**JAVA CONCURRENCY**

Creating Threads In Java:

There are two ways to create threads in java language.

1) By extending ****java.lang.Thread**** class.

2) By implementing ****java.lang.Runnable**** interface.

## 1) By Extending java.lang.Thread Class

**class** MyThread **extends** Thread

{

    @Override

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

    {

        //Keep the task to be performed here

    }

}

MyThread myThread = **new** MyThread();

myThread.start();

## 2) By Implementing java.lang.Runnable Interface.

**class** MyThread **implements** Runnable

{

    @Override

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

    {

        //Keep the task to be performed here

    }

}

MyThread myThread = **new** MyThread();//Creating object of your thread that implements Runnable interface

Thread t = **new** Thread(myThread); //passing your thread object to the constructor of Thread class

t.start();

# [Different Ways Of Defining Threads In Java](https://javaconceptoftheday.com/defining-threads-in-java/)

1. Thread As A Separate Concrete Class
2. Thread As A Nested Class or Static Inner Class
3. Thread As A Member Inner Class or Non-static Inner Class
4. Thread As A Local Inner Class
5. Thread As An Anonymous Inner Class
6. Usage class itself as a thread class.

There are two types of Threads in java.

1) User Thread

2) Daemon Thread

## Some Things-To-Remember about user threads and daemon threads In Java :

* You can convert user thread into daemon thread explicitly by calling setDaemon() method of the thread.
* You can’t set a daemon property after starting the thread. If you try to set the daemon property when the thread is active, It will throw a IllegalThreadStateException at run time.
* You can check whether the thread is user thread or a daemon thread by using isDaemon() method of Thread class. This method returns “true” for a daemon thread and “false” for a user thread.
* Daemon property of a thread is inherited from it’s parent thread. i.e The thread created by user thread will be user thread and the thread created by daemon thread will be a daemon thread.
* The task of daemon thread will not be completed. Program terminates as soon as user thread finishes it’s task. It will not wait for daemon thread to finish it’s task.

# [Naming A Thread In Java](https://javaconceptoftheday.com/naming-thread-java/)

You can give a name to a thread by using ****setName()**** method of Thread class. You can also retrieve the name of a thread using ****getName()**** method of a Thread class. These two methods are public and final. Below is the method signatures of these methods.

****1)  public final void setName(String name)****   —-> It changes the name of the thread to “name”.

****2)  public final String getName()****   —-> Returns the name of the thread.

## Some Things-To-Remember about Naming a thread in java :

* setName() method may throw a SecurityException at run time if the current thread can not modify the name of the specified thread.
* You can change the name of a thread at any state of the thread.
* ****Default Name Of The Thread :**** In Java, All threads have names. If you are not providing the name to a thread, thread will get default name. Default name of the thread will be consist of a word “Thread”, followed by hyphen (-) and followed by an integer number starting with 0.
* ****How to retrieve a name of the primary thread or main thread? :**** First, get the reference of the main thread by using ****currentThread() method**** of Thread class. currentThread() method returns the reference of currently executing thread. After, getting the reference of the main thread, use the getName() method to retrieve the name of the thread.
* ****Can we change the name of the main thread ?**** Yes, we can change the name of the main thread. It can be done by first getting the reference of the main thread by using currentThread() method and then calling setName() method on it

# [How to identify a thread in java?](https://javaconceptoftheday.com/how-to-identify-a-thread-in-java/)

The answer which effortlessly comes to our mind is “through it’s name”. Of course, you can identify a thread by it’s name.  But, more than one threads can have the same name. This makes identifying a thread more difficult. There is a solution for this problem from JDK 1.5 onward. JVM assigns one unique long number for every thread created. This remains unchanged for the whole life term of a thread. This number can be used to identify a thread.

From JDK 1.5 onward, One more method added to java.lang.Thread class. That is ****getID() method****. This method returns the unique long number associated with a thread. That can be used as an identifier of a thread. Below is the method signature of getID() method.

****public long getID()****

## Some Things-To-Remember about Identifying a Thread In Java :

****1)**** Thread ID is a ****unique positive long number****. It remains the same for a thread during its whole life term. Thread ID may be reused when the thread is terminated.

****2) Can we use the thread ID before a thread is started?****

Thread ID is generated as soon as the thread is created. So, you can use the thread ID before starting the thread.

****3) Does thread ID changes when the thread name is changed?.****

Thread ID doesn’t change when the name of a thread is changed. Therefore, if the thread name is changed, still thread can be identified by it’s ID.

# [Priority Of A Thread In Java](https://javaconceptoftheday.com/priority-thread-java/)

 Java application can have more than one threads running simultaneously. When an application has multiple threads they are choosen to execute on priority basis. A thread with highest priority is choosen first for execution than the thread with lowest priority.

There are two methods in java.lang.Thread class related to priority of a thread.

****public final void setPriority(int newPriority)****  —> Changes the priority of a thread to newPriority.

****public final int getPriority()****  —>  Returns the priority of a thread.

## Some Things-To-Remember about priority of a thread in java :

* There are three constant fields in java.lang.Thread class related to priority of a thread. They are,

****MIN\_PRIORITY****   —> It defines the lowest priority that a thread can have and It’s value is 1.

****NORM\_PRIORITY****  —> It defines the normal priority that a thread can have and it’s value is 5.

****MAX\_PRIORITY****  —> It defines the highest priority that a thread can have and it’s value is 10.

* setPriority() method may throw two exceptions. One is ****IllegelArgumentException**** if supplied priority is not in the range of MIN\_PRIORITY and MAX\_PRIORITY and another one is ****SecurityException**** if current thread can not modify the priority of a specified thread.
* ****How to retrieve the priority of a main thread?**** get the reference to a main thread using currentThread() method of Thread class. After getting the reference of main thread, call getPriority() method on it.
* ****Can we change the priority of a main thread?.**** Yes, we can change the priority of a main thread. First, get the reference of main thread using CurrentThread() method. Then call setPriority() method on it.
* The priority of a main thread, if explicitly not set, is always 5 i.e NORM\_PRIORITY.
* The default priority of a thread is same as that of it’s parent.

# [Thread.sleep() Method In Java](https://javaconceptoftheday.com/thread-sleep-method-java/)

Thread.sleep() method makes the currently executing thread to pause it’s execution for a specified period of time. There are two overloaded forms of sleep() method available in java.lang.Thread class. They are,

1) ****public static void sleep(long millis) throws InterruptedException****

—> It causes the currently executing thread to sleep for specified number of milliseconds.

2) ****public static void sleep(long millis, int nanos) throws InterruptedException****

—> It makes the currently executing thread to sleep for specified number of milliseconds plus specified number of nanoseconds.

Thread.sleep() method throws InterruptedException if a thread in sleep is interrupted by other threads. InterruptedException is a checked type of exception. That means, “Thread.sleep()” statement must be enclosed within try-catch blocks or it must be specified with throws clause.

## Some Things-To-Remember About Thread.sleep() Method In Java :

It is always current thread that is going to sleep.

**class** MyThread **extends** Thread

{

**public** MyThread(String name)

    {

**super**(name);

    }

    @Override

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

    {

**for**(**int** i = 0; i <= 1000; i++)

        {

            System.out.println(i);

        }

    }

}

**public** **class** ThreadsInJava

{

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

    {

        MyThread thread = **new** MyThread("My Thread");

        thread.start();

        System.out.println("Before Sleeping");

**try**

        {

            thread.sleep(5000);       //main thread is going for sleep not My Thread

        }

**catch** (InterruptedException e)

        {

            // TODO Auto-generated catch block

            e.printStackTrace();

        }

        System.out.println("After Sleeping");

    }

}

* It is a bad practice to call sleep() method with an instance of Thread class as in the above example. If you want a particular thread to sleep for a while, then call sleep() method inside the run() method of that thread.

**class** MyThread **extends** Thread

{

    @Override

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

    {

**for**(**int** i = 0; i <= 10; i++)

        {

            System.out.println(i);

**try**

            {

                sleep(1000);            //this thread sleeps for 1 second

            }

**catch** (InterruptedException e)

            {

                e.printStackTrace();

            }

        }

    }

}

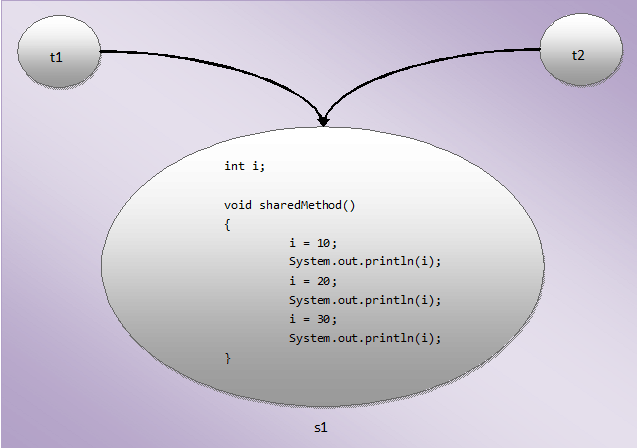
* Thread.sleep() method may also throws IllegalArgumentException if miilis value is negative or nanos value is not in the range 0 – 999999.
* When the thread is going for sleep, it does not release the synchronized locks it holds.

# [Joining The Threads In Java](https://javaconceptoftheday.com/joining-threads-java/)

Like sleep() method, join() method also throws InterruptedException. Therefore, you have to keep calling statement to join() method in try-catch blocks or else propagate the exception with throws clause.

# [Thread Interference In Java](https://javaconceptoftheday.com/thread-interference-in-java/)

****Thread interference in java**** is a condition which occurs when more than one threads, executing simultaneously, access same piece of data. When more than one threads have access to same data, it is possible that data may get corrupted or one may not get the desired output. Thread interference occurs when code written is not ****thread safe****.



## How To Avoid Thread Interference or How To Acheive Thread Safeness?

Following are some methods which are used to avoid thread interference in java.(These methods will be discussed in detail in subsequent articles).

* By declaring the method as synchronized.
* By declaring the variables as final.
* By declaring the variable as volatile.
* By creating the immutable objects.
* By using Atomic operations.
* By restricting the access to same object by multiple threads.

# [Synchronization In Java](https://javaconceptoftheday.com/synchronization-in-java/)

****Synchronization in java**** is a strategy or a method to avoid [thread interference](https://javaconceptoftheday.com/thread-interference-in-java/" \t "https://javaconceptoftheday.com/synchronization-in-java/_blank) and hence protecting the data from inconsistency. synchronization is also one of the way to make code thread safe. Through synchronization, we can make the threads to execute particular method or block in sync not simultaneously.

Synchronization in java is implemented using ****synchronized**** keyword. synchronized keyword can be used with methods or blocks but not with the variables.

When a method or block is declared as synchronized, only one thread can enter into that method or block. When one thread is executing synchronized method or block, the other threads which wants to execute that method or block wait or suspend their execution until first thread is done with that method or block. Thus avoiding the thread interference and achieving thread safeness.

## The Logic Behind The Synchronization In Java :

The synchronization in java is built around an entity called ****object lock****or****monitor****. Here is the brief description about lock or monitor.

* Whenever an object is created to any class, an object lock is created and is stored inside the object.
* One object will have only one object lock associated with it.
* Any thread wants to enter into synchronized methods or blocks of any object, they must acquire object lock associated with that object and release the lock after they are done with the execution.
* The other threads which wants to enter into synchronized methods of that object have to wait until the currently executing thread releases the object lock.
* To enter into static synchronized methods or blocks, threads have to acquire class lock associated with that class as static members are stored inside the class memory.

## Synchronized Blocks :

Some times, you need only some part of the method to be synchronized not the whole method. This can be achieved with synchronized blocks. Synchronized blocks must be defined inside a definition blocks like methods, constructors, static initializer or instance initializer.

synchronized block takes one argument and it is called ****mutex****.

if synchronized block is defined inside non-static definition blocks like non-static methods, instance initializer or constructors, then this mutex must be an instance of that class.

If synchronized block is defined inside static definition blocks like static methods or static initializer, then this mutex must be like ClassName.class.

**class** Shared

{

**static** **void** staticMethod()

    {

**synchronized** (Shared.**class**)

        {

            //static synchronized block

        }

    }

**void** NonStaticMethod()

    {

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

        {

            //Non-static synchronized block

        }

    }

**void** anotherNonStaticMethod()

    {

**synchronized** (**new** Shared())

        {

            //Non-static synchronized block

        }

    }

}

## 10 Points-To-Remember About Synchronization In Java :

1. You can use ****synchronized**** keyword only with methods but not with variables, constructors, static initializer and instance initializers.
2. Constructors, Static initializer and instance initializer can’t be declared with synchronized keyword, but they can contain synchronized blocks.
3. Both static and non-static methods can use synchronized keyword. For static methods, thread need class level lock and for non-static methods, thread need object level lock.
4. It is possible that both static synchronized and non-static synchronized methods can run simultaneously. Because, static methods need class level lock and non-static methods need object level lock.
5. A method can contain any number of synchronized blocks. This is like synchronizing multiple parts of a method.
6. Synchronization blocks can be nested.
7. Lock acquired by the thread before executing a synchronized method or block must be released after the completion of execution, no matter whether execution is completed normally or abnormally (due to exceptions).
8. Synchronization in java is ****Re-entrant in nature****. A thread can not acquire a lock that is owned by another thread. But, a thread can acquire a lock that it already owns. That means if a synchronized method gives a call to another synchronized method which needs same lock, then currently executing thread can directly enter into that method or block without acquiring the lock.
9. synchronized method or block is very slow. They decrease the performance of an application. So, special care need to be taken while using synchronization. Use synchronization only when you needed it the most.
10. Use synchronized blocks instead of synchronized methods. Because, synchronizing some part of a method improves the performance than synchronizing the whole method.

## What Is Deadlock In Java?

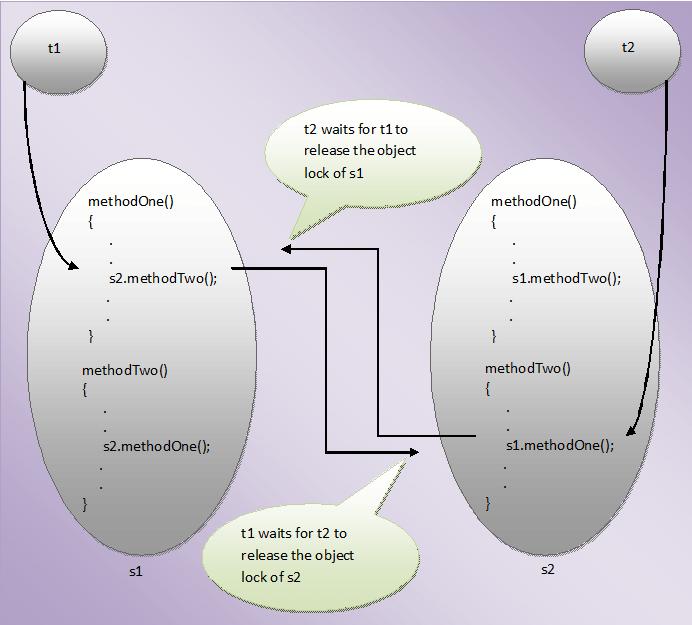
Deadlock in java is a condition which occurs when two or more threads get blocked waiting for each other for an infinite period of time to release the resources(Locks) they hold. Deadlock is the common problem in multi threaded programming which can completely stops the execution of an application. So, extra care need to be taken while writing the multi threaded programs so that deadlock never occurs.

thread ****t1**** and ****t2**** are concurrent threads i.e they are executing their task simultaneously. There are two Shared class objects, ****s1**** and ****s2****, which are shared by both the threads. Shared class has two synchronized methods, ****methodOne()**** and ****methodTwo()****. That means, only one thread can execute these methods at a given time.

First, thread ****t1**** enters the ****methodOne()**** of ****s1**** object by acquiring the object lock of ****s1****. At the same time, thread ****t2**** also enters the ****methodTwo()**** of ****s2**** object by acquiring the object lock of ****s2****. ****methodOne()**** of ****s1**** object, currently executing by thread ****t1****, calls ****methodTwo()**** of ****s2**** object from it’s body. So, thead ****t1**** tries to acquire the object lock of ****s2**** object. But object lock of ****s2**** object is already acquired by thread ****t2****. So, thread ****t1**** waits for thread ****t2**** to release the object lock of ****s2**** object.

At the same time, thread ****t2**** is also executing ****methodTwo()**** of ****s2**** object. ****methodTwo()**** of ****s2**** object also makes a call to ****methodOne()**** of ****s1**** object. So, thread ****t2**** tries to acquire the object lock of ****s1**** object. But, it is already acquired by thread ****t1****. So, thread ****t2**** also waits for thread ****t1**** to release the object lock of ****s1**** object.

Thus, both the threads wait for each other to release the object locks they own. They wait for infinite period of time to get the object locks owned by opposite threads. This condition of threads waiting forever is called Deadlock.



# [How To Detect The Deadlocked Threads In Java](https://javaconceptoftheday.com/detect-deadlocked-threads-using-threadmxbean-class-java/)

Programmatically, You can detect the threads which have entered into deadlock condition and also you can retrieve the details about them.

This can be done using ****ThreadMXBean**** interface of ****java.lang.Management**** package.

ThreadMXBean bean = ManagementFactory.getThreadMXBean();

After getting an instance of ThreadMXBean, call ****findMonitorDeadlockedThreads()**** method on it. It returns an array of type long containing ids of all currently deadlocked threads.

**long** ids[] = bean.findMonitorDeadlockedThreads();

After getting the ids of deadlocked threads, pass these ids to ****getThreadInfo()**** method of ThreadMXBean. It will return an array of ThreadInfo objects, where one ThreadInfo object contains the details of one deadlocked thread.

ThreadInfo threadInfo[] = bean.getThreadInfo(ids);

Iterate the ThreadInfo array to get the details of individual deadlocked thread.

**for** (ThreadInfo threadInfo1 : threadInfo)

{

    System.out.println(threadInfo1.getThreadName());    //Prints the name of deadlocked thread

}

Here are the some methods of ThreadInfo class which are useful to retrieve the details of deadlocked threads.

****getThreadId()****               —>    Returns the ID of a deadlocked thread.

****getThreadName()****         —>     Returns the name of a deadlocked thread.

****getBlockedTime()****          —>    Returns the elapsed time in milli seconds that a thread is in deadlock condition.

****getLockName()****             —>    Returns string representation of an object for which thread has been waiting.

****getLockOwnerId()****         —>    Returns ID of a thread that currently owns the object lock.

****getLockOwnerName()****    —>    Returns the name of a thread that currently owns the object lock.

# [How To Avoid The Deadlock In Java](https://javaconceptoftheday.com/avoid-the-deadlock-java/)

Deadlock is a dangerous condition, if it happens , it will bring the whole application to complete halt. So, extra care need to be taken to avoid the deadlock. Followings are some tips that can be used to avoid the deadlock in java.

* Try to avoid nested synchronized blocks. Nested synchronized blocks makes a thread to acquire another lock while it is already holding one lock. This may create the deadlock if another thread wants the same lock which is currently held by this thread.
* ****Lock Ordering :****

If you needed nested synchronized blocks at any cost, then make sure that threads acquire the needed locks in some predefined order. For example, If there are three threads t1, t2 and t3 running concurrently and they needed locks A, B and C in the following manner,

Thread t1 :

        Lock A

        Lock B

Thread t2 :

        Lock A

        Lock C

Thread t3 :

        Lock A

        Lock B

        Lock C

In the above scenario, t1 needs A and B locks, t2 needs A and C locks and t3 needs A, B and C locks. If you define an order to acquire the locks like, Lock A must be acquired before Lock B and Lock B must be acquired before Lock c, then deadlock never occurs in the above case.

If you define such lock ordering, then thread t2 never acquire lock C and t3 never acquire lock B and lock C until they got lock A. They will wait for lock A until it is released by t1. After lock A is released by t1, any one of these threads will acquire lock A on the priority basis and finishes their task. Other thread which is waiting for lock A, will never try to acquire remaining locks.

By defining such lock ordering, you can avoid the deadlock.

* ****Lock Timeout :****

Another deadlock preventive tip is to specify the time for a thread to acquire the lock. If it fails to acquire the specified lock in the given time, then it should give up trying for a lock and retry after some time. Such method of specifying time to acquire the lock is called lock timeout.

* Lock the code where it is actually needed. For example,If you want only some part of the method to be thread safety, then lock only that part not the whole method.

**void** method()

{

    //Some statements

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

    {

        //Locking only some part of the method

    }

    //Some statements

}

## Some Things-To-Remember About wait(), notify() and notifyAll() :

* If a thread calls notify() method and more than one threads are waiting for the object lock, then only one thread will be notified randomly.
* When a thread calls notifyAll() method on an object, it notifies all the threads which are waiting for this object lock. But, only one thread will acquire this object lock depending upon priority.
* When you call sleep() method on a thread, thread goes to sleep with holding the object lock with it. But, if you call wait() method, thread releases the object lock and goes for sleep. This is the main difference between wait() and sleep() methods.
* wait(), notify() and notifyAll() are final methods of ****java.lang.Object**** class not java.lang.Thread class.
* wait(), notify() and notifyAll() – all these three methods throw ****IllegalMonitorStateException**** if the calling thread does not owns the object lock.

wait() method is overloaded in Object class. There are two more wait() methods available in Object class.

# [Thread Interruption In Java](https://javaconceptoftheday.com/thread-interruption-java/)

****Thread interruption in java**** is a mechanism in which a thread which is either sleeping or waiting can be made to stop sleeping or waiting. Thread interruption is like telling the thread that it should stop waiting or sleeping and return to running status. Thread interruption is programmatically implemented using ****interrupt()**** method of java.lang.Thread class. interrupt() method is a non-static public method of Thread class. Here is the method signature of interrupt() method.

****public void interrupt()****

The whole thread interruption mechanism depends on an internal flag called ****interrupt status****. The initial value of this flag for any thread is false. When you call interrupt() method on a thread, interrupt status of that thread will be set to true. When a thread throws InterruptedException, this status will be set to false again. Remember, InterruptedException is thrown when a thread is interrupted while it is sleeping or waiting. Many methods of Thread class like sleep(), wait(), join() throw InterruptedException.

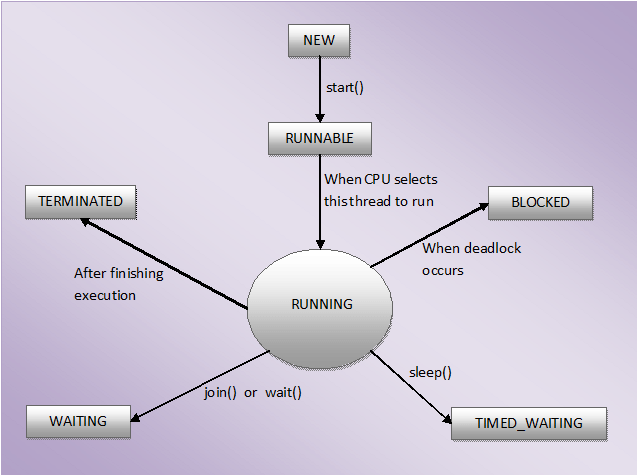
## Some Things-To-Remember About Thread Interruption In Java :

* You can check whether a particular thread is interrupted or not using ****isInterrupted()**** method of Thread class. This method returns current interrupt status of a thread.
* A thread can interrupt itself. i.e a thread can call interrupt() method on it’s own.
* There is one more method to check interrupt status of a thread, called ****interrupted()**** method. It is a static method of Thread class. It also returns the current interrupt status of a thread like isInterrupted() method. But, it clears interrupt status of a thread. i.e if interrupt status of a thread is true, then it will set the status to false.
* interrupt() method will throw ****SecurityException**** if current thread can not interrupt a calling thread.

# [Thread Life Cycle OR Thread States In Java](https://javaconceptoftheday.com/thread-life-cycle-thread-states-java/)

There are six thread states. They are NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING and TERMINATED. At any point of time, thread will be in any one of these states.

java.lang.Thread class has one member of enum type called ****State****.



# [Thread Group In Java](https://javaconceptoftheday.com/thread-group-java/)

Thread group in java is used to group similar threads into one unit. A thread group can also contain other thread groups. Thread groups are constructed using ****java.lang.ThreadGroup**** class. The main use of thread groups is that you can handle multiple threads simultaneously.

While creating the threads itself, you can specify it’s group using constructor which takes ThreadGroup and name of a thread as arguments.

# [7 Things Every Java Programmer Should Know About Threads In Java](https://javaconceptoftheday.com/7-things-every-java-programmer-should-know-about-threads-in-java/)

1. If you start a thread that is already started, you will get java.lang.IllegalThreadStateException at run time. There will be no compilation errors.
2. Exception is thread wise not execution wise. i.e exception effects the thread in which it occurs. Other threads will execute normally.
3. As we all know that start() method internally calls run() method. ****What happens when you call run() method directly?****. When you call run() method of a thread directly, calling thread will execute the task defined in the run() method. If in the program main thread is calling run() method of thread t. In this case, main thread will execute run() method not thread t.

**public** **class** ThreadsInJava

{

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

    {

        Thread t = **new** Thread()

        {

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

            {

                System.out.println(Thread.currentThread().getName());    //Output : main

            }

        };

        t.run();

    }

}

1. Which one is better way to implement threads in java. Is it using Thread class or using Runnable interface?. It is the most confusing question for a java developer. I am of opinion that when multiple threads need to execute same task, then use Runnable interface. If multiple threads need to execute different tasks, then go for Thread class.
2. Setting the priority to a thread is not effective as we thought. Setting Priority of a thread is just an advice to OS not an instruction. It is up to OS to consider this advice.
3. Every thread in java is a member of a thread group. When a java application first starts up, Java runtime system creates a thread group called ****main****. main thread is also member of this group.

**public** **class** ThreadsInJava

{

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

    {

        Thread t = Thread.currentThread();

        System.out.println(t.getThreadGroup());

        //Output : java.lang.ThreadGroup[name=main,maxpri=10]

    }

}

1. A thread is a permanent member of a thread group to which it joins during creation. You can’t move a thread to a new group after creating it.

|  |  |
| --- | --- |
| wait() | sleep() |
| The thread which calls wait() method releases the lock it holds. | The thread which calls sleep() method doesn’t release the lock it holds. |
| The thread regains the lock after other threads call either notify() or notifyAll() methods on the same lock. | No question of regaining the lock as thread doesn’t release the lock. |
| wait() method must be called within the synchronized block. | sleep() method can be called within or outside the synchronized block. |
| wait() method is a member of java.lang.Object class. | sleep() method is a member of java.lang.Thread class. |
| wait() method is always called on objects. | sleep() method is always called on threads. |
| wait() is a non-static method of Object class. | sleep() is a static method of Thread class. |
| Waiting threads can be woken up by other threads by calling notify() or notifyAll() methods. | Sleeping threads can not be woken up by other threads. If done so, thread will throw InterruptedException. |
| To call wait() method, thread must have object lock. | To call sleep() method, thread need not to have object lock. |

|  |  |
| --- | --- |
| Process | Thread |
| Processes are heavy weight operations. | Threads are light weight operations. |
| Every process has its own memory space. | Threads use the memory of the process they belong to. |
| Inter process communication is slow as processes have different memory address. | Inter thread communication is fast as threads of the same process share the same memory address of the process they belong to. |
| Context switching between the process is more expensive. | Context switching between threads of the same process is less expensive. |
| Processes don’t share the memory with other processes. | Threads share the memory with other threads of the same process. |

|  |  |
| --- | --- |
| ****User Threads**** | ****Daemon Threads**** |
| JVM waits for user threads to finish their work. It will not exit until all user threads finish their work. | JVM will not wait for daemon threads to finish their work. It will exit as soon as all user threads finish their work. |
| User threads are foreground threads. | Daemon threads are background threads. |
| User threads are high priority threads. | Daemon threads are low priority threads. |
| User threads are created by the application. | Daemon threads, in most of time, are created by the JVM. |
| User threads are mainly designed to do some specific task. | Daemon threads are designed to support the user threads. |
| JVM will not force the user threads to terminate. It will wait for user threads to terminate themselves. | JVM will force the daemon threads to terminate if all user threads have finished their work. |

|  |  |
| --- | --- |
| ****Implements Runnable**** | ****Extends Thread**** |
| You can extend any other class. | You can’t extend any other class. |
| No overhead of additional methods . | Overhead of additional methods from Thread class. |
| Separates the task from the runner. | Doesn’t separate the task from the runner. |
| Best object oriented programming practice. | Not a good object oriented programming practice. |
| Loosely coupled. | Tightly coupled. |
| Improves the reusability of the code. | Doesn’t improve the reusability of the code. |
| More generalized task. | Thread specific task. |
| Maintenance  of the code will be easy. | Maintenance of the code will be time consuming. |

# [How To Stop A Thread In Java?](https://javaconceptoftheday.com/how-to-stop-a-thread-in-java/)

One is using *boolean variable* and second one is using *interrupt()* method.

# [Difference Between notify And notifyAll In Java](https://javaconceptoftheday.com/difference-between-notify-and-notifyall-in-java/)

*notify()* and *notifyAll()* methods along with *wait()* method are used to establish a communication between the threads. A thread goes into WAITING mode by calling *wait()* method. This thread will be in WAITING state until any other thread calls either *notify()* or *notifyAll()* method on the same object.Any thread calling *wait()*, *notify()* and *notifyAll()* must have lock of that object. In the other words, these methods must be called within synchronized method or synchronized block.

### notify() In Java :

When a thread calls *notify()* method on a particular object, only one thread will be notified which is waiting for the lock or monitor of that object. The thread chosen to notify is random i.e randomly one thread will be selected for notification. Notified thread doesn’t get the lock of the object immediately. It gets once the calling thread releases the lock of that object. Until that it will be in BLOCKED state. It will move from BLOCKED state to RUNNING state once it gets the lock.

****Note****: Before notification, the thread will be in WAITING state. Once it is notified, it will move to BLOCKED state. It remains in BLOCKED state until it gets the lock. Once it gets the lock, it moves from BLOCKED state to RUNNING state.

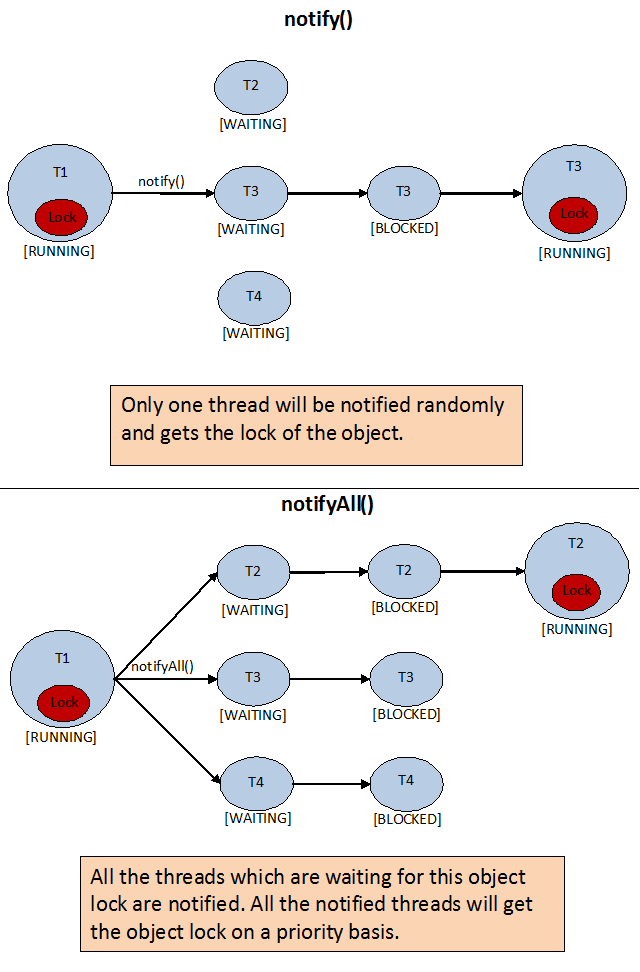
### notifyAll() In Java :

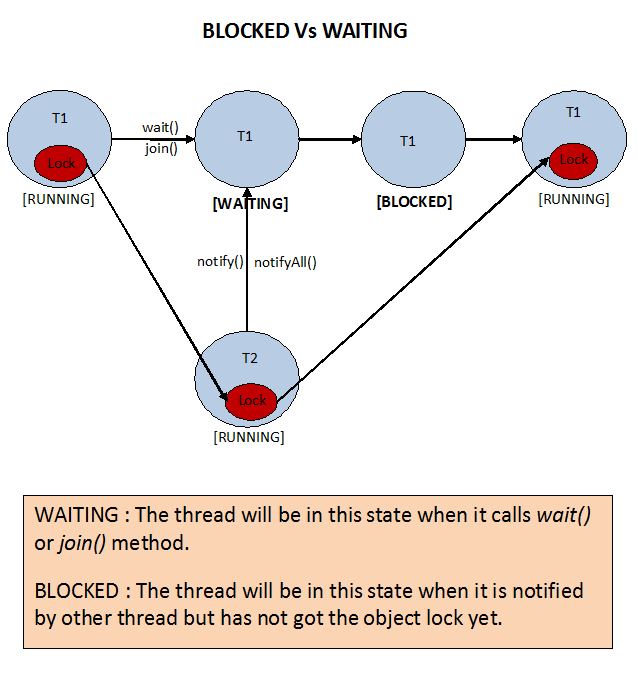
When a thread calls *notifyAll()* method on a particular object, all threads which are waiting for the lock of that object are notified. All notified threads will move from WAITING state to BLOCKED state. All these threads will get the lock of the object on a priority basis. The thread which gets the lock of the object moves to RUNNING state. The remaining threads will remain in BLOCKED state until they get the object lock.

### BLOCKED Vs WAITING States In Java :

|  |  |
| --- | --- |
| ****WAITING**** | ****BLOCKED**** |
| The thread will be in this state when it calls *wait()* or *join()* method. The thread will remain in WAITING state until any other thread calls *notify()* or *notifyAll()*. | The thread will be in this state when it is notified by other thread but has not got the object lock yet. |
| The WAITING thread is waiting for notification from other threads. | The BLOCKED thread is waiting for other thread to release the lock. |
| The WAITING thread can be interrupted. | The BLOCKED thread can’t be interrupted. |

### Difference Between notify And notifyAll In Java :





#### start() Vs run() Methods In Java Threads :

|  |  |
| --- | --- |
| ****start()**** | ****run()**** |
| New thread is created. | No new thread is created. |
| Newly created thread executes task kept in run() method. | Calling thread itself executes task kept in run() method. |
| It is a member of *java.lang.Thread* class. | It is a member of *java.lang.Runnable* interface. |
| You can’t call start() method more than once. | You can call run() method multiple times. |
| Use of multi-threaded programming concept. | No use of multi-threaded programming concept. |

# [7 Things Every Java Programmer Should Know About Threads In Java](https://javaconceptoftheday.com/7-things-every-java-programmer-should-know-about-threads-in-java/)

1. If you start a thread that is already started, you will get java.lang.IllegalThreadStateException at run time. There will be no compilation errors.
2. Exception is thread wise not execution wise. i.e exception effects the thread in which it occurs. Other threads will execute normally. In the below example, exception occurs in thread t1. only this thread will be terminated abruptly. Thread t2 will continue to execute it’s task.
3. As we all know that start() method internally calls run() method. ****What happens when you call run() method directly?****. When you call run() method of a thread directly, calling thread will execute the task defined in the run() method.
4. Which one is better way to implement threads in java. Is it using Thread class or using Runnable interface?. It is the most confusing question for a java developer. I am of opinion that when multiple threads need to execute same task, then use Runnable interface. If multiple threads need to execute different tasks, then go for Thread class.
5. Setting the priority to a thread is not effective as we thought. Setting Priority of a thread is just an advice to OS not an instruction. It is up to OS to consider this advice.
6. Every thread in java is a member of a thread group. When a java application first starts up, Java runtime system creates a thread group called ****main****. main thread is also member of this group.
7. A thread is a permanent member of a thread group to which it joins during creation. You can’t move a thread to a new group after creating it.

### [Executor and ExecutorService framework in java:](https://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html)

Executor and ExecutorService are used for  following purposes in java >

* creating thread,
* starting threads,
* managing whole [life cycle of Threads](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

Executor creates [pool of threads](http://www.javamadesoeasy.com/2015/03/implement-thread-pool-in-java.html) and manages life cycle of all threads in it in java.

java.util.concurrent.Executor interface defines very important execute() method which executes command in java.

**void execute(Runnable command)**

Executes the given command. Executor may execute command in a

* new thread, or
* in a pooled thread, or
* in the calling thread

at the discretion of the Executor implementation in java.

java.util.concurrent.ExecutorService interface extends Executor interface in java.

An Executor interface provides following type of methods in java >

1. methods for managing termination and
2. methods that can produce a Future for tracking progress of tasks.

An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks in java.

**ExecutorService methods in java:**

**boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException:**

Blocks until one of the following things happen >

* all tasks have completed execution after a shutdown request, or
* specified timeout elapses, or
* current thread is interrupted

**<T> Future<T> submit(Callable<T> task) :**

Submits a ***task*** for execution.

Method **returns** a Future which represents pending results of the task.

Once task is completed Future's get method will return the task's result.

**<T> Future<T> submit(Runnable task, T result)**

Submits a Runnable task for execution.

Method returns a Future which represents that task. Once task is completed Future's get method will return result.

**Future<?> submit(Runnable task)**

Submits a Runnable ***task*** for execution.

Method **returns** a Future which represents that task. Once task is completed Future's get method will return null in java.

*void* ***shutdown****()*

Initiates shutdown of executor, previously submitted tasks are executed, but no new tasks will be accepted in java.

**List<Runnable> shutdownNow()**

* executor shutDowns immediately,
* all actively executing tasks are stopped,
* awaiting tasks will never execute, and
* method returns list all tasks that were awaiting execution in java.

*boolean* ***isTerminated****()*

Method returns true if all tasks have completed following shut down in java.

*<T> List<Future<T>>* ***invokeAll****(Collection<? extends Callable<T>>* ***tasks****) throws InterruptedException*

Executes the given tasks and returns a list of Futures holding their status in java.