**Multithreading in Java**

**What is multitasking?**

Executing several tasks simultaneously is a concept of Multitasking. *Multitasking* is the ability of a computer's operating system to run several programs (or *processes*) concurrently on a single CPU.  This is done by switching from one program to another fast enough to create the appearance that all programs are executing simultaneously. There are two types of multitasking

1. Process based multitasking.
2. Thread based multitasking.

**Process based multitasking:**

Executing several tasks simultaneously where each task is **a separate independent program(process)** is called process-based multitasking.

Example:

While typing a java program in Editor we can listen audio songs from same system at the same time we can download a file from browser all these tasks will be executed simultaneously and independent of each other hence it is processed based multitasking.

Processed based multitasking is best suitable at OS level.

**Thread based multitasking:**

Executing several tasks simultaneously where each task is **a** **separate independent part of the same program** is called thread-based Multitasking and each independent part is called a Thread.

Thread based multitasking is best suitable at programmatical level for Programmatic level.

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

Main important application areas of multithreading are

1. To develop multimedia graphics.
2. To develop video games.
3. To develop Web servers and Application servers

And many more.

**What is a Thread?**

 A thread in Java is **the direction or path that is taken while a program is being executed**. Generally, all the programs have at least one thread, known as the **main thread**, that is provided by the JVM or Java Virtual Machine at the starting of the program's execution. At this point, when the main thread is provided, the main() method is invoked by the main thread.

**Creating a Thread in Java.**

A Thread in java can be created in two ways:

1. **Extending java.lang.Thread class**

**// Definition of the thread**

**class** GFG **extends** Thread {

// This part is JOB of the child thread

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

System.***out***.print("Welcome to Threads.");

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

System.***out***.println("Child Thread !!");

}

}

}

**class** ThreadsDemo {

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

MyThread t = **new** MyThread();// Thread instantiation

t.start(); // starting of the thread

// t.run();

// At this point we have two threads one is **Main thread** and one is **child thread**

// This part is always executed by **main thread**

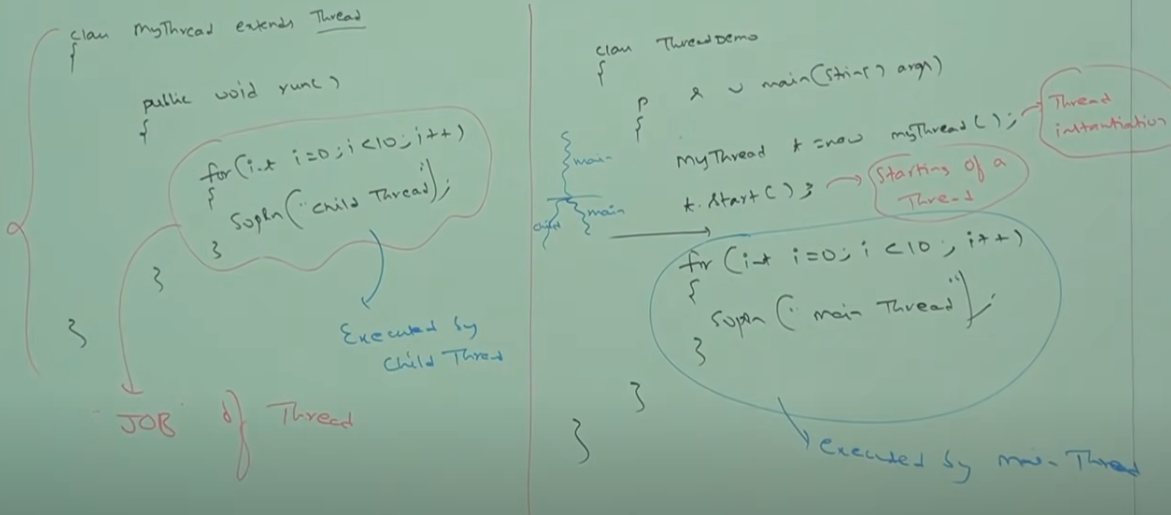
**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Main Thread !!");

}

}

}



**Thread Schedular**

It’s a part of JVM it is responsible to schedule threads i.e., if multiple threads are waiting to get the chance of execution, then in which order threads will be executed is decided by Thread schedular.

We can’t expect exact algorithm fallowed by Thread schedular it varies from JVM to JVM hence we can’t expect thread execution order and execute.

Hence whenever situation coms to MT then there is no guarantee for exact output but we can provide several possible outputs.

Different cases on Thread.

1. If we use ***thread.start()*** method a new child thread is created which is responsible for execution of run() method. Here we always have a chance to get different output because both child thread and main thread is executed.

Responsibilities of start() method

* Register the Thread with the Thread Schedular.
* Perform all the other mandatory activities.
* Invoke **run()** method of Thread class.// start will always invoke run() with no arguments.

1. If we override a start() method of Thread class then no new child thread is created
2. If we use **thread.run()** method new child thread is **not created** and main thread executes run() method normally. Here output is always same because *even run() is always executed by main thread*.
3. Creating Thread by implementing Runnable interface.

**Runnable interface:**

Any class with instances that are intended to be executed by a thread should implement the Runnable interface. The Runnable interface has only one method, which is called run().

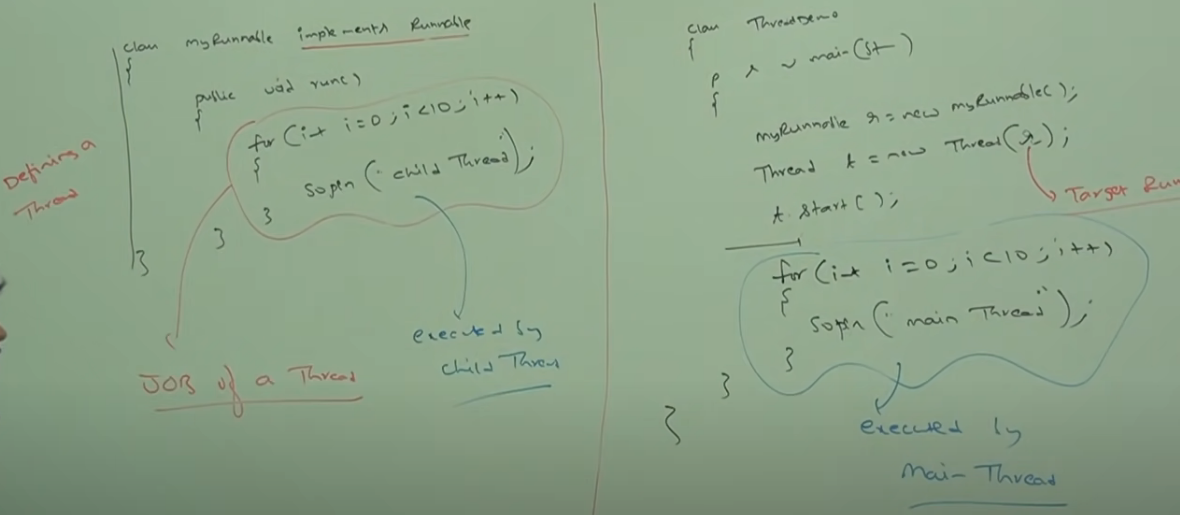
// Thread action is performed

public void run()

**Benefits of creating threads :**

* When compared to processes, Java Threads are more lightweight; it takes less time and resources to create a thread.
* Threads share the data and code of their parent process.
* Thread communication is simpler than process communication.
* Context switching between threads is usually cheaper than switching between processes.

**The Runnable interface is a functional interface defined in java.** **lang package**. This interface contains a single abstract method, run() with no arguments. When an object of a class implementing this interface used to create a thread, then run() method has invoked in a thread that executes separately



**class** MyRunnable **implements** Runnable {

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

System.***out***.print("Welcome to Threads.");

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

System.***out***.println("Child Thread !!");

}

}

}

**class** ThreadDemo {

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

MyRunnable runnable = **new** MyRunnable();

// We need to pass the target runnable to invoke the run() method we want else

// on start of the thread empty implementation of run() is called.

Thread myThread = **new** Thread(runnable);

myThread.start();

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

System.***out***.println("Main Thread !!");

}

}

}

If your class is *extending the Thread class* then it becomes a single thread which inherits the properties *Thread class*, so it'll be heavy. (When *extending Thread class* each of the threads creates unique object and associate with it, but when *implementing Runnable*, it shares the same object to multiple Threads).

**Calling run() instead of start()**

The common mistake is starting a thread using run() instead of start() method.

Thread myThread = new Thread(MyRunnable());

myThread.run(); //should be start();

The run() method is not called by the thread you created. Instead, it is called by the thread that created the **myThread(Main Thread)**.

Which is the best way to create a Thread?

Thread creation by extending Thread class: As we are extending a Thread class, we will not be able to inherit any other class and because of this we are inheritance benefit in this approach.

Thread creation by implementing Runnable Interface: Along with implementing Runnable we can also use extends to inherit any other classes so we have Inheritance benefit. Hence implementing Runnable is recommended to define a thread.

Thread class Constructors

Thread t = new Thread();

Thread t = new Thread(Runnable impl);

Thread t = new Thread(String name);

Thread t = new Thread(Runnable impl,String name);

Thread t = new Thread(ThreadGroup g, String name);

Thread t = new Thread(ThreadGroup g, Runnable impl);

Thread t = new Thread(ThreadGroup g, Runnable impl,String name);

Thread t = new Thread(ThreadGroup g, Runnable impl,String name,long stackSize);

Getting and setting Name of a Thread

1. Every Thread in java has some name it may be default name generated by JVM or customized name provided by programmer.
2. We can get and set name of a Thread by using fallowing two methods of Thread class

Public final String getName();

Public final void setName(String name);

1. Thread.*currentThread*().getName() is used to get the name of the current executing thread.

**class** M15

{

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

{

Thread t1 = Thread.*currentThread*();

System.***out***.println("id:"+ t1.getId());//v cannot change the id of the thread

System.***out***.println("name:"+ t1.getName());

System.***out***.println("demon:"+ t1.isDaemon());

System.***out***.println("priority:"+ t1.getPriority());

t1.setPriority(10);

t1.setPriority(11);// RE: IllegalArgumentException

System.***out***.println("domain:"+ t1.isDaemon());

t1.setDaemon(**true**);

System.***out***.println("name:"+ t1.isDaemon());

}

}

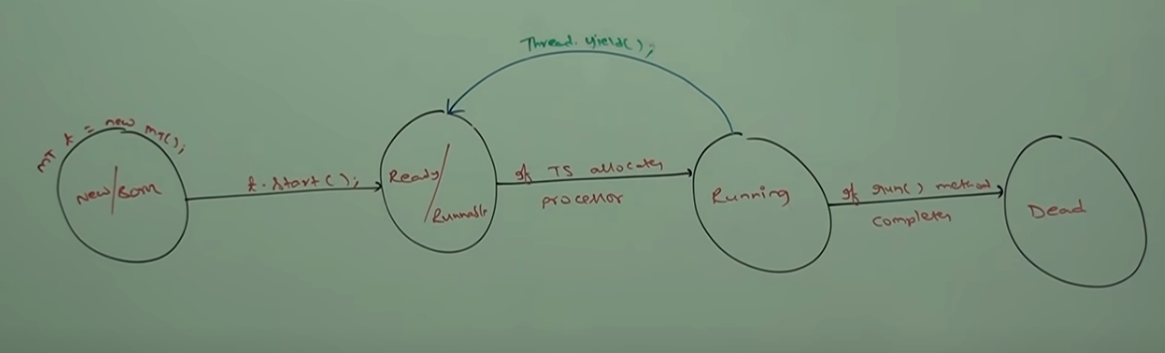
* setDemon method is used to convert a thread into demon
* We cannot convert main Thread into a demon thread.
* priority is an integer number with minimum value 1 and Maximum 10. Through set priority we can change the priority of the thread.public static int MIN\_PRIORITY,public static int NORM\_PRIORITY,public static int MAX\_PRIORITY. Default priority only for the main thread is 5 but for all remaining thread the default priority is inherited from Parent.
* Thread schedular will use the priorities while allocation processor, thread which is having highest priority will get the chance first. If two threads having same priority then we can’t expect exact execution order it depends on thread schedular.

Prevent a thread execution: By using this fallowing three methods we can prevent a thread execution

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

**Yield Method**: yield method causes to pause a current executing thread to give the chance for waiting threads of same priority. if there is no waiting or all waiting threads have low priority then same thread can continue its execution.

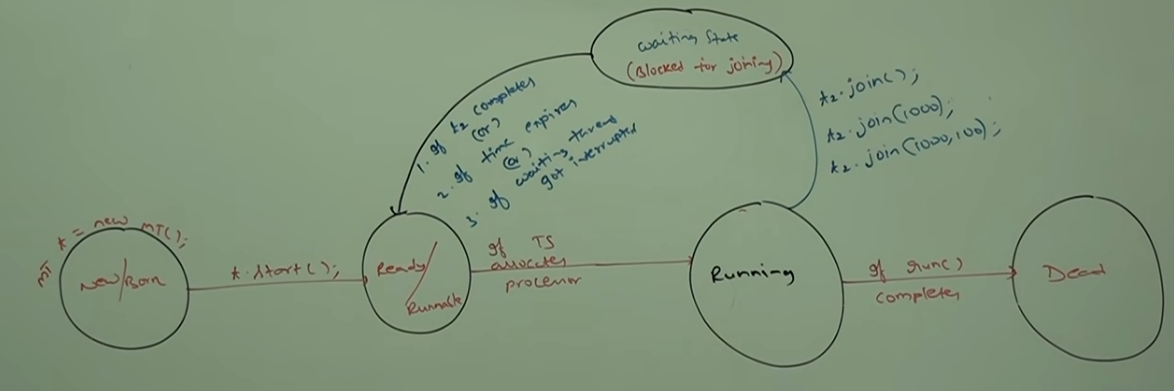
If multiple threads are waiting with same priority which waiting thread will get a change? We can’t expect it depends on thread schedular.



**Join():** If a thread wants to wait until completing some other thread then we should go for join method.

For example: if a Thread t1 wants to wait until completing t2 then t1 has to call t2.join(). If t1 executes t2.join() then immediately t1 will be entered into waiting state until t2 completes once t2 completes once then t1 can continue its execution.

Every join methods throws InterreptedException which is checked exception hence we need to handle either by using throws or try catch() block.



CASE 1 : Waiting of main thread until completing chaild thread

**class** A **extends** Thread {

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

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

System.***out***.println("child:" + i);

}

}

}

**class** M27 {

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

A a1 = **new** A();

a1.start();

**try** {

a1.join();// Main thread is calling join means main thread wants to wait untill child thread is completing its execution

} **catch** (InterruptedException ex) {

ex.printStackTrace();

}

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

System.***out***.println("main:" + i);

}

System.***out***.println("main end");

}

}

CASE 2 : Waiting of Child thread until completing main thread

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

**public** **static** Thread *thread*;

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

**try** {

*thread*.join();

**for** (**int** i = 0; i < 300; i++) {

System.***out***.println("child Thread" + i);

Util.*sleep*(3000);

}

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**class** JoinDemo {

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

MyThread.*thread* = Thread.*currentThread*();

MyThread a1 = **new** MyThread();

a1.start();

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

System.***out***.println("Main Thread !!");

}

}

}

CASE 3: If main thread calls join() method on child thread object and child thread calls join method on main thread object then both the threads will forever and the program will be stuck (This is something like **DEADLOCK** situation).

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

**public** **static** Thread *thread*;

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

**try** {

*thread*.join();

**for** (**int** i = 0; i < 300; i++) {

System.***out***.println("child Thread" + i);

Util.*sleep*(3000);

}

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**class** JoinDemo {

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

MyThread.*thread* = Thread.*currentThread*();

MyThread a1 = **new** MyThread();

a1.start();

a1.join();

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

System.***out***.println("Main Thread !!");

}

}

}

CASE 4: If a thread calls join on same thread it self then the program will be stuck this some thing like **DEADLOCK**

**class** Test {

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

**try** {

Thread.*currentThread*().join();

// Thread has to wait for infinite amount of time

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

Sleep():

The **sleep()**method is a **static**method of **Thread**class and it makes the thread **sleep/stop** working for a specific amount of time. The sleep() method throws an **InterruptedException**if a thread is interrupted by other threads, that means **Thread.sleep()** method must be enclosed within the t**ry and catch blocks** or it must be specified with **throws clause**. Whenever we call the **Thread.sleep()** method, it can interact with the **thread scheduler** to put the current thread to a **waiting state** for a specific period of time. Once the waiting time is over, the thread changes from **waiting**state to **runnable**state.

Syntax

public static void sleep(long milliseconds)

public static void sleep(long milliseconds, int nanoseconds)

**Example ::**

**class** M28

{

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

{

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

{

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

**try**

{

Thread.*sleep*(3000);

}

**catch** (InterruptedException ex)

{

ex.printStackTrace();

}

}

}

}

//join ,sleep and wait method requires a InterruptedException

//1000 milli sec is 1sec; for every iteration stops for 1 sec

**class** A **extends** Thread

{

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

{

System.***out***.println("child:begin");

Util.*sleep*(10000);

System.***out***.println("child:end");

}

}

**class** M35

{

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

{

A a1 = **new** A();

a1.start();

a1.start();

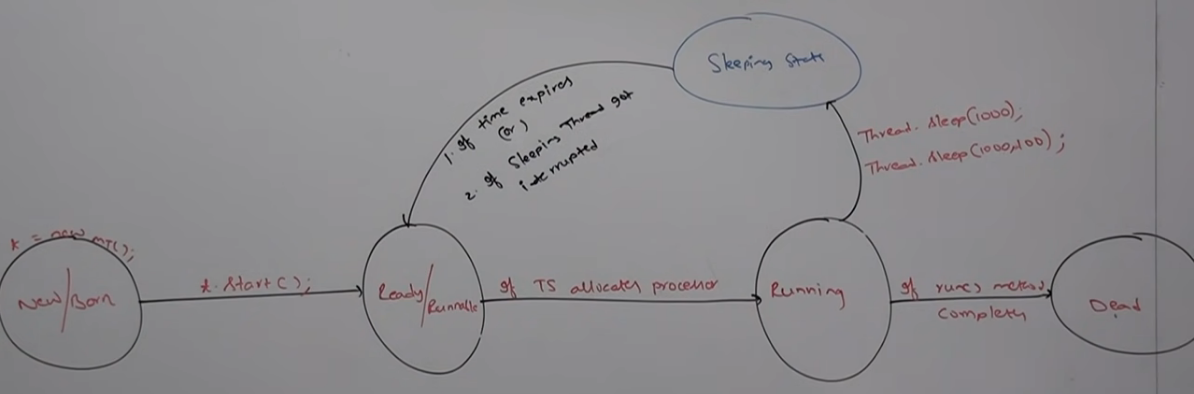
//CTE same thread should not be started second time

//thats an abnormal condition exception object is raised

System.***out***.println("main end") ;

}

}



**class** A **extends** Thread {

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

**try** {

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

System.***out***.println("child Thread");

Thread.*sleep*(10000);

}

} **catch** (InterruptedException ex) {

System.***out***.println("Got Interupted exception !!");

ex.printStackTrace();

}

System.***out***.println("child:end");

}

}

**class** M35 {

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

A a1 = **new** A();

a1.start();

a1.interrupt();// As soon as the child thread enters into the sleeping child thread is interrupted and InterruptedException is caught.

System.***out***.println("main end");

}

}

Output:

Main end

Child Thread

Got Interupted exception !!

If the target thread not in sleeping state or waiting state then there is no impact of interrupt call immediately. Interrupt call will be waited until target thread entered into waiting or sleeping state. If the target thread never entered into sleeping or waiting state in its life time then there is not impact of interrupt this is the only case where interrupt call will be wasted.

**Synchronized Keyword**

Synchronized is the modifier applicable only for methods and blocks but not for Class and variables.

If multiple threads are trying to operate simultaneously on the same java object, then there may be chance of Data inconsistency problem. To, overcome this problem, we should go for synchronized key word.

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

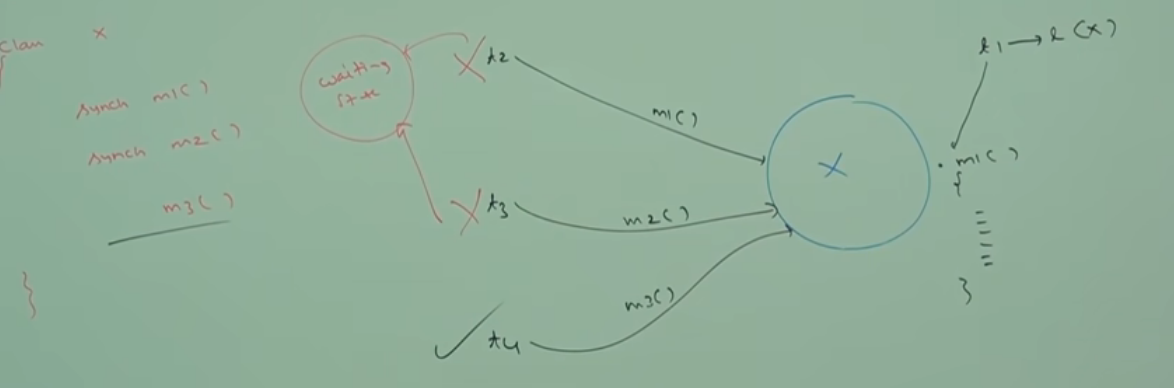
The main advantage of synchronized key word is we can resolve data inconsistency problems but the main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance problem.

Hence if there is not specific requirement then its not recommended to use synchronized keyword.

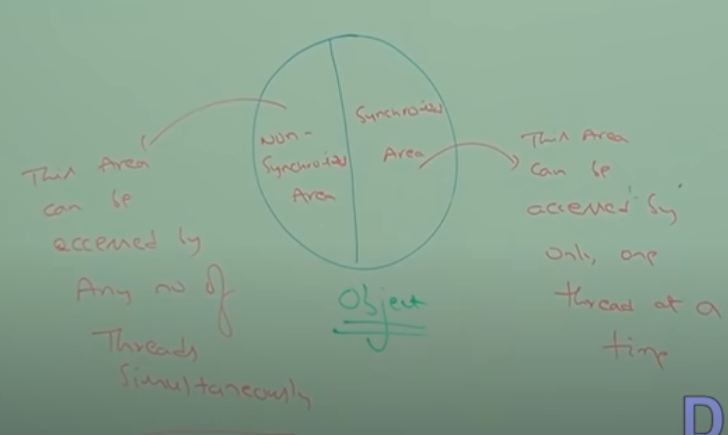
Internally synchronization concept is implemented by using lock. Every Object in java has a unique lock. When ever we using synchronized keywork then only lock concept comes into picture.

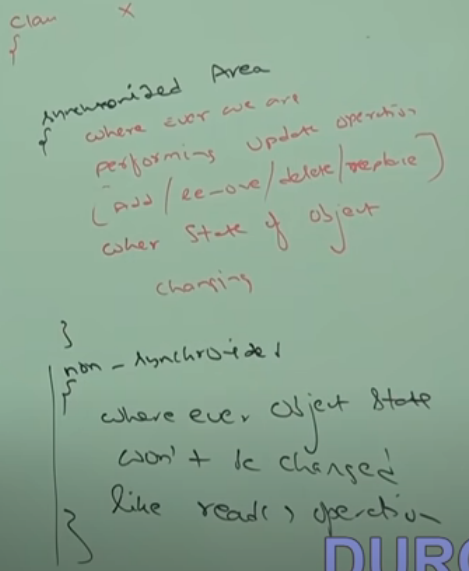
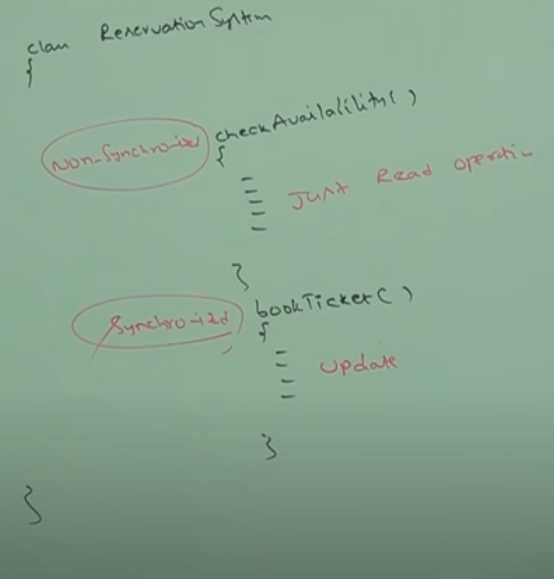
If a thread wants to execute synchronized method on the given object first it has to locj of that object. Once the thread got the lock then it is allowed to execute any synchronized method on that object. Once method execution completes automatically thread releases the Lock.

Accruing and releasing lock internally takes care by JVM and programmer is not responsible for this activity.



While a thread executing synchronized method on the given object, the remaining threads are not allowed to execute any synchronized method on the same object. But remaining threads are allowed to execute no synch method.



Example:

Multiple threads witout synchronized block which gives mixed output

**class** Display {

**public** **void** displayn() {

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

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

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**public** **void** displayc() {

**for** (**int** i = 65; i <= 75; i++) {

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

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

**class** myThread1 **extends** Thread {

Display d;

**public** myThread1() {

}

**public** myThread1(Display d) {

**this**.d = d;

}

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

d.displayn();

}

}

Output:

AB1CD23E4…

Multiple threads with synchronized block which gived sequential output oneaftre the other.

**class** Display {

**public synchronized** **void** displayn() {

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

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

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**public** **synchronized void** displayc() {

**for** (**int** i = 65; i <= 75; i++) {

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

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

**class** myThread1 **extends** Thread {

Display d;

**public** myThread1() {

}

**public** myThread1(Display d) {

**this**.d = d;

}

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

d.displayn();

}

}

Output:

123456789ABCDEFGHIJK

If multiple threads are operating simultaneously on the same java object then there may be chance of data inconsistency problem this is called Race Condition. We can over come this problem by using synchronized keyword

Synchronized Block

If very few lines of the code require synchronization, then it’s not recommended to declare entire method as syn. We have to enclose those few lines of the code by using synchronized block. The main advantage of Synchronized block over Synchronized method is it reduces waiting time of the thread and improve the performance of the system.

We can declare synchronized block as fallows.

1. To get lock of current object.

Synchronized(this){

}

1. To get lock of particular object.

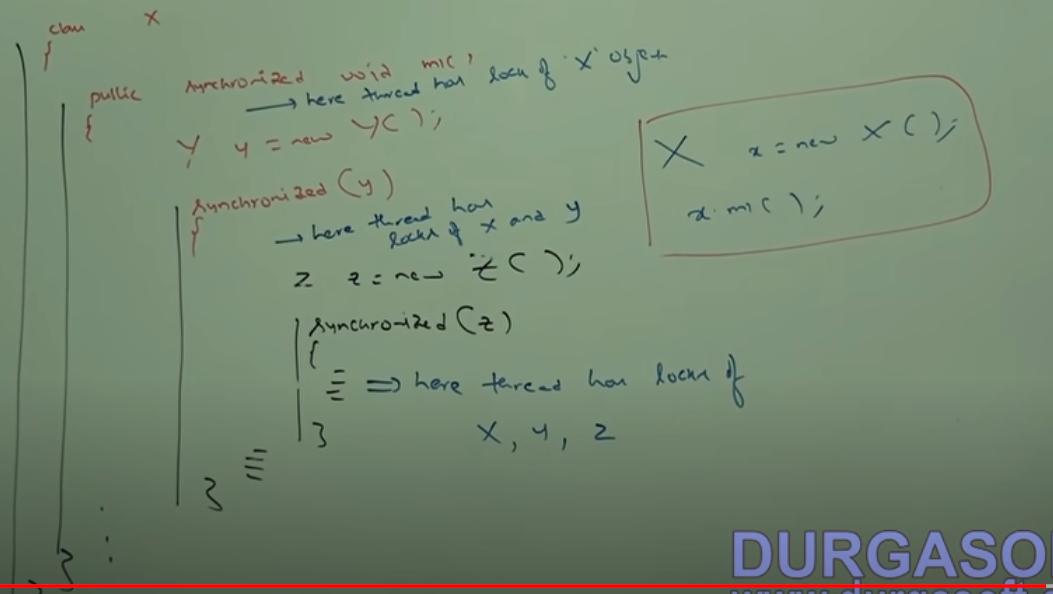
Synchronized (b) {

}

1. To get lock of Class level

Synchronized (Display.class) {

}



A single Thread can have multiple locks at a time.

**Inter Thread Communication**.

Two threads can communicate with each other by using wait(), notify() and notifyAll() methods.

The thread which is expecting update is responsible to call wait() method then immediately the thread will enter into waiting state.

The Thread which is responsible to perform update, after performing update it is responsible to call notify() then waiting Thread will get that notification and continue its execution with those updated items.

Wait(), notify() and notifuAll() methods are present in Object class but not in Thread class, because Thread can call this methods on any Java Object.

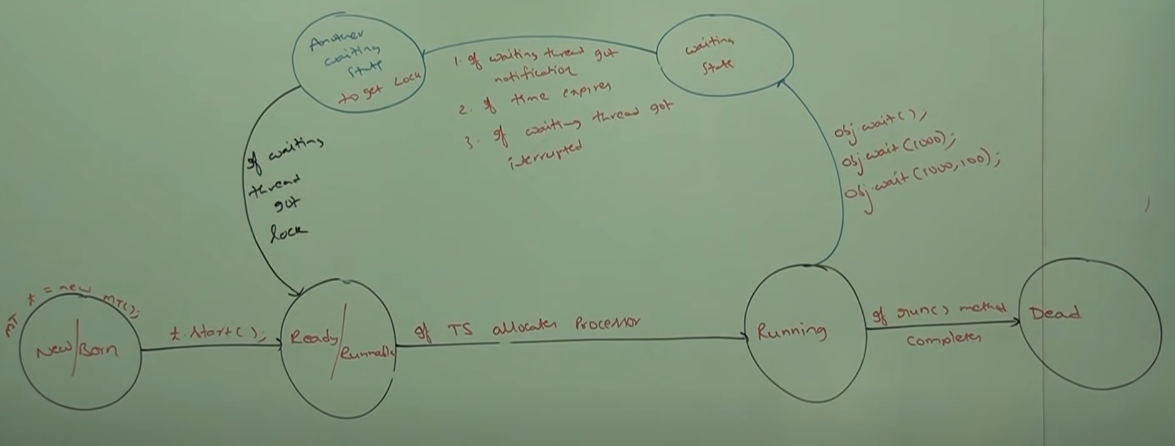
To call wait notify notifyAll methods on any object, thread should be owner of that Object i.e., thread should have lock of that object that is the thread should be inside synchronized area. Hence, we can call wait notify notify methods only from synchronized area otherwise we will get RTE IllegalMonitorStateException.

If a thread calls wait() on any object it immediately releases the lock of that particular object and enter into waiting state.

If a thread calls notify() on any object it releases lock but may not be immediately.

Except wait() notify() notifyAll() there is not other method where thread releases lock.(yield() sleep() join())

Wait() method throws Interrupted exception which is checked exception.



**class** ThreadA {

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

ThreadB b = **new** ThreadB();

b.start();

**synchronized** (b) {

System.***out***.println("Main thread trying to call wait() ");

**try** {

// main thread is calling b.wait and main is going to waiting state until child thread will notify

// main thread is expecting updates from child thread.

b.wait();

System.***out***.println("Main thread got notification ");

System.***out***.println(b.total);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

**class** ThreadB **extends** Thread {

**int** total = 0;

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

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

System.***out***.println("Child Thread starts calculation !!");

**for** (**int** i = 0; i <= 100; i++) {

total = total + 1;

}

System.***out***.println("Child Thread trying to give notification !!");

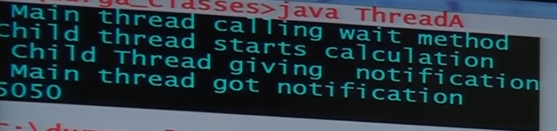
**this**.notify();

}

}

}

// What happens if the child thread has got chance first? Then main Thread will always be in the waiting state, as child got completed notifying before main thread came to waiting state. So in that case we can go with limited waiting time like wait(10000) wait for 10 sec to get notification else if still not got notified start main thread work.



Producer consumer problem…. (<https://medium.com/javarevisited/producer-consumer-problem-in-java-multi-threading-7648b8cc206d>)

DeadLock

If two threads are waiting for each other for ever such type of infinite waiting is called DeadLock.

Synchronized keyword is the only reason for deadlock situation hench while using synchronized keyword we have to take special care.

There are, no resolution technique for Deadlock but several prevention technique are available.

|  |
| --- |
| // Java program to illustrate Deadlock  // in multithreading.  **class** Util  {      // Util class to sleep a thread  **static** **void** sleep(**long** millis)      {  **try**          {              Thread.sleep(millis);          }  **catch** (InterruptedException e)          {              e.printStackTrace();          }      }  }    // This class is shared by both threads  **class** Shared  {      // first synchronized method  **synchronized** **void** test1(Shared s2)      {          System.out.println("test1-begin");          Util.sleep(1000);            // taking object lock of s2 enters          // into test2 method          s2.test2();          System.out.println("test1-end");      }        // second synchronized method  **synchronized** **void** test2()      {          System.out.println("test2-begin");          Util.sleep(1000);          // taking object lock of s1 enters          // into test1 method          System.out.println("test2-end");      }  }      **class** Thread1 **extends** Thread  {  **private** Shared s1;  **private** Shared s2;        // constructor to initialize fields  **public** Thread1(Shared s1, Shared s2)      {  **this**.s1 = s1;  **this**.s2 = s2;      }        // run method to start a thread      @Override  **public** **void** run()      {          // taking object lock of s1 enters          // into test1 method          s1.test1(s2);      }  }      **class** Thread2 **extends** Thread  {  **private** Shared s1;  **private** Shared s2;        // constructor to initialize fields  **public** Thread2(Shared s1, Shared s2)      {  **this**.s1 = s1;  **this**.s2 = s2;      }        // run method to start a thread      @Override  **public** **void** run()      {          // taking object lock of s2          // enters into test2 method          s2.test1(s1);      }  }      **public** **class** Deadlock  {  **public** **static** **void** main(String[] args)      {          // creating one object          Shared s1 = **new** Shared();            // creating second object          Shared s2 = **new** Shared();            // creating first thread and starting it          Thread1 t1 = **new** Thread1(s1, s2);          t1.start();            // creating second thread and starting it          Thread2 t2 = **new** Thread2(s1, s2);          t2.start();            // sleeping main thread          Util.sleep(2000);      }  } |

Output : test1-begin

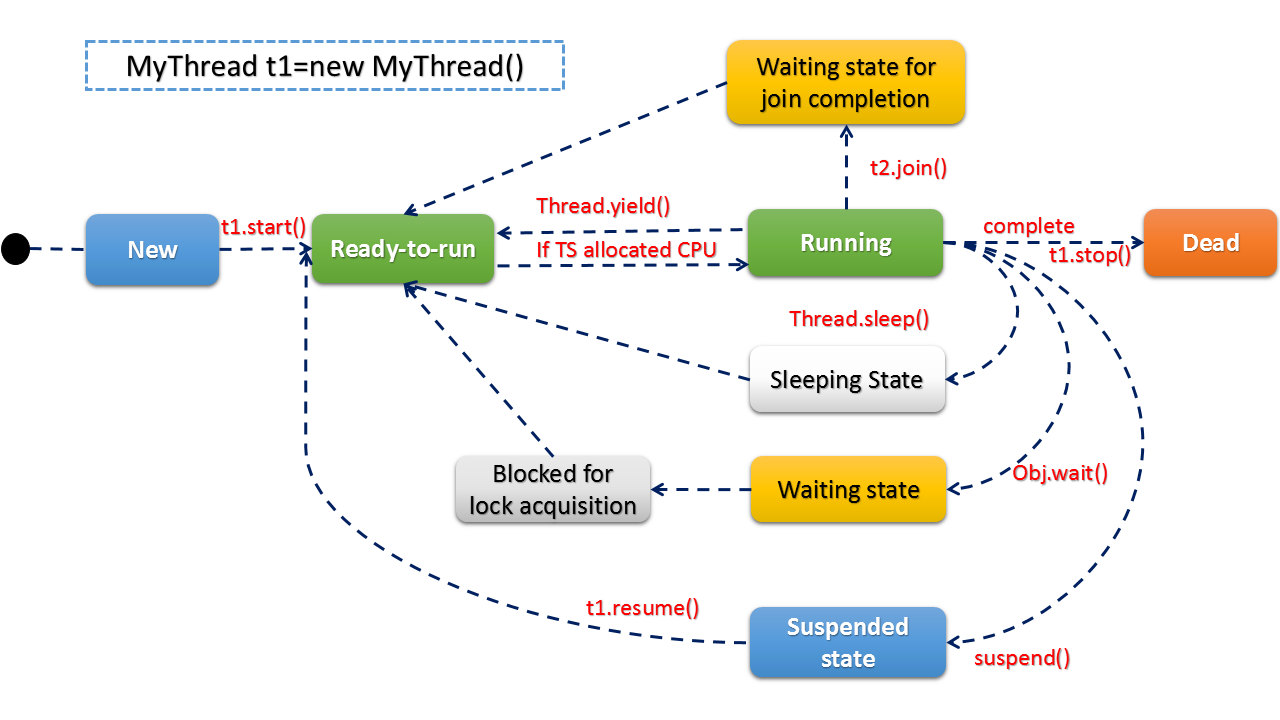
test2-begin

…………………………………(Both the threads are waiting for infinite time ……)

1. Thread t1 starts and calls test1 method by taking the object lock of s1.
2. Thread t2 starts and calls test1 method by taking the object lock of s2.
3. t1 prints test1-begin and t2 prints test-2 begin and both waits for 1 second, so that both threads can be started if any of them is not.
4. t1 tries to take object lock of s2 and call method test2 but as it is already acquired by t2 so it waits till it become free. It will not release lock of s1 until it gets lock of s2.
5. Same happens with t2. It tries to take object lock of s1 and call method test1 but it is already acquired by t1, so it has to wait till t1 release the lock. t2 will also not release lock of s2 until it gets lock of s1.
6. Now, both threads are in wait state, waiting for each other to release locks. Now there is a race around condition that who will release the lock first.
7. As none of them is ready to release lock, so this is the Dead Lock condition.
8. When you will run this program, it will be look like execution is paused.

 A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated DEAD

**Life Cycle of a thread**

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn’t started to execute.
2. **Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.   
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
3. **Blocked/Waiting state:** When a thread is temporarily inactive, then it’s in one of the following states:
   1. Blocked
   2. Waiting
4. **Timed Waiting:** A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
5. **Terminated State:** A thread terminates because of either of the following reasons:
   1. Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
   2. Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception

**NOTE: Thread.stop(), Thread.suspend(), and Thread.resume() Methods are Deprecated After JDK 1.1.**

**Thread.stop()** is being phased out due to its inherent risk. When you stop a thread, it unlocks all the monitors it has locked. Other threads might see these objects in an inconsistent state if any of the objects previously protected by these monitors were in an inconsistent state.  
Threads acting on damaged objects can act erratically, whether consciously or unconsciously. ThreadDeath, unlike other uncontrolled exceptions, silently kills threads, leaving the user with no warning that the program may be corrupted. After the damage has occurred, the corruption may appear at an unpredicted moment. Also, killing a thread will create a problem while working with DBMS – JDBC in a multithreaded environment.

**Thread.suspend()** is deprecated because it is inherently deadlock-prone. As a result, **Thread.resume()** must be deprecated as well. When the target thread is suspended, it holds a lock on the monitor protecting a crucial system resource, and no other thread may access it until the target thread is resumed. Deadlock occurs if the thread that would restart the target thread tries to lock this monitor before invoking resume().

Demon Threads

Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection. Daemon thread in Java is also a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.

In simple words, we can say that it provides services to user threads for background supporting tasks. It has no role in life other than to serve user threads.

**Example of Daemon Thread in Java:**Garbage collection in Java (gc), finalizer, etc.

**Properties of Java Daemon Thread**

* They cannot prevent the JVM from exiting when all the user threads finish their execution.
* JVM terminates itself when all user threads finish their execution.
* If JVM finds a running daemon thread, it terminates the thread and, after that, shutdown it. JVM does not care whether the Daemon thread is running or not.
* It is an utmost low priority thread.

By default, the main thread is always non-daemon but for all the remaining threads, daemon nature will be inherited from parent to child. That is, if the parent is Daemon, the child is also a Daemon and if the parent is a non-daemon, then the child is also a non-daemon.

***Note:****Whenever the last non-daemon thread terminates, all the daemon threads will be terminated automatically.*