Inter-thread Communication in Java

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (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 by following methods of **Object class**:

* wait()
* notify()
* notifyAll()

1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final void wait()throws InterruptedException | It waits until object is notified. |
| public final void wait(long timeout)throws InterruptedException | It waits for the specified amount of time. |

### 2) notify() method

The 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. The choice is arbitrary and occurs at the discretion of the implementation.

**Syntax:**

1. **public** **final** **void** notify()

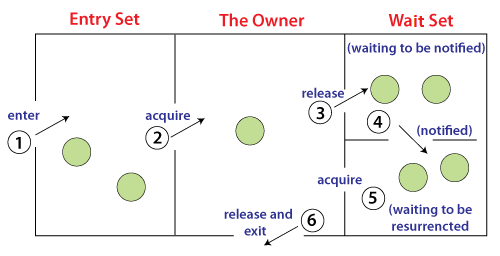
### 3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

**Syntax:**

1. **public** **final** **void** notifyAll()

## Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

### Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

### Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| The wait() method releases the lock. | The sleep() method doesn't release the lock. |
| It is a method of Object class | It is a method of Thread class |
| It is the non-static method | It is the static method |
| It should be notified by notify() or notifyAll() methods | After the specified amount of time, sleep is completed. |

### Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

**Test.java**

**class** Customer{

**int** amount=10000;

**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...");

}

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

System.out.println("going to deposit...");

**this**.amount+=amount;

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

notify();

}

}

**class** Test{

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

**final** Customer c=**new** Customer();

**new** Thread(){

**public** **void** run(){c.withdraw(15000);}

}.start();

**new** Thread(){

**public** **void** run(){c.deposit(10000);}

}.start();

}}

**Output:**

going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw completed

# Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

## Setter & Getter Method of Thread Priority

**public final int getPriority():** The java.lang.Thread.getPriority() method returns the priority of the given thread.

**public final void setPriority(int newPriority):** The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

## 3 constants defined in Thread class:

1. public static int MIN\_PRIORITY
2. public static int NORM\_PRIORITY
3. public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

### Example of priority of a Thread:

**class** ThreadPriorityExample **extends** Thread

{

**public** **void** run()

{

System.out.println("Inside the run() method");

}

**public** **static** **void** main(String argvs[])

{

ThreadPriorityExample th1 = **new** ThreadPriorityExample();

ThreadPriorityExample th2 = **new** ThreadPriorityExample();

ThreadPriorityExample th3 = **new** ThreadPriorityExample();

System.out.println("Priority of the thread th1 is : " + th1.getPriority());

System.out.println("Priority of the thread th2 is : " + th2.getPriority());

System.out.println("Priority of the thread th2 is : " + th2.getPriority());

th1.setPriority(6);

th2.setPriority(3);

th3.setPriority(9);

System.out.println("Priority of the thread th1 is : " + th1.getPriority());

System.out.println("Priority of the thread th2 is : " + th2.getPriority());

System.out.println("Priority of the thread th3 is : " + th3.getPriority());

System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

Thread.currentThread().setPriority(10);

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

}

}

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java thread scheduler. Refer to the following example to comprehend what will happen if two threads have the same priority.

**public** **class** ThreadPriorityExample1 **extends** Thread

{

**public** **void** run()

{

System.out.println("Inside the run() method");

}

**public** **static** **void** main(String argvs[])

{

Thread.currentThread().setPriority(7);

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

ThreadPriorityExample1 th1 = **new** ThreadPriorityExample1();

System.out.println("Priority of the thread th1 is : " + th1.getPriority());  }

}

**Explanation:** If there are two threads that have the same priority, then one can not predict which thread will get the chance to execute first. The execution then is dependent on the thread scheduler's algorithm (First Come First Serve, Round-Robin, etc.)

### Example of IllegalArgumentException

We know that if the value of the parameter newPriority of the method getPriority() goes out of the range (1 to 10), then we get the IllegalArgumentException. Let's observe the same with the help of an example.

**public** **class** IllegalArgumentException **extends** Thread

{

**public** **static** **void** main(String argvs[])

{

Thread.currentThread().setPriority(17);

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

}

}

When we execute the above program, we get the following exception:

Exception in thread "main" java.lang.IllegalArgumentException

at java.base/java.lang.Thread.setPriority(Thread.java:1141)

at IllegalArgumentException.main(IllegalArgumentException.java:12

# Daemon Thread in Java

**Daemon thread in Java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

## Points to remember for Daemon Thread in Java

* It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
* Its life depends on user threads.
* It is a low priority thread.

### Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

### Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public void setDaemon(boolean status) | is used to mark the current thread as daemon thread or user thread. |
| 2) | public boolean isDaemon() | is used to check that current is daemon. |

### Simple example of Daemon thread in java

*File: MyThread.java*

**class** TestDaemonThread1 **extends** Thread{

**public** **void** run(){

**if**(Thread.currentThread().isDaemon()){//checking for daemon thread

   System.out.println("daemon thread work");

  }

**else**{

  System.out.println("user thread work");

 }

 }

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

  TestDaemonThread1 t1=**new** TestDaemonThread1();//creating thread

  TestDaemonThread1 t2=**new** TestDaemonThread1();

  TestDaemonThread1 t3=**new** TestDaemonThread1();

  t1.setDaemon(**true**);//now t1 is daemon thread

  t1.start();//starting threads

  t2.start();

  t3.start();

 }

}

**Output:**

daemon thread work

user thread work

user thread work

#### Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

*File: MyThread.java*

**class** TestDaemonThread2 **extends** Thread{

**public** **void** run(){

  System.out.println("Name: "+Thread.currentThread().getName());

  System.out.println("Daemon: "+Thread.currentThread().isDaemon());

 }

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

  TestDaemonThread2 t1=**new** TestDaemonThread2();

  TestDaemonThread2 t2=**new** TestDaemonThread2();

  t1.start();

  t1.setDaemon(**true**);//will throw exception here

  t2.start();

 }

}

**Output:**

exception in thread main: java.lang.IllegalThreadStateException