**MultiThreading**

Syllabus:

1. Introduction
2. Ways to define the thread
   1. By extending thread Class
   2. By implementing Runnable(I)
3. Getting and Setting name of Thread
4. Thread Priorities
5. The methods to prevent thread execution
   1. Yield()
   2. Join()
   3. Sleep()
6. Synchronization
7. Inter Thread Communication
8. Deadlock
9. Daemon threads
10. Multithreading Enhancements

**INTRODUCTION**

Multitasking: Executing the several task simultaneously is known as multitasking. Two types of multitasking are there:

1. **Process based multitasking**: Executing several task simultaneous where each task is independent of each other executions and are separate independent process. Such type of multitasking is called **process based**. For example typing the program and also playing music form the same system and at the same time we want to download the file form net. All these activities will be performed simultaneously. Each task will not be having any dependency on each other and will executing independently. These tasks can be performed at OS level. If we want the multitasking at program level then we have to follow the Thread based multitasking.
2. **Thread based multitasking**: Suppose we have a program of 10k line of code and it takes the execution of program line by line 10 hours. If our first 5k line of code and 2nd 5k line of code is independent of each other’s functionality. So why we have to wait for execution of 2nd part until the first part execution is completed. In such case we can run both part of code simultaneously which will be executed in 5 hours which is haft of the previous time. Such type of program in which each thread runs independently is called **thread based multiprogramming**. Executing several tasks of simultaneously where each part is separate independent part of same program then it is called a Thread based multitasking. Each task is called thread.

Multitasking is done to reduce the processor idle time i.e. increase the utilization of process and to increase the performance of the system.

1. To develop webservers and application servers.

**Thread:** are lightweight process which are independent execution of process. Can also be called separate flow of execution. Every thread separate independent job is there. Within very less time overall job is completed. Different type we can declare a thread

1. By extending Thread Class

Class myThread extends Thread{

@override

public void run(){

for (int I, i<10; i++){

SOPLN(“Child thread”); //Job of Thread

}

}

}

Class ThreadDemo{

Public static void main(String args[]){

Mythread t= new MyThread(); //Main thread creates a child Thread Object

t.start(); //Starting of a child thread by Main thread

for (int I, i<10; i++){

SOPLN(“Child thread”); //Job of Main Thread

}

}

}

Here main thread and child thread will executed simultaneously and we’re going to get mixed output.

Main Thread

Child Thread

Thread Scheduler: part of JVM and responsible to schedule threads that is if we multiple threads are waiting to get the chance of execution then in which order threads will be executed is decided by thread scheduler. Is there any algorithm in which order it will execute the thread? We can’t expect exact algorithm followed by thread scheduler. It is varied from JVM to JVM hence we can’t expect thread execution order and exact output.

Hence whenever a situation comes to multithreading there is no guarantee for exact output but we can provide several possible outputs.

If we want start a thread we are calling t.start() method but there is no method in Mythread class.

Difference between **t.start() and t.run()**

In case of t.start() a **new thread will be created that is responsible for executing the run() method** of our thread class but if we take **t.run() then our method will run just like a normal method call by the main thread only** and other thread will be not be created. We can figure out exact output in this case.

Why we have to call t.start() method?  
start mthod of Thread class is a big utility to the programmer which contains 70k line of code. Without executing start() method there is no way a thread can be created. SO start() method is heart method

strart(){

1. Register this thread with Thread Scheduler
2. Perform all the mandatory activities.
3. Invoke the run() method;

}

Thread class start method is responsible to register the thread with thread Scheduler and all other mandatory activities. Hence without executing thread class start() method there is no chance of starting a new thread in Java. Due to this thread class start() method is considered is heart of multithreading.

**Overloading of run()** method is possible?

Class myThread extends Thread{

Public void run(){  
 SOpln(“no-arg run”)

}

Public void run(int i){  
 SOpln(“int-arg run”)

}

}

Out t.start() method of thread class will call the no argument run method only. Other overloaded method we have to call explicitly like a normal method call. So overloading of run() method is possible.

**package** test;

**import** thread.Mythread;

**public** **class** ThreadDemo {

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

Mythread t= **new** Mythread();

t.start();

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

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

}

}

}

**package** thread;

**public** **class** Mythread **extends** Thread {

@Override

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

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

System.***out***.println("no-arg run");

}

}

**public** **void** run(**int** i) {

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

System.***out***.println("int-arg run");

}

}

}

What if donot write any implementation of run() method in thread class?

A thread will be crated and run() method will be invoked which will have empty implementation and no output will be printed when we start this thread.

If we are not overriding run method then thread class run method will be executed which has empty implementation hence we wont get any output.

What if **start() method of Thread class is overridden**?

**A new thread will be created** and **start() method in Thread class will be called just like a normal method**.

**package** test;

**import** thread.Mythread;

**public** **class** ThreadDemo {

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

Mythread t= **new** Mythread();

t.start();

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

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

}

}

}

**package** thread;

**public** **class** Mythread **extends** Thread {

@Override

**public** **synchronized** **void** start() {

System.***out***.println("start method");

}

@Override

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

System.***out***.println("run method");

}

}

Note: It is not recommended to override start() method and recommended to override run() method otherwise don’t go for multithreading concept.

LifeCycle of Thread:

MyThread t = new MyThread()

New/born t.start Ready If Thread Schedular Running

/Runnable Assigns Processor

If run() method completes

Dead

If a thread is already started and if we are restarting that thread again then we’ll get IllegalThreadStateException at RunTime

Thread t = new Thread();

t.start();

---------

---------

t.start(); //RuntimeException: IllegalThreadStateException

1. **By implementing Runnable Interface**

Thread Class already implements Runnable Interface. So instead of extending the Thread class we can directly implement the Runnable Interface.

Runnable interface is present in java.lang package and it contains one method that is run() method

public void run();

Class MyRunnable implements Runnable{

Public void run(){

for(int i=0; i<n; i++){

SOPln(“Child Thread”);

}

}

}

Class ThreadDemo{

Public static void main(String[] args){

MyRunnable r= new MyRunnable(); //It doesn’t contain the start capability

Thread t= new Thread(r); //pass the runnable implementing class reference in thread

t.start();

for(int i=0; i<n; i++){

SOPln(“Main Thread”);

}

}

}

Case Study:

MyRunnable r= new MyRunnable();

Thread t= new Thread();

Thread t1= new Thread(r);

Case 1: t.start(); will start the thread without any runnable target

Case 2: t.run(); will invoke run() like normal method call and will not print anything

Case 3: t1.start(); will start the thread with runnable target r

Case 4: t1.run(); MyRunnable run method will be invoked as normal method call

Case 5: r.start(); Will give Runtime exceptions

Case 6: r.run() Will invoke run() method of MyRunnable class as a normal method call

Which method of defining the Thread is recommended?

2nd one is recommended i.e. by implementing the Runnable Interface. In first approach we need to extend our thread class using Thread Class and hence we can’t extend any other class as multiple inheritance is not allowed but in the second class which implements our Runnable class we can also extend this class with some other class and hence we won’t miss any inheritance capability.

Class MyThread extends Thread{

Public void run(){

}

}

Class MyRunnable extends Test implements Runnable{

Public void run(){

}

}

Thread Class Constructors

1. Thread t = new Thread();
2. Thread t = new Thread(Runnable r);
3. Thread t = new Thread(String name); //Name of thread as name
4. Thread t= new Thread(Runnable r, String name)
5. Thread t= new Thread(ThreadGroup g, String name)
6. Thread t= new Thread(ThreadGroup g, Runnable r)
7. Thread t= new Thread(ThreadGroup g, Runnable r, String name);
8. Thread t= new Thread(ThreadGroup g, Ruinnable r, String name, long stacksize);

Hybrid Thread: Following Thread is a valid Thread Implemenation

**package** thread;

**public** **class** Mythread **extends** Thread {

@Override

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

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

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

}

}

}

**package** test;

**import** thread.Mythread;

**public** **class** ThreadDemo {

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

Mythread t = **new** Mythread();

Thread t1= **new** Thread(t); //r= runnable Target

t1.start();

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

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

}

}

}

Every thread in Java has some name. It may be generated by JVM or it can be supplied by Programmer. Class Thread contains one static method called currentThread which gets the current thread object reference

Sample Example:  
**package** test;

**import** thread.Mythread;

**public** **class** ThreadDemo {

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

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

Mythread t = **new** Mythread();

Thread t1= **new** Thread(t); //r= runnable Target

System.***out***.println(t1.getName());

Thread.*currentThread*().setName("AK");

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

t1.setName("SRK");

System.***out***.println(t1.getName());

t1.start();

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

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

}

System.***out***.println(10/0);

}

}

**package** thread;

**public** **class** Mythread **extends** Thread {

@Override

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

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

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

}

System.***out***.println(10/0);

}

}

Output of the program is:

main

Thread-1

AK

SRK

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Exception in thread "AK" java.lang.ArithmeticException: / by zero

at test.ThreadDemo.main(ThreadDemo.java:31)

Child Thread

Child Thread

Child Thread

Child Thread

Child Thread

Child Thread

Exception in thread "SRK" java.lang.ArithmeticException: / by zero

at thread.Mythread.run(Mythread.java:12)

at java.lang.Thread.run(Unknown Source)

Child Thread

Child Thread

Child Thread

Child Thread

**Thread Priorities**

Every thread in java has some priority which can be given by default by Java or User can explicitly provide the same to the Thread.

What is valid range of **thread priorities**? It is **always (1 to 10**).

**1 is min priority** and **10 is max priority**

**Thread Class represents to define the Constants as follows:**

Thread.MIN\_PRIORITY value is 1

Thread.NORM\_PRIORITY value is 5

Thread.MAX\_PRIORITY value is 10

Every threads in Java has some priority it may be default priority generated by JVM or customized priority provided by programmer.

Thread Scheduler will use priorities while allocating processor. The thread which is having highest priority will get the chance first. If two threads have same priority then which thread will get the chance to execute first then we can’t expect which thread will get the chance to execute first. It is totally **dependent on** **thread scheduler**.

Thread class contains following method to set and get the priority of a Thread

**Public final int getPriority()**

**Public final void setPriority(int p)**

If we don’t provide the priority in range 1-10 then following RuntimeException will be raised IllegalArgumentException.

Exception in thread "main" java.lang.IllegalArgumentException

at java.lang.Thread.setPriority(Unknown Source)

at test.ThreadDemo.main(ThreadDemo.java:13)

Default priority for main() class is 5 and other thread will inherit the same priority of the parent class. If parent Thread Priority is 10 then child Thread will have the same priority.

Consider following example

**package** thread;

**public** **class** Mythread **extends** Thread {

@Override

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

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

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

}

}

}

**package** test;

**import** thread.Mythread;

**public** **class** ThreadDemo {

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

System.***out***.println("Default main Thread priority " + Thread.*currentThread*().getPriority());

Mythread t = **new** Mythread();

t.setPriority(10); // line 1

System.***out***.println("Child Thread priority " + t.getPriority());

t.start();

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

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

}

}

}

If we are commenting line 1 in above example then we don’t know in which order threads will be executed as both the threads will be having the same priority ie 5. Also we can’t expect the exact output

But if we uncomment the line 1 then we can expect that Child Thread gets executed before the main thread gets executed. Child thread have the priority iof 10 while Main thread has the priority of 5 hence the child thread will get the chance to to execute followed by main thread. Output of above example if 1 line is commented

Default main Thread priority 5

Child Thread priority 10

Child Thread

Child Thread

Child Thread

Child Thread

Child Thread

Child Thread

Child Thread

Child Thread

Child Thread

Child Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Some OS or platforms won’t provide proper support for thread priorities. In that case we may get desired output.

**Preventing a Thread Executing**

Different ways we can prevent or suspend a thread execution for some time

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

**Yield() Method**

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

If multiple thread are waiting with same priority then which waiting thread will get the chance?

**We can’t expect. It depends on Thread Scheduler**

The Thread which is yielded will be pushed into the thread list of having same priority containing various threads, when it will get the chance once again?

**It depends on thread scheduler and we can’t expect exactly**.

Complete prototype of yield Method

public static method native void yield();

Thread.yield()

New/born t.start Ready If Thread Schedular Running

/Runnable Assigns Processor

If run() method completes

Dead

**package** thread;

**public** **class** Mythread **extends** Thread {

@Override

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

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

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

Thread.*yield*();

}

}

}

**package** test;

**import** thread.Mythread;

**public** **class** ThreadDemo {

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

Mythread t= **new** Mythread();

t.start();

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

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

}

}

}

Here Main thread will get the more number of chance and the chance of completing main thread is high. Yield is called on thread which requires more time on processor. Some platforms won’t provide proper support for yield method.

**Join()**

If a thread wants to wait until completion of execution of other thread on which output the thread is dependent on then we should go for join method to be called on other t2 object inside t1,

For example if a thread t1 wants to wait until completing t2 then t1 has to call t2.join () method. If t1 executes the line t2.join then immediately t1 thread will enter into waiting state until thread t2 has completed its execution. Once t2 completes then t1 can continue its execution.

Consider following example.

Venue fixing Activity Wedding card printing Wedding card Dsitribution

t1 t2 t3

t1.join t2.join

Here before wedding card printing is started the venue for marriage has be fixed hence t2 thread resembling weeding card printing activity has to be suspended till venue is fixed i.e. t1 is completed. Likewise before distributing the wedding invitation card representing t3 the wedding card printing activity has to be completed hence the t2 thread has be suspended by thread t3.

public final void join() throws IterruptedException

public final void join(Long ms) throws IterruptedException : If a circumstances like waiting for some time in ms

public final void join(Long ms, int ns) throws IterruptedException.

Every join method throws InterruptedException which is a Checked Exceptions hence compulsory we should handle these Exceptions either by using try catch block or by throws Keyword otherwise we will get compile time error.

If t2 completes or

If time expires or

If waiting thread got interrupted t2.join()

t2.join(1000)

t2.join(1000, 100)

Thread.yield()

New/born t.start Ready If Thread Schedular Running

/Runnable Assigns Processor

If run() method completes

Dead

What is impact of join method on our thread

**package** thread;

**public** **class** Mythread **extends** Thread {

@Override

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

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

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

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**package** test;

**import** thread.Mythread;

**public** **class** ThreadDemo {

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

Mythread t= **new** Mythread();

t.start();

**try** {

t.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

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

}

}

}

Case 2: Waiting of child thread until completing the main thread

**package** thread;

**public** **class** Mythread **extends** Thread {

**public** **static** Thread *mt*;

@Override

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

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

**try** {

*mt*.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

}

}

}

**package** test;

**import** thread.Mythread;

**public** **class** ThreadDemo {

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

Mythread.*mt*= Thread.*currentThread*();

Mythread t= **new** Mythread();

t.start();

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

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

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

Case: 3

**package** thread;

**public** **class** Mythread **extends** Thread {

**public** **static** Thread *mt*;

@Override

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

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

**try** {

*mt*.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

}

}

}

**package** test;

**import** thread.Mythread;

**public** **class** ThreadDemo {

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

Mythread.*mt*= Thread.*currentThread*();

Mythread t= **new** Mythread();

t.start();

**try** {

t.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

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

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

In above case deadlock situation will occur as both the thread child and main thread will wait for one another to execute and hence program will be in deadlock state

Case4: If a thread calls a join method on the same thread itself then the program will be strucked. This is something like deadlock. In this case threads has to wait infinite amount of time. For example following program will enter into deadlock.

Public class Test{

Public static void main(String args[]) throws InterruptedException{

Thread.currentThread().join();

}

}

**Sleep()**

If a thread don’t want to perform any operations for a particular amount of time then we should go for sleep method.

Complete signature of sleep method

public static native void sleep(Long ms) throws InterruptedException

public static void sleep(Long ms, int ns) throws InterruptedException

Every Sleep method throws InterruptedException which is a checked exception, hence whenever we are using sleep method compulsorily we should handle InterruptedException either by Try catch block or by Throws keyword

If time expires or

If waiting thread got interrupted

Thread.sleep(1000)

Thread.join(1000, 100)

Thread.yield()

New/born t.start Ready If Thread Schedular Running

/Runnable Assigns Processor

If run() method completes

Dead

**package** test;

**public** **class** SlideRotator {

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

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

System.***out***.println("Slide " + (i+1));

Thread.*sleep*(1000);

}

}

}

How a Thread can interrupt another Thread?

A thread waiting and sleeping can be interrupted using interrupt method of the interrupting thread.

Public void interrupt();

**package** thread;

**public** **class** Mythread **extends** Thread {

@Override

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

**try**{

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

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

Thread.*sleep*(2000);

}

}**catch**(InterruptedException e){

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

}

}

}

**package** test;

**import** thread.Mythread;

**public** **class** ThreadDemo {

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

Mythread t= **new** Mythread();

t.start();

t.interrupt();

System.***out***.println("End of main Thread");

}

}

If the target thread is not in sleeping and waiting state and then there is no impact of interrupt method. Interrupt call be waited until target thread is entered into sleeping or waiting state. If the target state entered into sleeping and waiting state then immediately interrupt call will interrupt the target thread.

If the target thread never into sleeping and waiting state in its lifetime then there is no impact of interrupt call. This is the only case where interrupt call will be wasted.

Comparison table of yield join and sleep method

Property yield Join Sleep

Purpose If a thread wants to pass if a thread wants to wait If a thread don’t want to perform

its execution to give the until completing other any operation for a particular

chance for the reaming thread the it is required amount of time then its required

thread Of same priority

then its required

Overloaded? No yes yes

Final? No yes No

Throws IE? No Yes Yes

Native? Yes No Both

**Synchronization**

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

If multiple threads are tend 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 the synchronized keyword is it increases waiting time of threads and creates performance problems hence if there is no specific requirements then it is not recommended to use synchronized keyword.

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

If a thread wants to execute synchronized method on the given object first 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 of lock internally takes care by JVM and the programmer not responsible for these activities.

Class X{

Synch m1(){}

Synch m2(){}

M3(){}

}

T2 wants to execute m1()

But is not allowed

T1 wants to execute m1()

T3 wants to execute t4 wants to execute m3() and it

M2() but will be not allowed will be allowed to execute

Even m2 method is not executed by any thread prior to t3 thread it’s not allowed to execute m2 method because every synchronized method has to have locks released then only other threads can execute the synchronized methods.

While a thread wants to execute the non-synchronized method then thread is allowed to execute as this method don’t want the locks to be acquired by it to execute.

This area can be accessed be accessed by This area can be accessed by only one thread at a time

Multiple thread simultaneously

Non-Synchronized Synchronized

Area area

Class X{

Synchronized{

Wherever we are performing update operations like add, delete, remove, replace where state of Object changes

}

Non-synchronized{

Wherever object state won’t be changed like read operations

}

}

Class ReservationSystem{

Non-synchronized checkavailability{

Read Operations

}

Synchronized bookTicktes(){

Update Operations

}

}

Example

**package** thread;

**public** **class** Display {

**public** **synchronized** **void** wish(String name) {

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

System.***out***.print("Good Morning: ");

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

}

}

}

**package** thread;

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

**private** Display d;

**private** String name;

**public** MyThread(Display d, String name) {

**this**.d = d;

**this**.name = name;

}

@Override

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

d.wish(name);

}

}

**package** test;

**import** thread.Display;

**import** thread.MyThread;

**public** **class** ThreadDemo {

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

Display d= **new** Display();

MyThread t1= **new** MyThread(d, "Dhoni");

MyThread t2 = **new** MyThread(d, "Yuvraj");

MyThread t3 = **new** MyThread(d, "Kohli");

MyThread t4 = **new** MyThread(d, "Raina");

t1.start();

t2.start();

t3.start();

t4.start();

}

}

If we are not declaring wish method as Synchronized then all the threads will be executed simultaneously and hence we will get irregular output.

Output will be like as follows:

Good Morning: Good Morning: Good Morning: Good Morning: Dhoni

Good Morning: Kohli

Good Morning: Raina

Good Morning: Yuvraj

If we declare wish method as synchronized then at a time only one thread is allowed to execute wish method on the given Display object hence we will get regular output.

Output would be like as follows:

Good Morning: Dhoni

Good Morning: Dhoni

Good Morning: Dhoni

Good Morning: Dhoni

Good Morning: Dhoni

If we change main class as follows

**package** test;

**import** thread.Display;

**import** thread.MyThread;

**public** **class** ThreadDemo {

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

Display d= **new** Display();

Display d2= **new** Display();

MyThread t1= **new** MyThread(d, "Dhoni");

MyThread t2 = **new** MyThread(d2, "Yuvraj");

t1.start();

t2.start();

}

}

Even though wish method is synchronized we will get irregular output because threads are operating on different java objects.

If multiple threads are operating on same java objects then synchronization is required. If multiple threads are operating on multiple java objects then synchronization is not required.

Output of above program is as follows

Consider following example

**package** thread;

**public** **class** Display {

**public** **static** **synchronized** **void** wish(String name) {

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

System.***out***.print("Good Morning: ");

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

}

}

}

**package** thread;

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

**private** Display d;

**private** String name;

**public** MyThread(Display d, String name) {

**this**.d = d;

**this**.name = name;

}

@Override

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

d.*wish*(name);

}

}

**package** test;

**import** thread.Display;

**import** thread.MyThread;

**public** **class** ThreadDemo {

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

Display d= **new** Display();

Display d2= **new** Display();

MyThread t1= **new** MyThread(d, "Dhoni");

MyThread t2 = **new** MyThread(d2, "Yuvraj");

t1.start();

t2.start();

}

}

Every class in Java has a unique lock like every object has unique lock. When a method is declared with static synchronized then a class level is acquired by wish Method. So in above example a thread t1 comes to execute the wish method of Display class then it will acquire the class level lock and will execute the wish() while the thread t2 will wait for the locks to be released and begin its execution.

In Java 2 level lock is there

1. **Object level Lock** – Every object in Java has a unique lock.
2. **Class Level Lock**- Every class in java has a unique lock which is nothing but 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 then it is allowed to execute any static synchronized method of that class. Once method execution completes automatically thread releases the lock.

Consider Following class example

Class X{

Static synch m1()

Static synch m2()

Static m3()

Synch m4()

m5()

}

T1 wants to execute m1() need CL(X)

T2 wants execute m1() not allowed

And goes to waiting state

T3 wants to execute m2() not

allowed and goes waiting state T6 wants to execute m5() which is normal

method and allowed to execute

T4 wants to execute m3() and

Will allowed to execute

T5 wants to execute m4() and is allowed

To execute by acquiring Object level Lock(X)

The above example show that while a thread executing static synchronized method the reaming thread are not allowed to execute any static synchronized method of that class simultaneously but remaining threads are allowed to execute following methods simultaneously:

1. Normal static methods
2. Synchronized Instance methods
3. Normal instance methods

**Synchronized Block**

Consider following method

M1(){

Contains 10k only 10 lines required synchronizarion

Line of code

}

If very few lines of the code requires the synchronization then it’s not recommended to declare entire method as synchronized. We have to enclose those few lines of the code by using synchronized block.

Then main advantage of synchronized block over synchronized block is it reduces waiting time of threads and improves performance of the system.

**We can declare synchronized block as follows:**

1. To get the lock of current object

**Synchronized (this)**{

If a thread got the lock of current object then only it is allowed to execute this area

}

1. To get lock of particular object ‘b’

**Synchronized (b)**{

If a thread got lock of particular object b then only then it is allowed to execute this area

}

1. To get class level lock

**Synchronized (Display.class)**{

If a thread got class level lock of Display class then only it is allowed to execute this area.

}

Example:

**package** thread;

**public** **class** Display {

**public** **void** wish(String name) {

//;;;;;;; 1 lakh of lines

System.***out***.println("Before synchronized Block");

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

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

System.***out***.print("Good Moring: ");

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

}

}

System.***out***.println("After synchronized Block");

//;;;;;; 1 lakh of lines

}

}

**package** thread;

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

**private** Display d;

**private** String name;

**public** MyRunnable(Display d, String name) {

**super**();

**this**.d = d;

**this**.name= name;

}

@Override

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

d.wish(name);

}

}

**package** test;

**import** thread.Display;

**import** thread.MyRunnable;

**public** **class** SynchrinizedBlockDemo {

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

Display d = **new** Display();

MyRunnable r1= **new** MyRunnable(d, "Yuvraj");

MyRunnable r2= **new** MyRunnable(d, "Dhoni");

Thread t1= **new** Thread(r1);

Thread t2= **new** Thread(r2);

t1.start();

t2.start();

}

}

Output:

Before synchronized Block

Before synchronized Block

Good Moring: Dhoni

Good Moring: Dhoni

Good Moring: Dhoni

Good Moring: Dhoni

Good Moring: Dhoni

Good Moring: Dhoni

Good Moring: Dhoni

Good Moring: Dhoni

Good Moring: Dhoni

Good Moring: Dhoni

Good Moring: After synchronized Block

Yuvraj

Good Moring: Yuvraj

Good Moring: Yuvraj

Good Moring: Yuvraj

Good Moring: Yuvraj

Good Moring: Yuvraj

Good Moring: Yuvraj

Good Moring: Yuvraj

Good Moring: Yuvraj

Good Moring: Yuvraj

After synchronized Block

Class Level Lock Example

**package** test;

**import** thread.Display;

**import** thread.MyRunnable;

**public** **class** SynchrinizedBlockDemo {

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

Display d1 = **new** Display();

Display d2 = **new** Display();

MyRunnable r1= **new** MyRunnable(d1, "Yuvraj");

MyRunnable r2= **new** MyRunnable(d2, "Dhoni");

Thread t1= **new** Thread(r1);

Thread t2= **new** Thread(r2);

t1.start();

t2.start();

}

}

**package** thread;

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

**private** Display d;

**private** String name;

**public** MyRunnable(Display d, String name) {

**super**();

**this**.d = d;

**this**.name= name;

}

@Override

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

d.wish(name);

}

}

**package** thread;

**public** **class** Display {

**public** **void** wish(String name) {

//;;;;;;; 1 lakh of lines

System.***out***.println("Before synchronized Block");

**synchronized** (Display.**class**) {

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

System.***out***.print("Good Moring: ");

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

}

}

System.***out***.println("After synchronized Block");

//;;;;;; 1 lakh of lines

}

}

Note: Lock concept applicable for object types and class types but not for primitives hence we can’t pass primitive type argument to synchronized block otherwise we will get compiletime error saying unexpected type found: int required: reference

FAQs:

1. What is synchronized keyword? Where we can apply?
2. Explain advantage of synchronized keyword.
3. Explain Disadvantages of synchronized keyword.
4. What is race condition? Ans: Data inconsistency problem when multiple thread operating simultaneously on the same java object is called **race condition.**
5. What is object lock and when it is required?
6. What is Class Level Lock and when it is required?
7. What is difference between Object Level Lock and Class Level Lock?
8. What is synchronized block?
9. How to declare synchronized block to get the lock of current object?
10. How to declare synchronized block to get Class Level Lock?
11. What is advantage of synchronized block of over synchronized method?
12. Is a thread can acquire multiple locks simultaneously? Ans: Yes, From different objects

Class X{

Public synchronized void m1(){ //Here Thread has lock of X object

Y y = new Y();

Synchronized(y){ //Here Thread has Locks of X and Y object

Z z= new Z();

Synchronized(Z){ //Here Thread has locks of X, Y and Z

-----

-----

}

}

}

}

**InterThread Communication**

Two Threads can communicate with each other by using wait(), notify(), notifyAll() methods. The thread which expecting updation is responsible to call wait() method then immediately the thread will enter into waiting state.

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

Wait(), notify(), notifyAll() present in class in Object class. **Why it’s not in Thread Class**? **A thread can call wait(), notify() and notifyAll() method on any java Object that’s why wait() and other methods are defined in Object Class.**

To call wait(), notify() or notifyAll() methods on any object, Thread should be owner of that object ie the thread should have lock of that object. Thread should be inside should inside synchronized area. Hence wait(), notify() and notifyAll() methods only form synchronized are otherwise we will get RunTime Exception saying : **IllegalMonitorStateException**

If a Thread calls wait () method on nay object it immediately releases the lock of that particular object and enters into waiting state.

If a Thread calls a notify() method on any object it releases the lock of that object but may not immediately. Except wait(), notify() and notifyAll() there is no other method where threads releases the lock.

|  |  |
| --- | --- |
| Method | Is Threas releases the lock |
| Yield() | No |
| Join() | No |
| Sleep() | No |
| Wait() | Yes |
| Notify() | Yes |
| notifyAll() | Yes |

Which of the following is valid?

1. If a threads calls wait() method immediately it will enter into waiting state without releasing any lock. Invalid
2. If a Thread calls wait() method it releases the lock of that object but may not immediately. Invalid
3. If a thread calls wait() method on any object it releases all locks acquired by that thread and immediately entered into waiting state. invalid
4. If a thread calls wait() method on any object it immediately releases the lock of that particular and entered into waiting state. Valid
5. If a thread calls notify() method on any object it immediately releases the lock of that particular object. Invalid
6. If a thread calls notify() method on any object it releases the lock of that object but may not immediately. Valid

Complete Signature of wait(), notify(), notifyAll() methods

Public final void wait() throws InterrruptedException

Public final native void wait(Long ms) throws InterrruptedException

Public final void wait(Long ms, int ns) throws InterrruptedException

Public final native void notify()

Public final native void notifyAll()

Note: Every wait() method throws interrupted Exception which is checked Exception hence whenever we are using wait method compulsorily we should handle this interruptedException either by try catch block or by Throws keyword otherwise we will get CompileTime Error.

If waiting thread giot notify()

If time expires or

If waiting thread got interrupted obj.wait()

If waiting thread got lock obj.wait(1000)

obj.wait (1000, 100)

Thread.yield()

New/born t.start Ready If Thread Schedular Running

/Runnable Assigns Processor

If run() method completes

Dead

Consider Following Example

**package** test;

**import** thread.MyRunnable;

**public** **class** InterThreadComDemo {

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

MyRunnable r= **new** MyRunnable();

Thread t= **new** Thread(r);

t.start();

**try** {

t.join();

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

System.***out***.println("Main: " + r.getTotal());

}

}

**package** thread;

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

**private** **int** total=0;

**public** **int** getTotal() {

**return** total;

}

**public** **void** setTotal(**int** total) {

**this**.total = total;

}

@Override

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

**for** (**int** i = 1; i <=100000; i++) {

total+=i;

}

//10k lines are theres

System.***out***.println("Child: " + **this**.getTotal());

}

}

Sleep() method is not all recommended if our main thread is waiting for the result produced by Child Thread because we can’t expect the exact output.

Join() method is also not recommended because after the for loop in thread class where total value is being calculated, there is 10k line of code. Executing these code is unnecessary if the value of total is calculated after the for loop only. So what is the need to waiting till this thread completes it execution. This will unnecessarily will increase the waiting time to get the result.

In this case wait() method on child thread will tell other threads to wait for the total till it has been calculated i.e. for loop code will enclosed inside synchronized block. And after the updation of total value the thread is going to call notify() method to notify other threads that value has been updated and they can access it.

Following is the exception we will get if our wait() and notify() block is not put into synchronized block

Exception in thread "main" Exception in thread "Thread-0" java.lang.IllegalMonitorStateException

at java.lang.Object.wait(Native Method)

at java.lang.Object.wait(Unknown Source)

at test.InterThreadComDemo.main(InterThreadComDemo.java:16)

java.lang.IllegalMonitorStateException

at java.lang.Object.notify(Native Method)

at thread.MyRunnable.run(MyRunnable.java:20)

at java.lang.Thread.run(Unknown Source)

**package** thread;

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

**private** **int** total=0;

**public** **int** getTotal() {

**return** total;

}

**public** **void** setTotal(**int** total) {

**this**.total = total;

}

@Override

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

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

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

**for** (**int** i = 1; i <=100000; i++) {

total+=i;

}

1. System.***out***.println("Child thread trying to give notification");

**this**.notify();

}

System.***out***.println("Child: " + **this**.getTotal());

}

}

**package** test;

**import** thread.MyRunnable;

**public** **class** InterThreadComDemo {

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

MyRunnable r= **new** MyRunnable();

Thread t= **new** Thread(r);

t.start();

**synchronized** (t) {

1 System.***out***.println("main thread trying to call wait method");

t.wait();

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

5 System.***out***.println("Main: " + r.getTotal());

}

}

}

Output:

main thread trying to call wait method

Child Thread starts calculation

Child thread trying to give notification

Child: 705082704

main thread got notification

Main: 705082704

**Producer and Consumer Algorithm**

Producer Thread{

Produce(){

Synchronize(q){

If (q is not full){

Produce items to Queue;

q.notify

}

}

}

}

Consumer Thread{

Consume(){

Synchnronize(q){

If (q is empty){

q.wait();

}

Else{

Consume items

}

}

}

}

Difference between notify() and notifyAll() methods

We can use notify() method to give the notification for only one waiting thread. If multiple threads are waiting then only one thread will be notified and the remaining threads have to wait for further notifications. Which thread will be notified we can’t expect it depends on JVM.

Total waiting threads: 100 Threads

Obj1.wait(); 60 Threads waiting for obj1

Obj2.wait(); 40 Threads waiting for obj2

We can use notifyAll() method to give the notification for all waiting threads of a particular object. Even though multiple threads notified but execution will be performed one by one because threads required lock and only one lock is available for a given object.

Consider this case:

Stack s1 = new Stack();

Stack s2= new Stack();

|  |  |
| --- | --- |
| Synchronized(s1){  ------------------------------  s2.wait()  }  We will get RE: IllegalMonitorStateException because s2 is not the valid object on which wait method can be called | Synchronized(s1){  --------------------  s1.wait();  -------------------  }  S1.wait() is valid call of wait() as current thread is owner of the s1 object so on this wait() method is valid. |

Note: On which object we are calling wait() method thread required the lock of that particular object for example if we are calling wait() method on s1 then we have to get lock of s1 object but not s2 object.

**Deadlock**

If two threads are waiting for each other forever. Such type infinite waiting is called deadlock. Synchronized keyword is the only reason for the deadlock situation hence while using synchronized keyword we have to take especial care.

There are no resolution techniques for deadlock but several prevention techniques are available.

Following is the scenario of deadlock.

|  |  |
| --- | --- |
| Class A{    synchronized(){  d1(B b){  --------  --------  b.last();  }    }  synchronized(){  last(){  -------------  -------------    }  }  } | Class B{    synchronized(){  d2(A a){  --------  --------  a.last();  }    }  synchronized(){  last(){  -------------  -------------    }  }  } |
| T1 need lock(a)  Will try to execute d1(b)  method  Here in above example Thread t1 will acquire lock on object a and will try to execute d1 method which contains the sync method last() of object b and hence will required the lock of object b to execute this method which already acquired by Thread t2 | T2 need lock(b)  Will try to execute d2(a)  method  At the same time thread t2 holds the lock on object b and will try to execute the d2 method and this method contains synch method last() of object a and in order to execute this method thread T2 needs lock of Object a which is acquired by thread t1 already. So both the threads will keep on waiting to acquire the locks on each other and will never get the chance acquire the lock as they will wait for each other to release the lock. |

Following is the implementation of above example

**package** thread;

**public** **class** A {

**public** **synchronized** **void** d1(B b) {

System.***out***.println("Thread1 starts the execution of d1 method");

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Thread1 trying to call B's last method");

b.last();

}

**public** **synchronized** **void** last() {

System.***out***.println("Inside A's Last Method");

}

}

**package** thread;

**public** **class** B {

**public** **synchronized** **void** d2(A a) {

System.***out***.println("Thread2 starts the execution of d2 method");

**try** {

Thread.*sleep*(5000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Thread2 trying to call A's last method");

a.last();

}

**public** **synchronized** **void** last() {

System.***out***.println("Inside B's last method");

}

}

**package** thread;

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

A a= **new** A();

B b = **new** B();

**public** **void** m1() {

**this**.start();

a.d1(b); //executed by Main Thread

}

@Override

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

b.d2(a); //executed by Child Thread

}

}

**package** test;

**import** thread.MyThread;

**public** **class** DeadLockDemo {

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

MyThread t= **new** MyThread();

t.m1();

}

}

Output:

Thread1 starts the execution of d1 method

Thread2 starts the execution of d2 method

Thread1 trying to call B's last method

Thread2 trying to call A's last method

In the above program if we remove at least one synchronized keyword then program would not have entered into deadlock hence synchronized keyword is only reason for deadlock situation. Due to this while using synchronized keyword we have to take special care.

Starvation and deadlock is related terminologies

**Deadlock vs Starvation: Long waiting of thread where waiting never ends such type of thing is deadlock while long waiting of thread where waiting will end at certain point is called starvation.**

For example low priority thread has to wait until completing all high priority threads. It may be long waiting but ends at a certain point, which is nothing but starvation.

**Daemon Threads**

The threads which are executing in the background are called **Daemon Threads**. Example is Garbage Collector, SignalDispatcher, AttachListener. The main objective of daemon threads is to provide support for non-daemon thread (main thread) for example main thread runs with low memory then JVM runs Garbage Collector to destroy useless objects so that number of bytes of free memory will be improved. With this free memory main thread can continue its execution. Usually daemon threads having low priority but based on our requirement daemon threads can run with high priority also.

We can check daemon nature of thread by using isDaemon() method of Thread Class

**Public Boolean isDaemon() //To know a thread is daemon**

We can change Daemon nature of Nature of thread by using setDaemon() but changing Daemon nature is possible before starting of a thread only. After starting a thread if we are trying to change daemon nature then we will get runtime Exception IllegalThreadStateException.

Public void setDaemon(Boolean b) throws IllegalThreadStateException //to set a thread daemon

**Default Nature of a Thread**

By default a main thread is always non-daemon and for all remaining daemon nature will be inherited from parent to child i.e. if the parent thread is daemon then automatically child thread is also a daemon and if the thread is non-daemon then automatically the child is also non-daemon. **There is no way main thread can be set to non-daemon thread as main thread is already started by JM as non-daemon thread.**

For example

**package** test;

**public** **class** DaemonThreadDemo {

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

System.***out***.println(Thread.*currentThread*().isDaemon());

Thread.*currentThread*().setDaemon(**true**);

}

}

Output:

false

Exception in thread "main" java.lang.IllegalThreadStateException

at java.lang.Thread.setDaemon(Unknown Source)

at test.DaemonThreadDemo.main(DaemonThreadDemo.java:9)

Changing the Daemon nature of a Thread

**package** test;

**public** **class** DaemonThreadDemo {

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

System.***out***.println(Thread.*currentThread*().isDaemon());

//Thread.currentThread().setDaemon(true);

Thread t= **new** Thread();

System.***out***.println(t.isDaemon());

// /t.start();

t.setDaemon(**true**);

System.***out***.println(t.isDaemon());

}

}

Whenever last non-daemon thread terminates automatically all daemon threads will be terminated irrespective of their position.

**package** test;

**import** thread.MyRunnable;

**public** **class** DaemonThreadDemo {

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

MyRunnable r= **new** MyRunnable();

Thread t= **new** Thread(r);

t.setDaemon(**true**);

t.start();

System.***out***.println("End of main");

}

}

**package** thread;

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

@Override

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

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

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

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

If t.setDaemon(true) is commented then both main and child thread will be executed till both of them has finished their execution. If it is not commented, child thread become daemon. Once the main thread is has finished its execution then daemon child thread will also terminated.

**What is Green Thread?**

Java multithreading concept is implemented by using following 2 models:

1. Green Thread Model
2. Native OS Model

Green Thread Model: The Thread which managed completely by JVM without taking underlying OS support is called Green thread. Very few operating System like Sun Solaris will provides support for green thread model. Any way Green thread model is deprecated and not recommended to use.

Native OS Model: The Thread which is managed by the JVM with the help of underlying OS, is called native OS Model. All windows based operating systems provide support for native OS model.

How to stop a Thread?

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

Public void stop()

If we call stop() method then immediately thread will enter into dead state. Anyway stop() method is deprecated and not recommended to use.

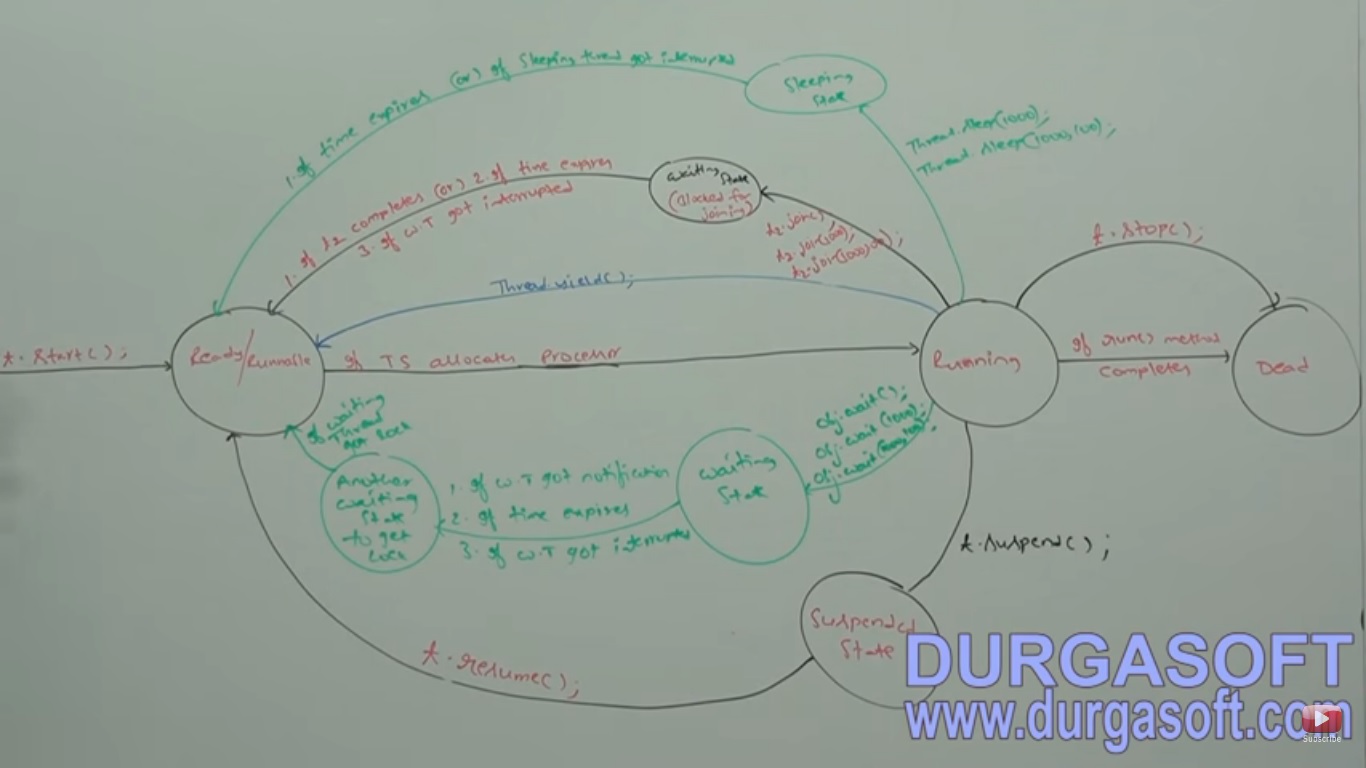
How to suspend and resume a Thread?

We can suspend a thread by using suspend() method of Thread Class them immediately the thread will be entered into suspended state. We can resume a suspended a thread by using resume() method of Thread class then suspended thread can continue its execution.

Public void suspend()

Public void resume()

Anyway these methods are deprecated and not recommended to use.



If time expires or

If waiting thread got interrupted

Thread.sleep(1000)

Thread.join(1000, 100)

Thread.yield()

New/born t.start Ready If Thread Schedular Running/

Assigns Processor /Runnable

If run() method completes

Dead