**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();

            }

        }

    }

}

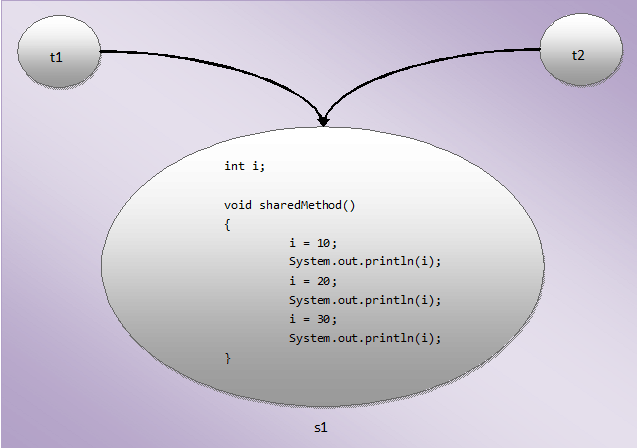
* Thread.sleep() method may also throws IllegalArgumentException if miilis value is negative or nanos value is not in the range 0 – 999999.
* When the thread is going for sleep, it does not release the synchronized locks it holds.

# [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.