, the hw.join(); statement ensures that hw2 is finished before the program does the execution of next statement i.e. hw3 object . When there are more than one thread invoking the join() method, then it leads to overloading on the join() method that permits the programmer to mention waiting period

Syntax –

public final void join() throws InterruptedException

public final synchronized void join(long millisecs) throws InterruptedException

public final synchronized void join(long millisecs, int nanos) throws InterruptedException

Example:

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

**for**(**int** i=0;i<2;i++) {

**try** {

Thread.*sleep*(500);

System.***out***.println("current thread name is "+Thread.*currentThread*().getName());

}**catch**(Exception e) {

System.***out***.println(e);

}

System.***out***.println(i);

}

}

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

// **TODO** Auto-generated method stub

HelloWorld2 hw2 = **new** HelloWorld2();

HelloWorld2 hw3 = **new** HelloWorld2();

HelloWorld2 hw4 = **new** HelloWorld2();

//thread hw2 starts

hw2.start();

//starting the second thread after

//when first thread hw2 ended or died

**try** {

System.***out***.println("the main program current thread is "+Thread.*currentThread*().getName());

//invoking join() method means it will ensure

//hw2 thread is finished before hw3 execution starts

hw2.join();

}**catch**(Exception e) {

System.***out***.println(e);

}

//thread hw3 starts

hw3.start();

//starting the third thread after when

//first thread hw2 ended or died

**try** {

System.***out***.println("the main program current thread is "+Thread.*currentThread*().getName());

//invoking join() method means it will ensure

//hw3 thread is finished before hw4 execution starts

hw3.join();

}**catch**(Exception e) {

System.***out***.println(e);

}

hw4.start();

}

}

Naming Thread: Thread class has methods to change (th.setName()) and get the name (th.getName()) of thread.

By default thread names are thread-0, thread-1 and so on.

Example:

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

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

}

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

// **TODO** Auto-generated method stub

HelloWorld2 hw2 = **new** HelloWorld2();

HelloWorld2 hw3 = **new** HelloWorld2();

System.***out***.println(hw2.getName());

System.***out***.println(hw3.getName());

hw2.setName("ABC");

hw3.setName("XYZ");

System.***out***.println(hw2.getName());

System.***out***.println(hw3.getName());

}

}

Priority of Thread: Each thread has a priority. Priority are from 1 to 10. Thread scheduler schedules the threads according to priority (like first come first serve…). Programmer can also assign the priorities of a thread explicitly in program

Public final int getPriority() - returns priority of given thread

Th.getPriority();

Public final int setPriority() - updates or assigns priority of thread to new peiority

Th.setPriority(2);

3 constants in Thread class

Public static int MIN\_PRIORITY ----value is 1

Public static int NORM\_PRIORITY ---value is 5

Public static int MAX\_PRIORITY ---value is 10

By default it is 5

Example:

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

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

}

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

// **TODO** Auto-generated method stub

HelloWorld2 hw2 = **new** HelloWorld2();

HelloWorld2 hw3 = **new** HelloWorld2();

HelloWorld2 hw4 = **new** HelloWorld2();

System.***out***.println(hw2.getPriority());

System.***out***.println(hw3.getPriority());

System.***out***.println(hw4.getPriority());

hw2.setPriority(6);

hw3.setPriority(2);

hw4.setPriority(9);

System.***out***.println(hw2.getPriority());

System.***out***.println(hw3.getPriority());

System.***out***.println(hw4.getPriority());

System.***out***.println("currently executin the thread "+Thread.*currentThread*().getName());

System.***out***.println("priority of thread "+Thread.*currentThread*().getPriority());

Thread.*currentThread*().setPriority(10);

System.***out***.println("priority of thread "+Thread.*currentThread*().getPriority());

hw2.setPriority(***MIN\_PRIORITY***);

hw3.setPriority(***NORM\_PRIORITY***);

hw4.setPriority(***MAX\_PRIORITY***);

System.***out***.println(hw2.getPriority());

System.***out***.println(hw3.getPriority());

System.***out***.println(hw4.getPriority());

}

}

Daemon Thread: is a service provider thread that provides service to user thread (for background supporting tasks). Its life depends upon mercy of user threads i.e. when all user threads dies, JVM terminates this thread automatically. It is a low priority thread

There are many daemon threads running automatically example Finalizer…etc

Example:

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

**if**(Thread.*currentThread*().isDaemon()) {

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

}**else** {

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

}

}

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

// **TODO** Auto-generated method stub

HelloWorld2 hw2 = **new** HelloWorld2();

HelloWorld2 hw3 = **new** HelloWorld2();

HelloWorld2 hw4 = **new** HelloWorld2();

//hw2 is daemon thread

hw2.setDaemon(**true**);

hw2.start();

hw3.start();

hw4.start();

}

}

Thread Pool: represents a group of worker threads that waiting for the job and reused many times

In thread pool, a group of fixed size threads are created. A thread from thread pool is pulled out and assigned a job by service provider. After completion of job the thread is contained in thread pool again.

Example: Below is an example where threads are re-used since for loop has 10 times to run. The 5 fixed threads were re-used to run all 10 instances

**package** com.lokesh;

**public** **class** HelloWorld **implements** Runnable {

**private** String S1;

**public** HelloWorld(String S2) {

**this**.S1 =S2;

}

**public** **void** run() {

System.***out***.println(Thread.*currentThread*().getName()+" message "+S1);

//calls the method process S1 to sleep the thread for 2 seconds

processS1();

//prints the thread name

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

}

**private** **void** processS1() {

**try** {

Thread.*sleep*(2000);

}**catch**(InterruptedException e) {

System.***out***.println(e);

}

}

}

**package** com.lokesh;

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

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

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

**public** **class** HelloWorld2 {

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

// **TODO** Auto-generated method stub

//creating a pool of 5 threads

ExecutorService exec=Executors.*newFixedThreadPool*(5);

**for**(**int** i=1;i<10;i++) {

Runnable r1=**new** HelloWorld(" "+i);

//calling the execute method ExecutorService

exec.execute(r1);

}

//terminating all the threads

exec.shutdown();

**while**(!exec.isTerminated()) {

}

**if**(exec.isTerminated()) {

System.***out***.println("all threads are terminated");

}**else** {

System.***out***.println("all threads are not yet terminated");

}

}

}

Thread Group: represent group of threads. It can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread groups parent thread group or any other thread group.

Example: In below we get one thread group “tg“ and it has three threads “one’ , “two” and “three”

**package** com.lokesh;

**public** **class** HelloWorld2 **implements** Runnable {

**public** **void** run() {

System.***out***.println(Thread.*currentThread*().getName());

}

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

HelloWorld2 hw2 = **new** HelloWorld2();

ThreadGroup tg = **new** ThreadGroup("Parent threadGroup");

Thread t1=**new** Thread(tg, hw2,"one");

t1.start();

Thread t2=**new** Thread(tg, hw2,"two");

t2.start();

Thread t3=**new** Thread(tg, hw2,"three");

t3.start();

System.***out***.println("Thread group name is "+tg.getName());

tg.list();

}

}

Shutdown Hook: It facilitates programmers to add some code (or some operation) that has to be run when JVM is shutting down

Example:

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

System.***out***.println("shut down hook task completed");

}

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

Runtime r=Runtime.*getRuntime*();

r.addShutdownHook(**new** HelloWorld2());

System.***out***.println("now main is sleeping");

**try** {

Thread.*sleep*(3000);

}**catch**(Exception e) {

}

}

}

Garbage collection: garbage means unreferenced objects. Garbage collection is a process of reclaiming the runtime unused memory automatically. In other words it is a way to destroy unused objects

The object can be unreferenced in three ways

By nullying the reference

HelloWorld2 hw2 =new HelloWorld2();

Hw2 =null;

By assigning a reference to another

HelloWorld2 hw2 =new HelloWorld2();

HelloWorld2 hw3 =new HelloWorld2();

hw2 = hw3;

By anonymous object

new HelloWorld2();

Finalize method – this method is invoked each time before the object is garbage collected. This method is used to perform cleanup processing

gc() method – used to invoke the garbage collector to perform cleanup processing

Example:

**package** com.lokesh;

**public** **class** HelloWorld2 {

**public** **void** finalize() {

System.***out***.println("Object is garbage collected");

}

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

HelloWorld2 hw2 =**new** HelloWorld2();

HelloWorld2 hw3 =**new** HelloWorld2();

hw2 =**null**;

hw3 =**null**;

System.*gc*();

}

}

Runtime class: used to interact with java runtime environment.

Example: Below is an example to open notepad

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

Runtime.*getRuntime*().exec("notepad");

}

}

Synchronization: is the capability to control the access of multiple threads to any shared resource

It is used mainly to prevent thread interference and to prevent consistency problem

Two types of synchronization –

Mutual Exclusive – Synchronized method, Synchronized block, Static Synchronization

Cooperation (inter-thread communicarion)

Basically synchronization means we are locking the object for all shared resources so that one shared resources use one after other rather than using parallel

Synchronized method –

Below is an example without synchronization due to which since hw2 and hw3 instances started the output will be generated in mix like 5 100 200 10 15 and so on

**package** com.lokesh;

**public** **class** HelloWorld {

//not synchonized

**public** **void** method1(**int** n) {

**for**(**int** i=1;i<=5;i++) {

System.***out***.println(n\*i);

**try** {

Thread.*sleep*(300);

}**catch**(Exception e) {

System.***out***.println(e);

}

}

}

}

**package** com.lokesh;

**public** **class** HelloWorld1 **extends** Thread {

HelloWorld hw;

**public** HelloWorld1(HelloWorld hw) {

**this**.hw =hw;

}

**public** **void** run() {

hw.method1(5);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

HelloWorld hw;

**public** HelloWorld2(HelloWorld hw) {

**this**.hw =hw;

}

**public** **void** run() {

hw.method1(100);

}

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

HelloWorld hw =**new** HelloWorld();

HelloWorld1 hw2 =**new** HelloWorld1(hw);

HelloWorld2 hw3 =**new** HelloWorld2(hw);

hw2.start();

hw3.start();

}

}

Below is an example with synchronization due to which since hw2 and hw3 instances started the output will be generated of first hw2’s and later hw3’s like 5 10 15….100 200 300…

Just added synchronized in front of method

**package** com.lokesh;

**public** **class** HelloWorld {

//synchronized

**public** **synchronized** **void** method1(**int** n) {

**for**(**int** i=1;i<=5;i++) {

System.***out***.println(n\*i);

**try** {

Thread.*sleep*(300);

}**catch**(Exception e) {

System.***out***.println(e);

}

}

}

}

**package** com.lokesh;

**public** **class** HelloWorld1 **extends** Thread {

HelloWorld hw;

**public** HelloWorld1(HelloWorld hw) {

**this**.hw =hw;

}

**public** **void** run() {

hw.method1(5);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

HelloWorld hw;

**public** HelloWorld2(HelloWorld hw) {

**this**.hw =hw;

}

**public** **void** run() {

hw.method1(100);

}

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

HelloWorld hw =**new** HelloWorld();

HelloWorld1 hw2 =**new** HelloWorld1(hw);

HelloWorld2 hw3 =**new** HelloWorld2(hw);

hw2.start();

hw3.start();

}

}

Synchronized block – used to perform synchronization on any specific resource of the method

Suppose if we have 50 lines of code in our method, but we want to synchronize only 5 lines in such cases we can use synchronize block

**package** com.lokesh;

**public** **class** HelloWorld {

**public** **void** method1(**int** n) {

//synchronized block

**synchronized**(**this**) {

**for**(**int** i=1;i<=5;i++) {

System.***out***.println(n\*i);

**try** {

Thread.*sleep*(300);

}**catch**(Exception e) {

System.***out***.println(e);

}

}

} //till here

}

}

**package** com.lokesh;

**public** **class** HelloWorld1 **extends** Thread {

HelloWorld hw;

**public** HelloWorld1(HelloWorld hw) {

**this**.hw =hw;

}

**public** **void** run() {

hw.method1(5);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

HelloWorld hw;

**public** HelloWorld2(HelloWorld hw) {

**this**.hw =hw;

}

**public** **void** run() {

hw.method1(100);

}

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

HelloWorld hw =**new** HelloWorld();

HelloWorld1 hw2 =**new** HelloWorld1(hw);

HelloWorld2 hw3 =**new** HelloWorld2(hw);

hw2.start();

hw3.start();

}

}

Static synchronization – If you make any static method as synchronized, the lock will be on class not on object

Problem without static synchronization is suppose there are two objects named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1, t2 ,t3 and t4 because t1 and t2 both refers to common object that have a single lock. But there can be interference between t1 and t3 and t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don’t want interference between t1, t2 , t3 and t4 static synchronization solves the problem

**package** com.lokesh;

**public** **class** HelloWorld {

**public** **static** **synchronized** **void** method1(**int** n) {

**for**(**int** i=1;i<=5;i++) {

System.***out***.println(n\*i);

**try** {

Thread.*sleep*(300);

}**catch**(Exception e) {

System.***out***.println(e);

}

}

} //till here

}

**package** com.lokesh;

**public** **class** HelloWorld1 **extends** Thread {

**public** **void** run() {

HelloWorld.method1(5);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

HelloWorld.*method1*(100);

}

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

HelloWorld1 hw =**new** HelloWorld1();

HelloWorld2 hw2 =**new** HelloWorld2();

hw.start();

hw2.start();

}

}

Deadlock – Deadlock can occur in situation when a thread is waiting for an object lock, that is acquired another thread and second thread is waiting for object lock that is required by first thread

Since both threads are waiting for each other to release the lock the condition is called deadlock

Inter thread communication (cooperation) – is all about allowing synchronized threads to communicate with each other

Inter thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

It is implemented using following methods

wait() – wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or notify

It waits until object is notified

notify() – notify() method wakes up a single thread that is waiting on this object’s monitor. If any threads are waiting on this object, one of them is chosen to be awakened.

notifyAll() – notifyAll() method wakes up all the threads that are waiting on object’s monitor

Example:

**package** com.lokesh;

**public** **class** HelloWorld1 **extends** Thread {

**int** amount =1000;

**public** **synchronized** **void** withdraw(**int** amount) {

System.***out***.println("going to withdraw");

**if**(**this**.amount<amount) {

System.***out***.println("less balance, waiting for deposit");

**try** {

wait();

}**catch**(Exception e) {

}

**this**.amount-=amount;

System.***out***.println("withdraw completed");

}

}

**public** **synchronized** **void** deposit(**int** amount) {

System.***out***.println("goint to deposit");

**this**.amount+=amount;

System.***out***.println("deposit completed");

notify();

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

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

HelloWorld1 hw =**new** HelloWorld1();

**new** Thread(){

**public** **void** run() {

hw.withdraw(2000);

}

}.start();

**new** Thread() {

**public** **void** run() {

hw.deposit(1000);

}

}.start();

}

}

Interrupting a thread – if any thread is in sleeping or waiting state, calling interrupt() method on the thread breaks the sleeping or waiting state throwing interrupted exception. If the thread is not sleeping or waiting state calling the interrupt() method performs normal behaviour and doesn’t interrupt the thread but sets the interrupt flag to true

Example: Interrupting a thread that stopped working…..this results in throwing interrupted exception

**package** com.lokesh;

**import** javax.management.RuntimeErrorException;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

**try** {

Thread.*sleep*(5000);

System.***out***.println("lets see program comes here");

}**catch**(InterruptedException e) {

**throw** **new** RuntimeException("interrupted exception "+e);

}

System.***out***.println("thread is running came here");

}

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

HelloWorld2 hw2=**new** HelloWorld2();

hw2.start();

hw2.interrupt();

}

}

Example: Interrupting a thread that does not stop working…..since in below we handled exception so it will break out of sleeping but will not stop working

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

**try** {

Thread.*sleep*(5000);

System.***out***.println("lets see program comes here");

}**catch**(InterruptedException e) {

System.***out***.println("interrupted exception "+e);

}

System.***out***.println("thread is running came here");

}

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

HelloWorld2 hw2=**new** HelloWorld2();

hw2.start();

hw2.interrupt();

}

}

Example: If a thread is not in sleeping or waiting state , calling the interrupt() method sets the interrupt flag to true that can be used to stop the thread by programmer later

**package** com.lokesh;

**import** javax.management.RuntimeErrorException;

**public** **class** HelloWorld2 **extends** Thread {

**public** **void** run() {

**for**(**int** i=1;i<=5;i++) {

System.***out***.println(i);

}

}

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

HelloWorld2 hw2=**new** HelloWorld2();

hw2.start();

hw2.interrupt();

}

}