How many ways thread can be created?

1) By extending Thread class

2) Implementing Runnable interface

Getter setter for Thread name

Getter and setter for thread priority

Methods prevent thread execution:

1. Join()
2. Yield()
3. Sleep()

Synchronisation

Inter thread communication

1. wait()
2. notify()
3. notifyAll()

Deadlock

Daemon Threads

Multitasking: executing multiple task simultaneously is called multitasking

**Process based multitasking**: executing multiple task simultaneously where each task is separate independent process.

This type of multitasking is applicable in OS level.

Ex: multiple programs running on the same windows system

**Thread based multitasking:** executing multiple task simultaneously where each task is separate independent part of the same program is called process based multitasking.

Each independent part is called thread.

This type of multitasking is applicable in program level.

The main objective of multitasking is to reduce response time of the system and improve performance.

Ex: webserver or application server web container to handle multiple request internally multithreading concept is used.

Thread scheduler is part of jvm .It is responsible to schedule thread. If multiple threads are waiting to get chance of execution then in which threads will be executed is decided by **thread scheduler**.

Thread scheduler very from jvm to jvm hence we cannot expect threads execution order and exact output. Hence whenever situation comes to multi-threading the is no guarantee for exact output .we can provide possible outputs.

t.start() a new thread will be created.

Register the thread with the thread scheduler.

t.run() the run method will be executed in main thread instead of separated thread.

After starting a thread if we call the start() method then it will throw (runtime exception) IllegalThreadStateException

|  |
| --- |
| [**Thread**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread())()  Allocates a new Thread object. |
| [**Thread**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.Runnable))([**Runnable**](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target)  Allocates a new Thread object. |
| [**Thread**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.Runnable,%20java.lang.String))([**Runnable**](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target, [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name)  Allocates a new Thread object. |
| [**Thread**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.String))([**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name)  Allocates a new Thread object. |
| [**Thread**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.ThreadGroup,%20java.lang.Runnable))([**ThreadGroup**](https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadGroup.html) group, [**Runnable**](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target)  Allocates a new Thread object. |
| [**Thread**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.ThreadGroup,%20java.lang.Runnable,%20java.lang.String))([**ThreadGroup**](https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadGroup.html) group, [**Runnable**](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target, [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name)  Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group. |
| [**Thread**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.ThreadGroup,%20java.lang.Runnable,%20java.lang.String,%20long))([**ThreadGroup**](https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadGroup.html) group, [**Runnable**](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) target, [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name, long stackSize)  Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified *stack size*. |
| [**Thread**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#Thread(java.lang.ThreadGroup,%20java.lang.String))([**ThreadGroup**](https://docs.oracle.com/javase/7/docs/api/java/lang/ThreadGroup.html) group, [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) name)  Allocates a new Thread object. |

Every thread by default has a name and we can change to programmatically.

**Thread priorities:**

Thread.MIN\_PRIORITY=1

Thread.NORMAL\_PRIORITY=5

Thread.MAX\_PRIORITY=10

Every thread has some priority it can be either default priority (which is 5) or user defined priorities.

Thread scheduler will use the priority while allocation processer. The thread which is having highest priority will get the chance first.

If two threads having same priority then we cannot expect exact execution order.it depends on thread scheduler.

Any value out of range 0-10 we will get IllegalArgumentException

Some platforms do not provide support for thread priority.

How many ways we can prevent Thread execution?

Yield(),join(),sleep().

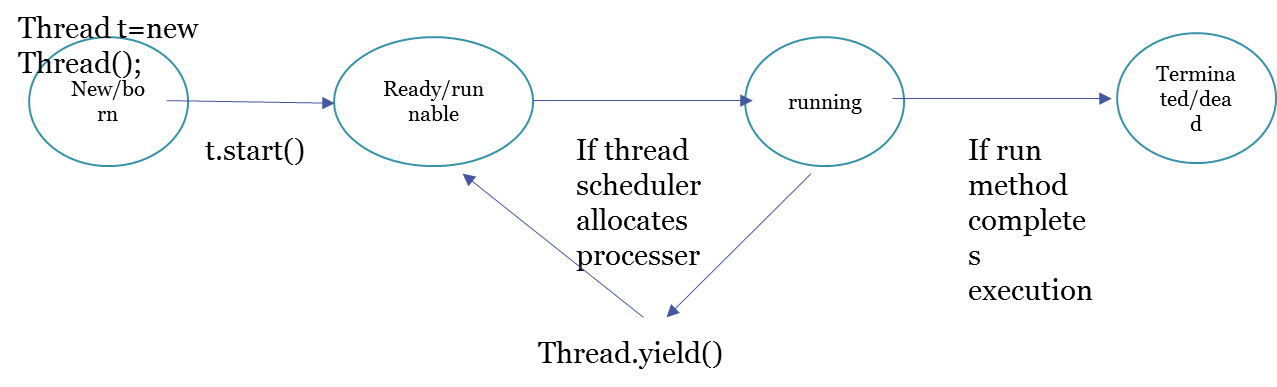
Yield() method causes to pass current executing thread to give the chance for waiting threads of same priority. If there is no waiting thread or waiting threads are of low priority then same thread can continue its execution.

If multiple treads are waiting with same priority then which waiting thread will get the chance we cannot expect it depends on thread scheduler.

Thread.yield()

The thread which is yielded when it will get the chance once again it depends of thread scheduler, we cannot expect exactly.

Public static native void yield()



Some platform will not provide support for yield() method

If a thread wants to wait until completing another thread then we should go for join() method.

If a thread t1 wants to wait until t2 is finished the t1 has to call t2.join ().

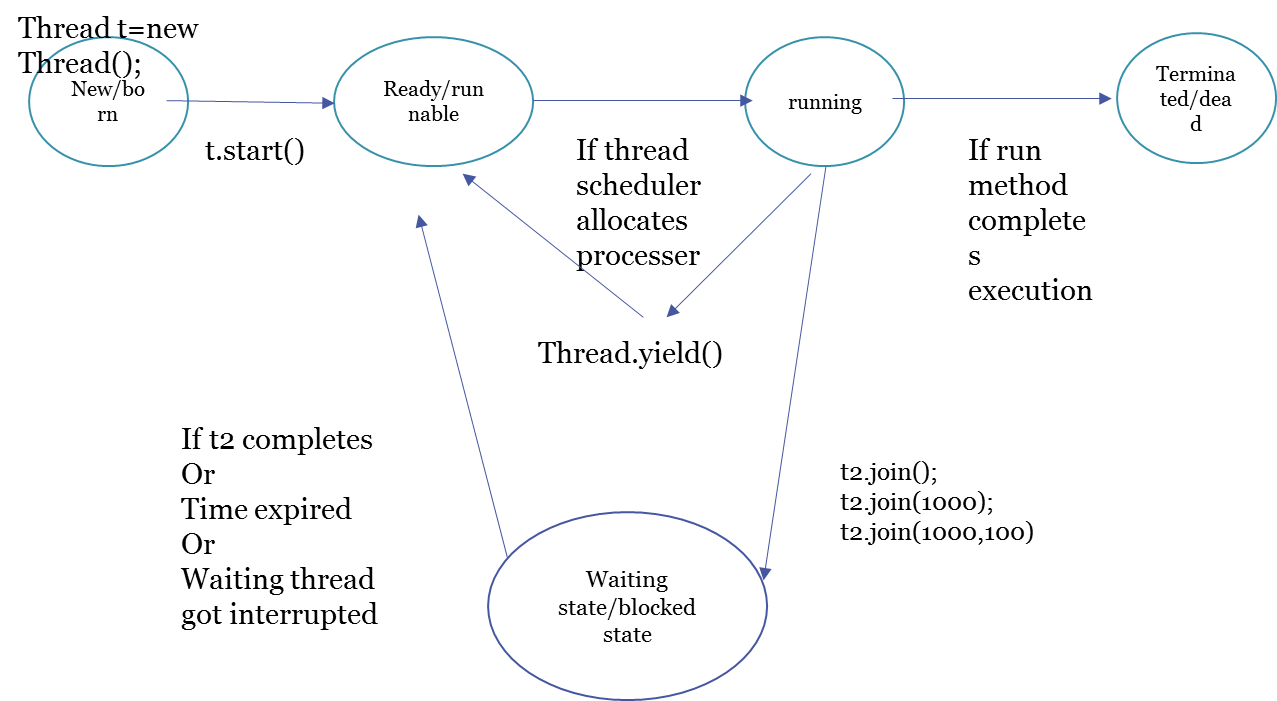
If t1 executes t2.join () then immediately t1 will entered into waiting state until t2 completes.

Once t2 completes t1 can continue its execution.

|  |  |
| --- | --- |
| void | [**join**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#join())()  Waits for this thread to die. |
| void | [**join**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#join(long))(long millis)  Waits at most millis milliseconds for this thread to die. |
| void | [**join**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#join(long,%20int))(long millis, int nanos)  Waits at most millis milliseconds plus nanos nanoseconds for this thread to die. |

All these methods throws InterruptedException.

Every join method throws InterruptedException which is checked exception compulsory we should handle this exception either by using try catch block or throws key word otherwise we will get compile time exception.



public class ThreadJoinExample {

public static void main(String[] args) {

Thread t1 = new Thread(new MyRunnable(), "t1");

Thread t2 = new Thread(new MyRunnable(), "t2");

Thread t3 = new Thread(new MyRunnable(), "t3");

t1.start();

//start second thread after waiting for 2 seconds or if it's dead

try {

t1.join(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

t2.start();

//start third thread only when first thread is dead

try {

t1.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

t3.start();

//let all threads finish execution before finishing main thread

try {

t1.join();

t2.join();

t3.join();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("All threads are dead, exiting main thread");

}

}

class MyRunnable implements Runnable{

@Override

public void run() {

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

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

e.printStackTrace();

}

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

}

}

If my child thread has to wait until main thread is completed, then my child thread has to call join () method on main thread object.

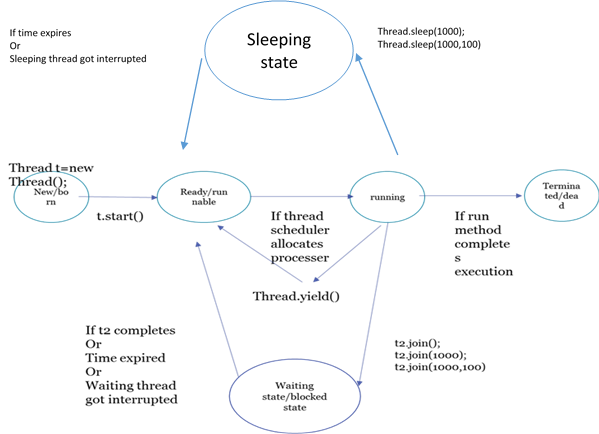
If a thread calls join () on the same thread itself it will stocked it’s kind of dead lock .thread will wait for infinite amount of time.

**Sleep()**

If a thread do not want to perform any operation for a particular amount of time then we should go for sleep method.

|  |  |
| --- | --- |
| static void | [**sleep**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#sleep(long))(long millis)  Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers. |
| static void | [**sleep**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#sleep(long,%20int))(long millis, int nanos)  Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers. |

Sleep () method throws InterruptedException which is checked exception.



If thread is in sleep state it will come to ready or runnable state if

Time expires or the thread got interrupted

How a thread can interrupt another thread

|  |  |
| --- | --- |
| void | [**interrupt**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt())()  Interrupts this thread. |
| static boolean | [**interrupted**](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupted())()  Tests whether the current thread has been interrupted. |

A sleeping/waiting threading can be interrupted using interrupt method.

In interrupt method called on a thread which is not in waiting or sleeping state interrupt call will wait until the thread again goes to sleep/wait state. If the target thread never entered into sleeping or waiting state in its life time then there is no impact of interrupt call. This is the only case where interrupt call will be wasted.

public static native Thread currentThread();

public static native void yield();

public static native void sleep(long millis) throws InterruptedException;

|  |  |  |  |
| --- | --- | --- | --- |
| Yield() | Join() | Sleep() | properties |
| If a Thread wants to pass its execution to give the chance to remaining threads of same priority then we should go for yield() method. | If a thread want to wait until completion of another thread then we should go for join() method. | If a thread do not want to do any operation for a particular amount of time we should go for sleep() method. | purpose |
| no | yes | yes | Is overloaded? |
| no | yes | no | final |
| no | yes | yes | InterruptedException |
| yes | no | Sleep(millis)-> native  Sleep(millies,nanos)-> nonnative | native |
| yes | no | yes | Is it static? |

**Synchronization**

Synchronized is a modifier applicable only for methods and blocks but not for classes and variables.

If multiple Threads are trying to operate simultaneously on the same java object then there may be a chance of data inconsistency problem.

To overcome this problem we should go for synchronized keyword.

If a method or block is declared as synchronized then at a time only one thread is allowed to execute that method or block on the given object so that data inconsistency problem will be resolved.

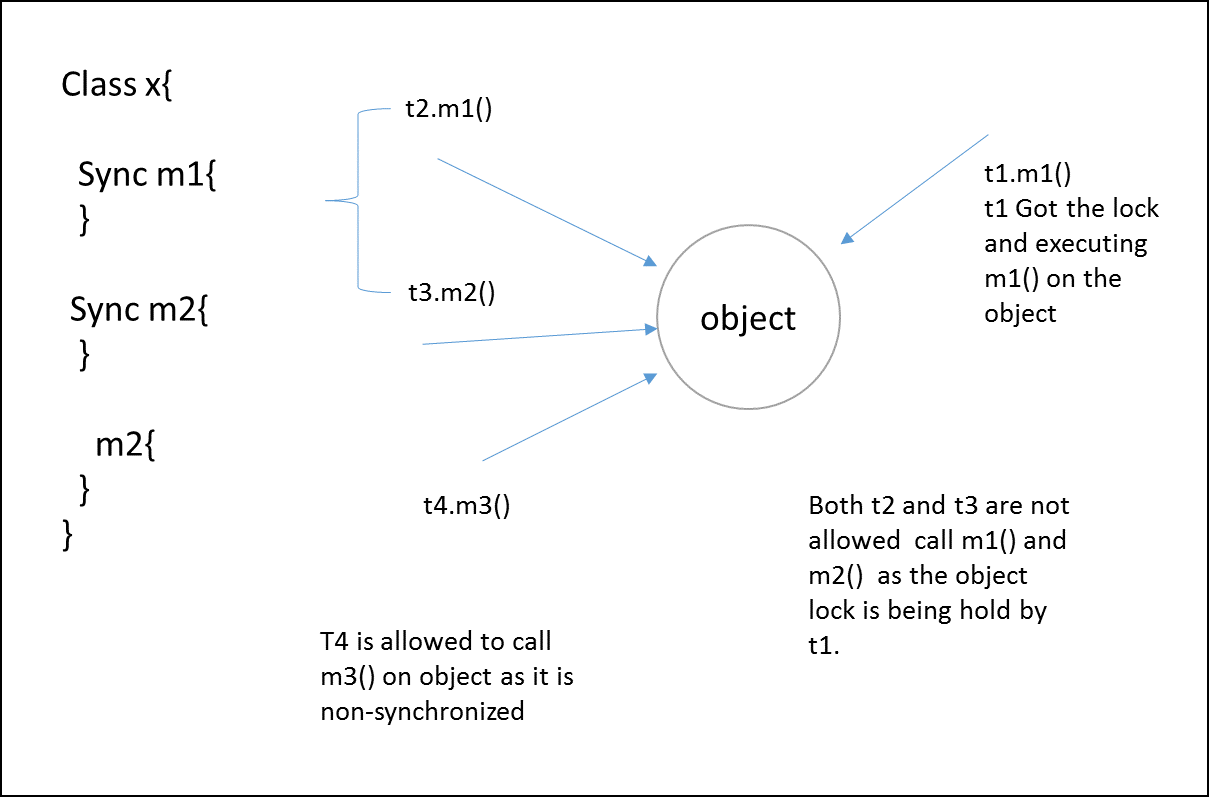
The main advantage of synchronized keyword is we can resolve data inconsistency problems. But the main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance problems. If there is no specific requirement then it is not recommended to use synchronized keyword.

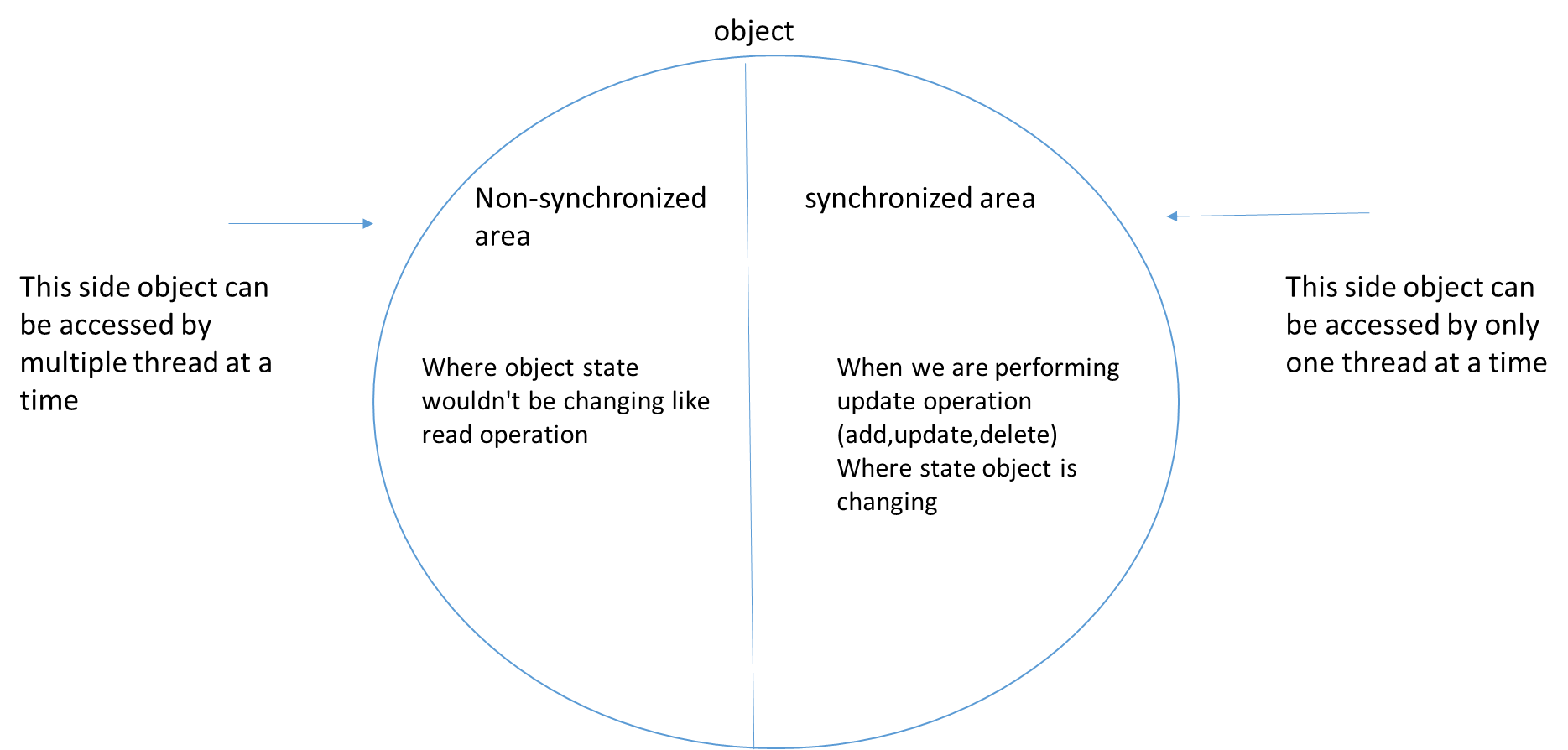
Internally synchronization concept is implemented using lock. Every object in java has a unique lock. Whenever we are using synchronized keyword then only Lock concept will come in to picture.

In a thread wants to execute synchronized method on the given object first thing it has to get lock of that object. Once thread got the lock then it is allowed to execute any synchronized method on that object. Once method execution completes automatically thread releases the lock. Acquiring and releasing lock internally taken care by JVM and programmer is not responsible for these activity

While a thread executing synchronized method on the given object the remaining threads are not allowed to execute any synchronized methods simultaneously on the same object but remaining threads are allowed to execute non-synchronized methods simultaneously.

Lock concept is implemented based on object but not based on methods.





**Class level lock**

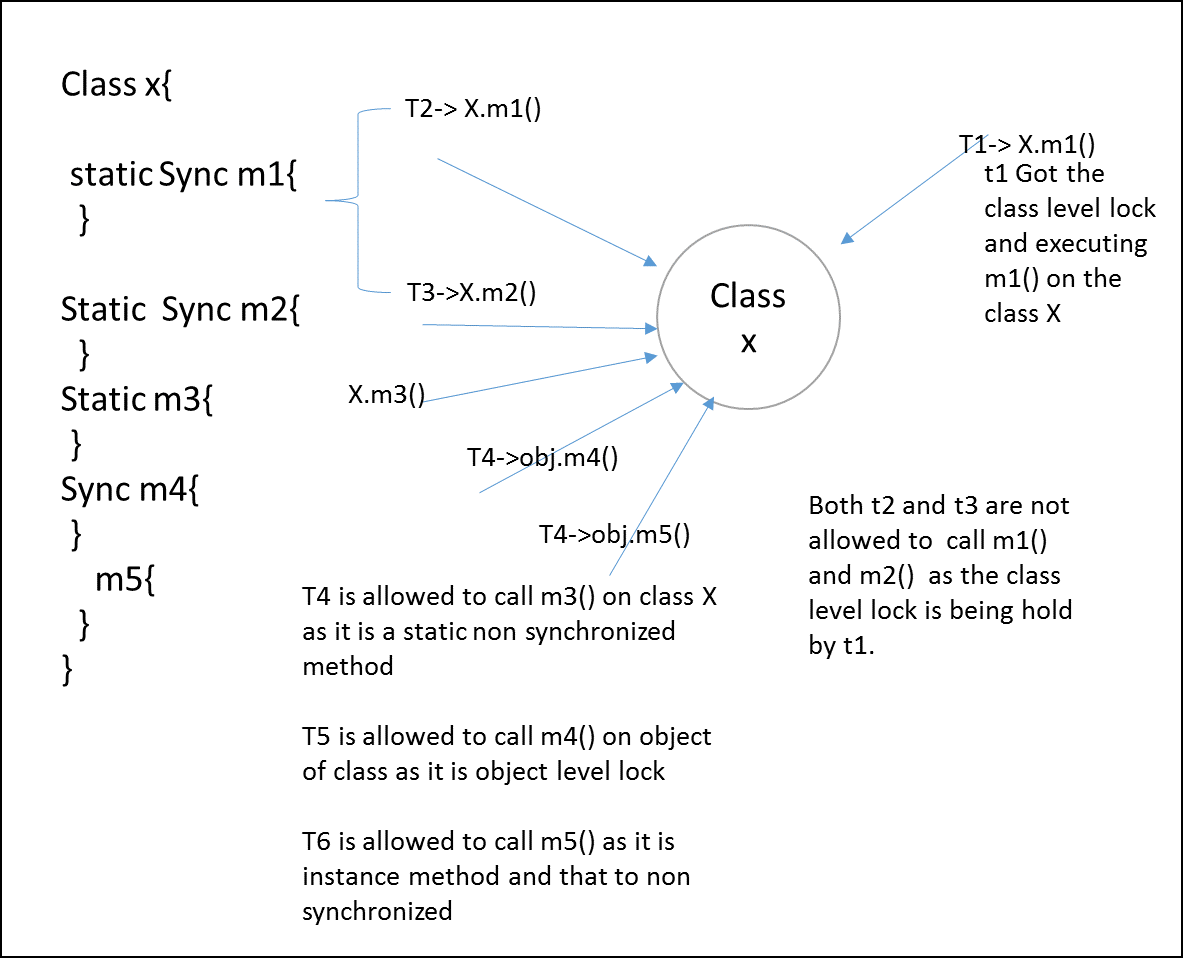
Every class in has a unique which is also known as class level lock.

If a thread wants to execute a static synchronized method then thread require class level lock.

Once thread got class level lock it is allowed to execute any static synchronized method of that class.

Once method execution completes automatically thread releases the lock.

While a thread executing static synchronized method the remaining threads are not allowed to execute any static synchronized method of that class simultaneously but remaining threads are allowed to execute the following methods simultaneously ,(normal static method, synchronized instance method, normal instance method)



**Synchronized block**

When very few lines in method requires synchronization then we should go for synchronized block.

The main benefit of using synchronization block is performance improvement (reduces the waiting time of the thread).

|  |  |  |
| --- | --- | --- |
| Synchronized(this) | Synchronized(b) | Synchronized(Display.class) |
| Synchronized(this){  ----------  ----------  }  If a thread got the lock of current object then only it is allowed to execute these lines of code. | Synchronized(b){  ----------  ----------  }  If a thread got the lock of some object b then only it is allowed to execute these lines of code. | Synchronized(b){  ----------  ----------  }  If a thread got the class level lock of display class then only it is allowed to execute these lines of code.  This synchronized block needs to be in static method |

We can write synchronized block as follows:-

To get lock of current object.

To get lock of particular object.

To get class level lock of particular class.

Lock concept is applicable to object types but not for primitives hence we can’t pass primitive type to synchronized block. Otherwise we will get compile time error.

Unexpected Type

If multiple threads are operating simultaneously on the same java object then there may be a chance of data in consistency problem. This is called Race condition. We can overcome this problem using synchronized keyword/modifier.

A thread can acquire multiple lock simultaneously but from different object.

Class X {

Public synchronized xyzMethod (){

Y y =new Y ();

Synchronized(y) {

Z z=new Z ();

Synchronized (z) {

Here Thread has lock of x,y,z

-----------------

-----------------

-----------------

}  
 }

}

}

**Inter Thread Communication**

wait (), notify(), notifyAll() present in Object class not in Thread class becoz thread can call these methods on any java objects.

To call wait() ,notify(),or notifyAll() method on any object thread should be owner of that object.

i.e. the thread should have lock of that object .i.e. The thread should be in synchronized area.

Hence we can call wait, notify, notifyAll only from synchronized area. Otherwise we will get runtime exception: IllegalMoniterStateException

If thread calls wait method on any object it immediately releases the lock of that particular object

And entered into waiting state.

If thread calls notify method any object it releases the lock of that object but may not immediately

Except wait,notify,notifyAll there is no other method where thread releases the lock.

public final native void wait(long timeout) throws InterruptedException

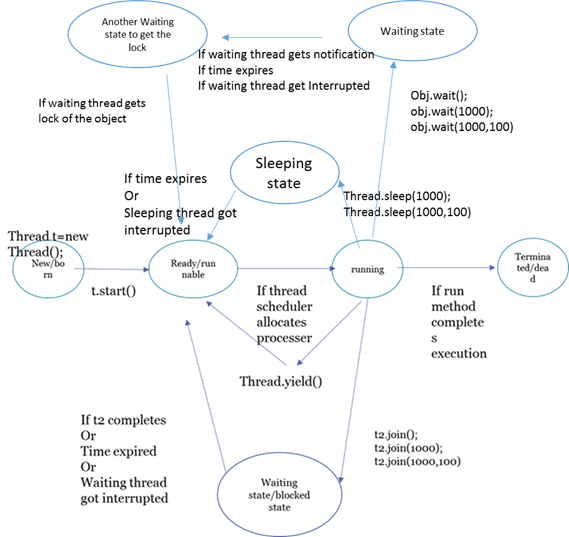
public final void wait(long timeout,int nanos) throws InterruptedException

public final void wait() throws InterruptedException

public final native void notify();

public final native void notifyAll();

Every Wait method throws InterruptedException which is checked exception so when we are using wait() method compulsory we should handle this exception either by try catch or throws keyword otherwise we will get compile time error.



**Producer consumer problem**

We can use notifyAll to give the notification for all waiting threads for a particular object .Even though multiple threads are notified but execution will occur one by one becoz threads require lock and only one lock is available for one object.

**Deadlock**

Synchronized keyword is the only reason for deadlock situation .hence while sync keyword we have to take special care.

There is resolution of deadlock but prevention techniques are available.

Long waiting of a thread where waiting never end is called deadlock.

Whereas long waiting of thread where waiting ends at certain point is called starvation.

Ex: low priority thread has to wait until completing all high priority threads.

This waiting may is long waiting but ends at certain point which is nothing but starvation.

The main objective of daemon threads is to provide support for non-demon threads. If Main

Thread runs with low memory then jvm runs garbage collector to destroy useless objects so that

So that free memory improved .with this free memory main thread can continue its execution.

**Daemon Thread**

Example:

Usually Daemon threads have low priority but based on our requirement daemon threads can run with high priority also.

Public boolean isDaemon()

Public void setDaemon(boolean b)

we can check daemon nature of a thread by using isDaemon() method of Thread class.

We can change daemon nature of a thread by using setDaemon(boolean b);

But changing daemon nature is possible before starting of a thread only. After starting a thread if we are trying to change daemon nature we will get runtime exception IllegalThreadStateException.

By default Main thread is always non daemon and for all remaining threads daemon nature will be inherited from parent to child. i.e. If parent thread is daemon then automatically child thread is daemon . If parent thread is non-daemon then automatically child thread is non-daemon.

It is impossible to change daemon nature of Main thread becoz it is already started my JVM in the beginning.

If child thread is daemon as soon as main thread terminates child thread also terminates.

**Green Thread**

In computer programming, **green threads** are **threads** that are scheduled by a runtime library or virtual machine (VM) instead of natively by the underlying operating system.

Example : Solaris os.

What is the difference between green and native threads?

Green threads are scheduled by a virtual machine.

Native threads are scheduled by a operational system.

Why does it named as green and native?

"Green" is earlier JVM threads project code-name. It is name of library, which provided VM-scheduled threads in Java 1.1

Native threads called so because they're belong to native platform.

How do we know that created thread is native or green?

Green threads are in past, JVMs work only with native threads since 1.3

"Green threads" refers to a model in which the Java virtual machine itself creates, manages, and context switches all Java threads within one operating system process. No operating system threads library is used.

"Native threads" refers to a in which the Java virtual machine creates and manages Java threads using the operating system threads library - named thread on UnixWare - and each Java thread is mapped to one threads library thread.

How to stop a thread?

We can stop a thread execution by using stop method of thread class

Public void stop();(depricated)

If we call stop method then immediately thread will enter into dead state.

Anyway stop method is deprecated and not recommended to use.

How to suspend and resume a thread?

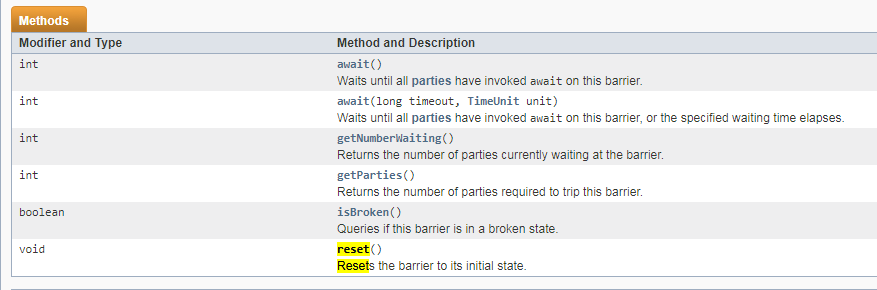
Public void suspend();(depricated)

Public void resume();(depricated)

We can suspend a thread by calling suspend method then immediately the thread will be entered in to suspended state. We can resume a thread by calling resume method, then suspended thread continue its execution.

These methods are deprecated and not recommended to use.

**Cyclic Barrier**



The java.util.concurrent.CyclicBarrier class is a synchronization mechanism that can synchronize threads progressing through some algorithm. In other words, it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue. Here is a diagram illustrating that:

Two threads waiting for each other at CyclicBarriers.

Two threads waiting for each other at CyclicBarriers.

The threads wait for each other by calling the await () method on the CyclicBarrier. Once N threads are waiting at the CyclicBarrier, all threads are released and can continue running.

Creating a CyclicBarrier

When you create a CyclicBarrier you specify how many threads are to wait at it, before releasing them. Here is how you create a CyclicBarrier:

CyclicBarrier barrier = new CyclicBarrier(2);

Waiting at a CyclicBarrier

Here is how a thread waits at a CyclicBarrier:

**barrier.await();**

You can also specify a timeout for the waiting thread. When the timeout has passed the thread is also released, even if not all N threads are waiting at the CyclicBarrier. Here is how you specify a timeout:

**barrier.await(10, TimeUnit.SECONDS);**

The waiting threads waits at the CyclicBarrier until either:

The last thread arrives (calls await() )

The thread is interrupted by another thread (another thread calls its interrupt() method)

Another waiting thread is interrupted

Another waiting thread times out while waiting at the CyclicBarrier

The **cyclicBarrier.reset ()** method is called by some external thread.

CyclicBarrier Action

The CyclicBarrier supports a barrier action, which is a Runnable that is executed once the last thread arrives. You pass the Runnable barrier action to the CyclicBarrier in its constructor, like this:

Runnable barrierAction = ... ;

CyclicBarrier barrier = new CyclicBarrier(2, barrierAction);

CyclicBarrier Example

Here is a code example that shows you how to use a CyclicBarrier:

Runnable barrier1Action = new Runnable() {

public void run() {

System.out.println("BarrierAction 1 executed ");

}

};

Runnable barrier2Action = new Runnable() {

public void run() {

System.out.println("BarrierAction 2 executed ");

}

};

CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);

CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);

CyclicBarrierRunnable barrierRunnable1 =

new CyclicBarrierRunnable(barrier1, barrier2);

CyclicBarrierRunnable barrierRunnable2 =

new CyclicBarrierRunnable(barrier1, barrier2);

new Thread(barrierRunnable1).start();

new Thread(barrierRunnable2).start();

Here is the CyclicBarrierRunnable class:

public class CyclicBarrierRunnable implements Runnable{

CyclicBarrier barrier1 = null;

CyclicBarrier barrier2 = null;

public CyclicBarrierRunnable(

CyclicBarrier barrier1,

CyclicBarrier barrier2) {

this.barrier1 = barrier1;

this.barrier2 = barrier2;

}

public void run() {

try {

Thread.sleep (1000);

System.out.println (Thread.currentThread ().getName () +

" Waiting at barrier 1");

this.barrier1.await ();

Thread.sleep(1000);

System.out.println (Thread.currentThread ().getName () +

" waiting at barrier 2");

this.barrier2.await ();

System.out.println(Thread.currentThread().getName() +

" done!");

} catch (InterruptedException e) {

e.printStackTrace ();

} catch (BrokenBarrierException e) {

e.printStackTrace ();

}

}

}

Here is the console output for an execution of the above code. Note that the sequence in which the threads gets to write to the console may vary from execution to execution. Sometimes Thread-0 prints first, sometimes Thread-1 prints first etc.

Thread-0 waiting at barrier 1

Thread-1 waiting at barrier 1

BarrierAction 1 executed

Thread-1 waiting at barrier 2

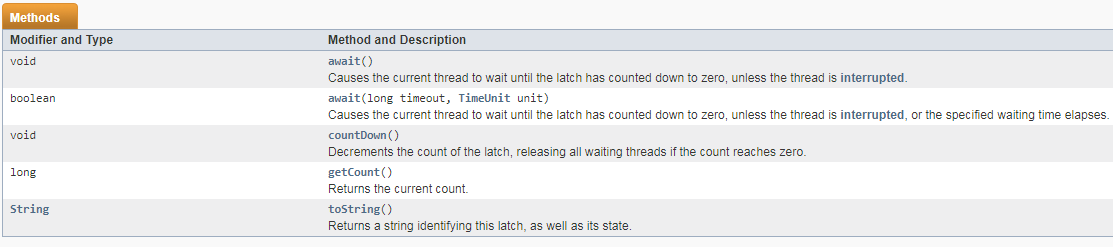
Thread-0 waiting at barrier 2

BarrierAction 2 executed

Thread-0 done!

Thread-1 done!

**CountDownLatch**



CountDownLatch is used to make sure that a task waits for other threads before it starts. To understand its application, let us consider a server where the main task can only start when all the required services have started.

Working of CountDownLatch:

When we create an object of CountDownLatch, we specify the number if threads it should wait for, all such thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. As soon as count reaches zero, the waiting task starts running.

Example of CountDownLatch in JAVA:

/\* Java Program to demonstrate how to use CountDownLatch,

Its used when a thread needs to wait for other threads

before starting its work. \*/

import java.util.concurrent.CountDownLatch;

public class CountDownLatchDemo

{

public static void main(String args[]) throws InterruptedException

{

// Let us create task that is going to wait for four

// threads before it starts

CountDownLatch latch = new CountDownLatch(4);

// Let us create four worker threads and start them.

Worker first = new Worker(1000, latch, "WORKER-1");

Worker second = new Worker(2000, latch, "WORKER-2");

Worker third = new Worker(3000, latch, "WORKER-3");

Worker fourth = new Worker(4000, latch, "WORKER-4");

first.start();

second.start();

third.start();

fourth.start();

// The main task waits for four threads

latch.await();

// Main thread has started

System.out.println (Thread.currentThread ().getName() +

" has finished");

}

}

// A class to represent threads for which the main thread

// waits.

class Worker extends Thread

{

private int delay;

private CountDownLatch latch;

public Worker(int delay, CountDownLatch latch,

String name)

{

super(name);

this.delay = delay;

this.latch = latch;

}

@Override

public void run()

{

try

{

Thread.sleep(delay);

latch.countDown();

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

+ " finished");

}

catch (InterruptedException e)

{

e.printStackTrace ();

}

}

}

Run on IDE

Output:

WORKER-1 finished

WORKER-2 finished

WORKER-3 finished

WORKER-4 finished

main has finished

Facts about CountDownLatch:

Creating an object of CountDownLatch by passing an int to its constructor (the count), is actually number of invited parties (threads) for an event.

The thread, which is dependent on other threads to start processing, waits on until every other thread has called count down. All threads, which are waiting on await () proceed together once count down reaches to zero.

CountDown () method decrements the count and await () method blocks until count == 0

**Semaphore**

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

Working of semaphore

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented.

Otherwise, the thread will be blocked until a permit can be acquired.

When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented.

If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

Java provide Semaphore class in java.util.concurrent package that implements this mechanism, so you don’t have to implement your own semaphores.

Constructors in Semaphore class : There are two constructors in Semaphore class.

Semaphore(int num)

Semaphore(int num, boolean how)

Here, num specifies the initial permit count. Thus, it specifies the number of threads that can access a shared resource at any one time. If it is one, then only one thread can access the resource at any one time. By default, all waiting threads are granted a permit in an undefined order. By setting how to true, you can ensure that waiting threads are granted a permit in the order in which they requested access.

Using Semaphores as Locks(preventing race condition)

We can use a semaphore to lock access to a resource, each thread that wants to use that resource must first call acquire( ) before accessing the resource to acquire the lock. When the thread is done with the resource, it must call release( ) to release lock. Here is an example that demonstrate this:

// java program to demonstrate

// use of semaphores Locks

import java.util.concurrent.\*;

//A shared resource/class.

class Shared

{

static int count = 0;

}

class MyThread extends Thread

{

Semaphore sem;

String threadName;

public MyThread(Semaphore sem, String threadName)

{

super(threadName);

this.sem = sem;

this.threadName = threadName;

}

@Override

public void run() {

// run by thread A

if(this.getName().equals("A"))

{

System.out.println("Starting " + threadName);

try

{

// First, get a permit.

System.out.println(threadName + " is waiting for a permit.");

// acquiring the lock

sem.acquire();

System.out.println(threadName + " gets a permit.");

// Now, accessing the shared resource.

// other waiting threads will wait, until this

// thread release the lock

for(int i=0; i < 5; i++)

{

Shared.count++;

System.out.println(threadName + ": " + Shared.count);

// Now, allowing a context switch -- if possible.

// for thread B to execute

Thread.sleep(10);

}

} catch (InterruptedException exc) {

System.out.println(exc);

}

// Release the permit.

System.out.println(threadName + " releases the permit.");

sem.release();

}

// run by thread B

else

{

System.out.println("Starting " + threadName);

try

{

// First, get a permit.

System.out.println(threadName + " is waiting for a permit.");

// acquiring the lock

sem.acquire();

System.out.println(threadName + " gets a permit.");

// Now, accessing the shared resource.

// other waiting threads will wait, until this

// thread release the lock

for(int i=0; i < 5; i++)

{

Shared.count--;

System.out.println(threadName + ": " + Shared.count);

// Now, allowing a context switch -- if possible.

// for thread A to execute

Thread.sleep(10);

}

} catch (InterruptedException exc) {

System.out.println(exc);

}

// Release the permit.

System.out.println(threadName + " releases the permit.");

sem.release();

}

}

}

// Driver class

public class SemaphoreDemo

{

public static void main(String args[]) throws InterruptedException

{

// creating a Semaphore object

// with number of permits 1

Semaphore sem = new Semaphore(1);

// creating two threads with name A and B

// Note that thread A will increment the count

// and thread B will decrement the count

MyThread mt1 = new MyThread(sem, "A");

MyThread mt2 = new MyThread(sem, "B");

// stating threads A and B

mt1.start();

mt2.start();

// waiting for threads A and B

mt1.join();

mt2.join();

// count will always remain 0 after

// both threads will complete their execution

System.out.println("count: " + Shared.count);

}

}

Run on IDE

Output:

Starting A

Starting B

B is waiting for a permit.

B gets a permit.

A is waiting for a permit.

B: -1

B: -2

B: -3

B: -4

B: -5

B releases the permit.

A gets a permit.

A: -4

A: -3

A: -2

A: -1

A: 0

A releases the permit.

count: 0

Note : The output can be different in different executions of above program, but final value of count variable will always remain 0.

Explanation of above program :

The program uses a semaphore to control access to the count variable, which is a static variable within the Shared class. Shared.count is incremented five times by thread A and decremented five times by thread B.To prevent these two threads from accessing Shared.count at the same time, access is allowed only after a permit is acquired from the controlling semaphore. After access is complete, the permit is released. In this way, only one thread at a time will access Shared.count, as the output shows.

Notice the call to sleep( ) within run( ) method inside MyThread class. It is used to “prove” that accesses to Shared.count are synchronized by the semaphore. In run( ), the call to sleep( ) causes the invoking thread to pause between each access to Shared.count. This would normally enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete. Thus, Shared.count is first incremented five times by thread A and then decremented five times by thread B. The increments and decrements are not intermixed at assembly code.

Without the use of the semaphore, accesses to Shared.count by both threads would have occurred simultaneously, and the increments and decrements would be intermixed.To confirm this, try commenting out the calls to acquire( ) and release( ). When you run the program, you will see that access to Shared.count is no longer synchronized, thus you will not always get count value 0.