# Threading

## Multitasking

Executing multiple task simultaneously is a concept of multitasking.

There are two type of multitasking.

1. Process based multi-tasking

2. Thread based multi-tasking

**1. Process based multitasking**

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

Ex: While typing a java program in the editor we can listen audio song from the same system at the same time we can download a file from net. These entire task will be executed simultaneously and independent of each other hence it is process based multitasking.

process based multi-tasking is based suitable at OS label.

2. **Thread based multitasking**

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

Thread based multitasking is based suitable at programmatic label.

Whether it is process based or thread based the main objective of multitasking is **reduce response of the system and to improve performance**.

The important application area of multithreading is

1. To develop multimedia graphics

2. To develop animations

3. To develop video games

4. To develop web servers and application servers etc

When compared with old languages developing multithreading applications in java is very easy because java provide in build support for multithreading with rich api[Thread,Runnable,Threadgroup ...]

## Thread

Thread is a flow of execution, for each thread has a separate job.

## Defining a thread

We can define a thread in the following two ways.

1. By extending thread class.

2. By implementing Runnable interface

1. By extending thread class

## Thread Schedular

It is the part of JVM, it is responsible to schedule threads that is if multiple threads are waiting to get the chance of execution then which other threads will be executed is decided by thread scheduler. We can't accept which algorithm followed by thread scheduler it is vary from JVM to JVM hence we can't expect thread execution order and exact output hence whenever situation coming in multithreading .There is no guarantee for exact output but we can provide several possible outputs.

Case 2. **Difference between t.start() and t.run()**

1. In the case of t.start() a new thread will be created which is a responsible for the execution of run method, but in the case of t.run a new thread won't be created and run method will be executed just like a normal method called by main thread

if we replace t.start() with t.run then output is child thread 10 times followed by main thread .

this total output produced by only main thread.

Case 3. Importance of start method

Thread class start method is responsible to **register the thread with thread scheduler** and all other mandatory activities hence, without executing start method there is no chance to start method in java due to this thread class start method is considered heart of multithreading

a.start()

{

1. Register thread with thread scheduler

2. perform all mandatory activities

3. invoke run() method

Case 4. Overloading of run method is always possible but thread class start method can invoke no argument run method the overloaded method we must call explicitly like normal method call.

If you are not overriding run method then thread class run method will be executed which has empty implementation hence

we won't get any output.

Class MyThread extend Thread

{

}

class Test

{

public static void main(String []args){

MyThread t=new MyThread();

t.start();

}

}

Output : No output

**Note**

It is highly recommended to override run method otherwise don't go for multithreading concept.

**case** 5 >overriding start method

If we override start method then our start method will be executed just like a normal method call and new thread won't be created.

Thread Life Cycle.

**case 6**-> After starting a Thread if we are trying to restart same thread then we will get run time exception saying illegal thread state exception.

Thread t=new Thread();

t.start();

t.start(); //illegal state execption

## Defining a thread by implemening Runnable interface

we can define a thread by implementing runnable interface

(i)myThread--------->Thread---------->Runnable()

(2)MyRunnable------>Runnable

Runnable interface presenting **java.lang** package and it contain only one method run()

**public void run();**

class MyRunnable implements Runnable{

public void run(){

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

sop("Child thread");-----------------------------------------------job of thread (executed by child thread)

}

}

}

class ThreadDemo {

public static void main(String []args){

MyRunnable r=new Runnalbe();

Thread t1=new Thread(r); ---r--->target runnable

t1.start();

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

sop("main thread");

}

}

}

we will get mixed output and we can't tell exact output

case study

====

MyRunnalbe r=new MyRunnalbe()

Thread t1=new Thread();

Thread t2=new Thread(r);

**case1. t1.start()**

-->A new thread will be created which is responsible for the execution of Thread class run method, which has empty implementation.

**case 2. t1.run()**

--->No new thread will be created and thread class run method will be executed just like a normal method call.

**case 3.t2.start()**

--->A new thread will be created which is responsible for the responsible of execution of MyRunnable run() method.

**case 4. t2.run()**

-->A new thread won't be created and my runnable run method will be executed just like a normal method call.

**case 5. r.start()**

--->we will get compile time error saying MyRunnable class doesn't have start capability.

(can't find symbol :method start() location myRunnable)

**case 6. r.run()**

-->No new thread will be created and MyRuunable run() will be executed like a normal method call.

Which approach is best to define a thread?

* Among two ways to defining a thread implements Runnable approach is recommended.
* In the first approach our class always extends Thread class, there is no chance of extending any other class hence we are missing inheritance benefit.
* But in the second approach while implementing Runnable interface we can extend any other class hence we won't miss any inheritance benefit.
* because of above reason implementing runnable interface approach is recommended then extending thread class.

## Thread Class Constuctors

**1.Thrad t=new Thread();**

**2.Thrad t=new Thread(Runnable r);**

**3.Thrad t=new Thread(String name);**

**4.Thrad t=new Thread(Runnalbe r,String Name);**

**5.Thrad t=new Thread(ThradGroup g,String Name);**

**6.Thrad t=new Thread(ThreadGroup g,Runnalbe r);**

**7.Thrad t=new Thread(ThreadGroup g,Runnalbe r,String Name);**

**8.Thrad t=new Thread(ThreadGroup g,Runnalbe r,String Name,long stacksize);**

Durga approach to define a thread (not recommended to use)

|  |
| --- |
| class MyThread extend Thrad  {  public void run()  {  sop("chiled thread")  }  }    class ThreadDemo  {  public static void main(String []args){  MyThread t=new MyThread();  Thrad t1=new Thread(t);  t.start();  sop("main thread")  }  }  output  child thread  main thread  or  main thread  child thread |

## Getting and Setting Name of a thread

Every thread in java has some name it may be default name generated by jvm or customizer name provide by programmer. We can get and set name of a thread by using following method

**public final string getName();**

**public final void setName(String name);**

## Thread Priority

Every thread in java has some priority it may be default priority generated my jvm or customize priority provided by programmer.

The valid range of thread priority is 1 To 10.

**Thread.Min.Priority=1**

**Thread.Norm.Priority=5**

**Thread.Max.Priority=10**

Thread scheduler will use priority while allocating processor, the thread which is having highest priority will get chance first. It two thread having same priority then we can't expect exact execution order, it depends on thread scheduler.

Thread class defined following method to get and set priority of a thread.

**public final int getPriority()**

**public final void setPriority(int p) if p range 1 to 10 otherwise run time exception :Illegal argument exception**

**Default Priority** ->The default priority only for the main thread is 5 for all remaining thread default priority will be inherited from parent to child

That is whatever priority parent thread has the same priority will be there for child thread.

**Program**

package oops.thread;

class MyThread3 extends Thread{

public void run(){

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

System.out.println("child thread");

}

}

}

public class ThreadPriority2 {

public static void main(String[] args) {

MyThread3 t1=new MyThread3();

// t1.setPriority(9); //line 1

t1.start();

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

System.out.println("Main thread "+t1.currentThread().getName()+"\t"+t1.getPriority());

}

}

}

If you are commenting line 1 then both main and child thread have same priority 5 and hence we can't expect execution order and exact output.If you are not commenting line 1 then main thread has priority 5 and child thread has priority 9 hence child thread will get chance 1st followed by main thread.

in this case output

## Prevent a thread execution

we can prevent a thread execution by using the following methods

1. yield()
2. join()
3. sleep()

### yield()

Yield method causes to pass current executing thread to give the chance for waiting thread 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, when it will get chance once again it depends on thread scheduler and we can't expect exactly.

**public static native void yield()**

class MyThread5 extends Thread{

public void run(){

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

System.out.println("Child thread");

Thread.yield(); --->line 1

}

}

}

public class ThreadYieldMethod {

public static void main(String[] args) {

MyThread5 t=new MyThread5();

t.start();

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

System.out.println("main thread");

}

}

}

In the above program if we are commenting line 1 then both thread will be execute simultaneously and we can't expect which thread will complete first.

If we are not commenting line 1 then child thread always call yield method because of that method thread will get a chance more number of times and chance of completing main thread 1st is high.

### Join

If a thread wants to wait until completing some other thread, then we should go for join method.

for Ex : 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 enter into waiting state until t2 completes.

Once t2 completes then t1 can continues its execution.

Ex:

**venue fixing wedding card printing wedding card distribution**

t1 t2 t3

t1.join t2.join

Wedding card printing thread(t2) has to wait until venue fixing thread(t1) completion hence t2 has to call t1.join() wedding card distribution thread(t3)has to wait until wedding card printing thread(t2) completion hence t3 has to call t2.join().

**final void join(); throw interrupted exception**

**final void join(long ms); throws Interrupted exception**

**final void join(long ms ,int ns); throw Interrupted**

**Note** ->Every join method throws interrupted exception which is checked exception hence compulsory we should handle this exception either by using

try catch or by throws keyword otherwise we will get compile time error.

**case 1** : waiting of main thread until completing child thread

example

class MyThread6 extends Thread{

public void run(){

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

System.out.println("Pooja's Thread ");

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

public class ThreadJoin2 {

public static void main(String[] args) {

MyThread6 t1=new MyThread6();

t1.start();

try {

// t1.join(); -->line 1

t1.join(1000); -->line 2

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

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

System.out.println("Binit's Thread");

}

}

}

**case1.**

if we comment line 1 then both main and child thread will be executed simultaneously and we can't expect output.

If we are not commenting line 1 then main thread call join method on child thread object hence main thread will wait until completing child thread.

**case 2** ->waiting of child thread until completing of main thread

package oops.thread;

// program for child thread will wait for main thread

class MyThread7 extends Thread{

static Thread mt;

public void run(){

try{

mt.join();

}catch(InterruptedException e){

e.printStackTrace();

}

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

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("child thread");

}

}

}

public class ThreadJoin3 {

public static void main(String[] args) {

MyThread7.mt=Thread.currentThread();

MyThread7 t=new MyThread7();

t.start();

// t.join(); //deadlock

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

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("main thread");

}

}

}

-->case 3

In the above program child thread calls join method on main thread object hence child thread must wait until completing main thread.

If main thread calls join method on child thread and child thread calls join method on main thread object both thread will wait forever and the program will be stuck.(This is something like **deadlock**)

**case 4**

if thread calls join method on the same thread itself then the program will be stuck(this is something like deadlock)

In this case thread has to wait infinite amount of time.

Ex:

public class ThreadJoin4 {

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

Thread.currentThread().join();

}

### Sleeping

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

**public static native void sleep(long ms) throws interrupted exception**  which is checked exception

**public static native void sleep(long ms,int ms) throws interrupted exception**

hence whenever we are using sleep method compulsory we should handle interrupted exception either by try/catch or throws keyword otherwise we will get compile time error.

Example:

public class ThreadSleep1 {

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

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

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

Thread.sleep(5000);

}

}

}

## How a thread can interrupt another thread

A thread can interrupt a sleeping thread or waiting thread by using interrupt of thread class.

public void interrupt

class MyThread8 extends Thread{

public void run(){

try{

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

System.out.println("i am lazzy");

Thread.sleep(2000);

}

}catch(InterruptedException ie){

System.out.println("i am interrupted");

}

}

}

public class ThreadInterrupt {

public static void main(String[] args) {

MyThread8 t1=new MyThread8();

t1.start();

t1.interrupt(); -->line 1

System.out.println("End of Main Thread");

}

}

}

If we comment line 1 then main thread won't interrupt child thread , in this case child thread will execute for 10 times

If we are not commenting line 1 then main thread interrupt child thread.

**Note**->whenever we are calling interrupt method if target thread not in sleeping/waiting state then there is no impact of interrupt call immediately

Interrupt call will be waited until target thread enter sleeping/waiting state. if the target thread enters into sleeping/waiting state

then immediately interrupt call will interrupt the target thread.

If the target thread never enter into sleeping/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.

class MyThread9 extends Thread{

public void run(){

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

System.out.println("I am lazy thread "+i);

}

System.out.println("I want to sleep");

try{

Thread.sleep(5000);

}catch(InterruptedException i){

System.out.println("I am interrupted");

}

}

}

public class ThreadInterrupt2 {

public static void main(String[] args) {

MyThread9 t=new MyThread9();

t.start();

t.interrupt();

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

}

}

In the above example interrupt call waited until child thread completed for 100 times.

## Different b/w yield, join and sleep method

|  |  |  |  |
| --- | --- | --- | --- |
| Properties | Yield | Join | Sleep |
| Purpose | If a thread wants to pass its execution to give the remaining thread of same priority, then we should go for yield method | If a thread wants to wait until completing some other thread, then we should go for join method | If a thread doesn’t want to perform any operation for a particular amount of time then we should go for sleep method |
| Is it overloaded? | No | Yes | Yes |
| Is it final? | No | Yes | No |
| It it throws interrupted exception? | No | Yes | Yes |
| Is it native? | Yes | No | Sleep(long) 🡪native  Sleep(long ms,int ns)->native |
| Is it static ? | Yes | No | Yes |

## Synchronization

**Synchronizer** :

1. synchronizer is a modifier applicable only for method and block but not for classes and variable.
2. **If a multiple thread trying to operate simultaneously at the same java object, then may be a chance of data inconsistency problem. To overcome this problem, we should go for synchronizer keywords.**
3. If a method or block declared as **synchronizer**, then at a time only one thread can execute that method or block on the given object so that data inconsistency problem will be resolve.
4. The main advantage of synchronizer keyword is we can **resolve data inconsistency problem** but the main disadvantage of synchronizer keyword is **it increases waiting time of threads and the create performance problems** hence if there are no any specific requirements then it is not recommended synchronizer keywords.
5. Internally synchronization is implemented by using **lock,** every object in java has a unique lock. Whenever we are using synchronizer keyword then only lock concept come into picture.
6. If a thread wants to execute synchronizer method on the given object first it must get lock of that object. Once thread got the lock then it is allowed to execute any synchronizer method and that object. Once method execution completes automatically thread release locks.
7. Acquiring and releasing lock internally take care by jvm and programmer not responsible for this activity.
8. While a thread executing synchronizer method on the given object the remaining thread are not allowed to execute any synchronizer method simultaneously on the same object. but remaining thread are allowed to execute non-synchronizer method simultaneously.

Class x

{

Synchronizer m1()

Synchronizer m2()

m3()

}

T1 start execute m1

T2 execute m1 (waiting stage)

T3 execute m2 (waiting stage)

T4 execute m3

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

Java object

**Non-Synchronized Area:** This area can be accessed by any number of thread simultaneously

**Synchronized Area:**This area can be accessed by only one thread at a time.

Class X {

Synchronized Area {

Wherever we are performing update operation (**add/update/delete**) i.e. where state of object changing

}

Non-synchronized Area {

Wherever object state won’t be change like read operation.

}

}

Class ReservationSystem

{

checkAvailabilityofTicket ()-------------------🡪**Non-synchronized Area**

{

Just read operation

}

bookTicker(){-----------------------🡪**Synchronized Area**

update

}

}

**Program (use of synchronized)**

**package** oops.thread.synchronization;

**class** Display {

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

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

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

**try**{

Thread.*sleep*(2000);

}**catch**(InterruptedException ie){

}

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

}

}

}

**class** MyThread10 **extends** Thread{

Display d;

String name;

MyThread10(Display d2,String name){

**this**.d=d2;

**this**.name=name;

}

**publicvoid** run(){

d.wish(name);

}

}

**class** SynchronizationDemo{

**publicstaticvoid** main(String[] args) {

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

MyThread10 t1=**new** MyThread10(d,"Binit");

MyThread10 t2=**new** MyThread10(d,"Binay");

MyThread10 t3=**new** MyThread10(d,"Abhay");

MyThread10 t4=**new** MyThread10(d,"Manoj");

t1.start();

t2.start();

t3.start();

t4.start();

}

}

If we are not declaring wish method as synchronized then both thread will be executed simultaneously and hence we will be get irregular output.

If we declare wish method as synchronized then only one thread is allowed to execute wish method and given display object hence we will be get regular output.

**Case Study**

Display d1=new Display();

Display d2=new Display();

MyThread10 t1=new MyThread10(d1,"Binit");

MyThread10 t2=new MyThread10(d2,"Binay");

t1.start();

t2.start();

t1🡪 d1 wish Binit

t2🡪 d2 wish “Binay”

1. Even though wish method is synchronized we will get regular output because threads are operating on different java object.

**Conclusion** :

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

### Class Level Lock

Every class in java has a unique lock which is nothing but class level lock if thread wants to execute static synchronizer 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 release the lock.

While a thread executing static synchronized method, the remaining thread are not allowed to execute static synchronized method of that class simultaneously but remaining thread are allowed to execute the following method simultaneously.

1. normal static method

2.synchronized instance method

3. normal instance methods

Class X {

static synchronized m1()

static synchronized m2()

Static m3()

Synchronized m4()

m5()

}

t2 m1() T1->cl(x)

t3 m2() m1(){

t4 m3()

t5 m4() }

t6 m5()

t2 and t3 goes waiting state and t4,t5,t6 will execute.

**package** oops.thread.synchronization;

**class** Display1{

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

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

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

Thread.*sleep*(2000);

}

}

**Public synchronized void** displayc() **throws** InterruptedException{

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

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

Thread.*sleep*(2000);

}

}

}

**class** MyThread11 **extends** Thread{

Display1 d1;

MyThread11(Display1 d1){

**this**.d1=d1;

}

**publicvoid** run(){

**try** {

d1.displayn();

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

}

}

**class** MyThread12 **extends** Thread{

Display1 d1;

MyThread12(Display1 d1){

**this**.d1=d1;

}

**publicvoid** run(){

**try** {

d1.displayc();

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

}

}

**publicclass** SynchronizationDemo1 {

**publicstaticvoid** main(String[] args) {

Display1 d=**new** Display1();

MyThread11 t1=**new** MyThread11(d);

MyThread12 t2=**new** MyThread12(d);

t1.start();

t2.start();

}

}

* If synchronized keyword is not used in method displayn and displayc then it will be print number and character simultaneously .
* If synchronized keyword is used in method displayn and displayc then it will be print number and character one by one.

### Synchronized Block

If very few lines of code required synchronization, then it is not recommended declaring entire method synchronized 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 thread and improves performance of the system**.

We can declare synchronized block as follows.

1. **To get lock of current object**

Synchronized(this){

------

-------

-----

}

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

1. **To get lock of particular object**

Synchronized(b){

---

---

---

}

If a thread got lock of particular object ‘b’ then only 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.

Prorgram (synchronized block)

**package** oops.thread.synchronization;

**class** Display2{

**publicvoid** wish(String name){

;;;;;;;;;;;;;; //1 lack lines of code

// synchronized(this) //synchronized block object level lock

**synchronized**(Display2.**class**)

{

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

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

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

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

}

}

;;;;;;;;;;;;;;;;;; //1 lack lines of code

}

}

**class** MyThread13 **extends** Thread{

Display2 d;

String name;

MyThread13(Display2 d,String name){

**this**.d=d;

**this**.name=name;

}

**publicvoid** run(){

d.wish(name);

}

}

**publicclass** SynchornizationBlockDemo2 {

**publicstaticvoid** main(String[] args) {

Display2 d=**new** Display2();

MyThread13 t1=**new** MyThread13(d,"Dhoni");

MyThread13 t2=**new** MyThread13(d,"Youraj");

t1.start();

t2.start();

}

}

**Note**🡪Lock concept is applicable for object types and class Types but not for premitives hence we can’t pass primitive type as argument to synchronized block otherwise we will get compile time error saying unexpected type found int required reference.

Int x=10;

synchronized(x){

}

### FAQs

1. What is synchronized keyword where we can apply.
2. Explain advantage of synchronized keyword.
3. Explain disadvantage of synchronized keyword.
4. What is **race condition**

### race condition

If multiple threads are operating simultaneously on same java object, then there may be a chance of data inconsistency problem.

This is called race condition we can overcome this problem by using synchronized keyword.

1. What is object lock and when it is required.
2. What is class level lock and when it is required?
3. What is the difference b/w class level and object level lock
4. While a thread executing synchronized method and the given object is remaining thread are allowed to execute any other synchronized method simultaneously on the same object.

Ans ; No

1. What is synchronized block.
2. How to declare synchronized block to get lock of current object.
3. How to declare synchronized block to get lock of class level lock.
4. What is the advantage of synchronized block over synchronized method.
5. Is a thread can acquire multiple lock simultaneously?

Yes

From different objects

Class X{

Public synchronized void m1(){

Y y=new Y();

Synchronized(Y){

}

}

}

X x=new X();

1. What is **synchronizer statement**

The statement present synchronized method and synchronized block are called synchronized statements.

## Inter Thread Communication

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

**Wait() :** Tells the current thread to release the lock and go to sleep until some other thread enters the same monitor and calls notify().

**Nofify() :** wakes up a single thread that is waiting on this object’s monitor.

**NotifyAll() :** It wakes up all the threads that called wait() on the same object.

Thread which is expecting updation is responsible to call wait method then immediately the thread will enter into waiting state.

1. The thread which is responsible to perform operation, after updating it is responsible to call **notify** method then waiting thread will get the notification and continue its execution with those updated items.
2. **Wait ,notify, notifyAll** method present in object class but not in thread class because thread can call these methods on any java object.
3. To call wait or **notify /notifyAll** methods on any object, thread should be owner of that object that is the thread should has lock of that object that is thread should be inside synchronized area.

Hence we can call wait ,notify/notifyAll only from synchronized area otherwise we will get run time exception saying **IllegalMonitorStateexception**.

1. If a thread calls wait method on any object it immediately releases lock of that particular object and enter into waiting state.
2. If a thread calls notify method on any object it releases the lock of that object may not immediately except **wait,notify/notifyAll**there is no other method where thread releases the lock.

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

True or false

1. If a thread 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 the object but may not immediately—invalid
3. If a thread calls wait method on any object it releases all locks acquired by the thread and immediately enter into waiting state—invalid (it will release lock particular object)
4. If a thread calls wait method on any object it immediately releases the lock of that particular object and enter into waiting state.-- Valid
5. If a thread calls **notify** method on any object it immediately releases the lock of that particular object.—invalid (not immediatly)
6. If a thread calls notify method on any object it release the lock of that object but may not immediately.--valid

public final void wait() throws Interrupted Exception

public final native void wait(long ms) throws Interrupted Exception

public final void wait(long ms,int ns) throws interrupted Exception

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 compulsory we should handle this interrupted exception either by try/catch or by throws keywords otherwise we will get compile time error.

Diagram

**1.if waiting thread**

**Got notification**

**2.if time expires**

**3.if waiting thread got interrupted**

**Obj.wait();**

**Obj.wait(1000);**

**If waiting got lock Obj.wait(1000,100);**

Mt t=new Mt();

**t.start() If TS allocate processor**

**If run method got Completed**

Program

package oops.thread.synchronization;

class MyThread14 extends Thread{

int total=0;

public void run(){

synchronized (this) {

System.out.println("Child thread starts calculation...");//(2)

for(int i=1;i<=100;i++){

total=total+i;

}

System.out.println("child thread trying to give notification"); // (3)

this.notify();

}

}

}

class SynchronizationDemo3 {

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

MyThread14 t=new MyThread14();

t.start();

synchronized (t) {

System.out.println("Main thread trying to call wait() method"); //(1)

t.wait(); //main thread waiting state

System.out.println("Main thread got notification");//(4)

System.out.println("Total =="+t.total); //(5)

}

}

}

Output

Main thread trying to call wait() method

Child thread starts calculation...

child thread trying to give notification

Main thread got notification

Total ==5050

### Producer Consumer Problem

Producer thread is responsible to produce item to the queue and consumer thread is responsible to consume items from the queue. if queue is empty then consumer thread will call wait method and entered into waiting state.

After producing item to the queue producer thread is responsible to call notify method then waiting consumer will get that notification and continue its execution with updated items.

producer Thread Consumer Thread

|  |  |
| --- | --- |
| **Producer** | **Consumer** |
| Class Producer{  Produce(){  Synchronized(q){  Produce item to the queue  q.notify()  }  }  } | Class Consumer{  Consume(){  Synchronized(q){  If(q is empty){  q.wait()  else  Consume item from the queue  }  }  } |

### Difference between notify and notifyAll

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

We can use **notifyAll** to give the notification for all waiting threads of a particular object even though multiple threads notified but execution will be performing one by one because thread requires lock and only one lock is available.

**Note :** and which object we are calling wait method thread require the lock of that particular object

**Ex :** If we are calling wait method on s1 then we have to get lock on s1 object but not s2 object

|  |  |
| --- | --- |
| Synchronized(s1){  ---  ---  S2.wait()  }  Will get **IllegalMonitorState Exception** | Synchronized(s1){  ---  ---  S1.wait()  } |
|  |  |

## DeadLock

If two threads are waiting for each other forever such type of infinite waiting is called **deadlock.**

Synchronized is the only reason for deadlock situation hence while using synchronized keyword we have to take special care. There is no resolution technique of deadlock but several prevention technique are available.

Prorgram

**package** oops.thread;

**class** A{

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

System.***out***.println("Thread1 start executing d1 method");

**try**{

Thread.*sleep*(3000);

}**catch**(InterruptedException ie){

ie.printStackTrace();

}

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

b.last();

}

**Public synchronized void** last(){

System.***out***.println("Inside A list method");

}

}

**class** B{

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

System.***out***.println("Thread2 start executing d2 method");

**try**{

Thread.*sleep*(3000);

}**catch**(InterruptedException ie){

ie.printStackTrace();

}

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

a.last();

}

**Public synchronized void** last(){

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

}

}

**class**DeadLock**extends** Thread{

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

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

**Public void** m1(){

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

**this**.start();

a.d1(b); //main thread

}

**Public void** run(){

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

b.d2(a); //child thread

}

}

**publicclass** DeadLockDemo {

**publicstaticvoid** main(String[] args) {

DeadLockt1=**new**DeadLock();

t1.m1();

}

}

In the above program if we remove at least one synchronized keyword then the program won’t enter into deadlock hence synchronized keyword is only reason for deadlock situation due to this while using synchronized keyword we have to take special care.

## DeadLock vs Starvation

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

Whereas long waiting of a thread where waiting ends at certain points is called **starvation**.

For ex : Low priority thread has to wait until completing all high priority threads it may be long waiting but ends at certain point which is nothing but starvation.

## Daemon Thread

The thread which are executing in the background are called daemon thread

**Ex :**  Garbage collector ,Signal dispatcher ,attach listener

The main objective of daemon thread is provide backend support for non-daemon threads(main thread).

For ex : If main threads runs with low memory then jvm runs garbage collector to destroy useless object so that number of bytes of free memory will be improve with this free memory main thread can continue its execution .

Usually daemon threads having low priority but based on our requirements daemon threads can run with high priority also.

We can check daemon nature of a thread by using **isDaemon** method of thread class.

**public Boolean isDaemon()**

we can change daemon nature of a thread by using set daemon method

**public void setDaemon(boolean b)**

but changing daemon nature is passible before starting of a thread only after starting a thread if we are trying to change daemon nature then we will get runtime exectption saying **illegalthreadstateException**

### Default nature of a thread

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

It is impossible to daemon nature of main thread because it is already started by jvm at beginning.

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

**package** oops.thread.daemonThread;

**class** MyThread16 **extends** Thread{

}

**publicclass** DaemonThreadDemo {

**publicstaticvoid** main(String[] args) {

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

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

MyThread16 t=**new** MyThread16();

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

t.setDaemon(**true**);

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

}

}

2.

**package** oops.thread.daemonThread;

**class** MyThread17 **extends** Thread{

**publicvoid** run(){

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

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

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**publicclass** DaemonThreadDemo2 {

**publicstaticvoid** main(String[] args) {

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

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

MyThread17 t=**new** MyThread17();

t.start();

// t.setDaemon(true); //line 1

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

System.***out***.println("Main thread end");

}

}

If we are commenting line 1 both main and child thread are non-daemon and hence both thread will be executed until their completion.

If we are not commenting line 1 then main thread is non-daemon and child is daemon hence whenever main thread terminates automatically child thread will be terminated.

## Green Thread

Java multithreading concept is implemented is following two models.

1. Green thread model
2. Native os model

### Green Thread model

The thread which is completely managed by jvm without taking underlying os support is called **Green thread**.very few operating system like sun solaries provide support for green thread model anyway green thread model is deprecated and not recommended to use.

1. **Native OS Model**

The thread which is managed by jvm with the help of underlying os,is called native os model.all window based operating system provide support for native os model.

## How can we stop a thread

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

public void **stop**()

if we call stop method then immediately thread is enter into dead state anyway stop method is deprecated and not recommended to use.

## How to suspend and resume of a thread

We can suspend a thread by using suspend method of thread class then immediately the thread will be enter into suspended state. we can resume a suspended thread by using resume method of thread class then thread suspended thread can continue its execution.

**public void suspend()**

**public void resume()**

anyway this method are deprecated and not recommended to use.

## ThreadGroup

Based on functionality we can group thread into a single unit which is nothing but **ThreadGroup**contains a group of threads

In addition to thread threadGroup also contains sub Threadgroups.

The main advantage of ThreadGroup maintaining threads in the form ThreadGroup is **we can perform common operation very easly**.

Every thread in java belongs to some group main thread belongs to main group

Every thread group in java is the child group of **System group** either directly or indirectly hence system group is root of all threadgroups in java.

System Group contains several system label threads like

**finalizer ,**

**referece handler,**

**signal dispatcher,**

**attach listener.**

ThreadGroup is a class of java.lang package and it is direct child class of **object**.

### Constructor :

ThreadGroup g=new ThreadGroup(String gname)

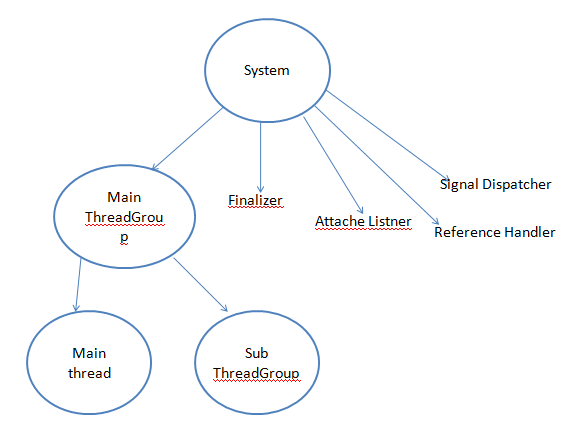
Creates a new thread group with the specified group name,the parent of this new group is the thread group of currently executing thread.

ThreadGroup g=new ThreadGroup(“FirstGroup”)

ThreadGroup g=new ThreadGroup(ThreadGroup g,String GroupName)

Create a ThreadGroup with the specified group name.The parent of new Thread group is specified parent group

ThreadGroup g1=new ThreadGroup(g,”SecondGrup”)



**Program**

**==**

**publicclass** ThreadGroupDemo1 {

**publicstaticvoid** main(String[] args) {

System.***out***.println(Thread.*currentThread*().getThreadGroup().getName());//main

System.***out***.println(Thread.*currentThread*().getThreadGroup().getParent().getName()); //system

}

}

Program2

**publicclass** ThreadGroupDemo1 {

ThreadGroup g1=**new** ThreadGroup("FirstGroup");

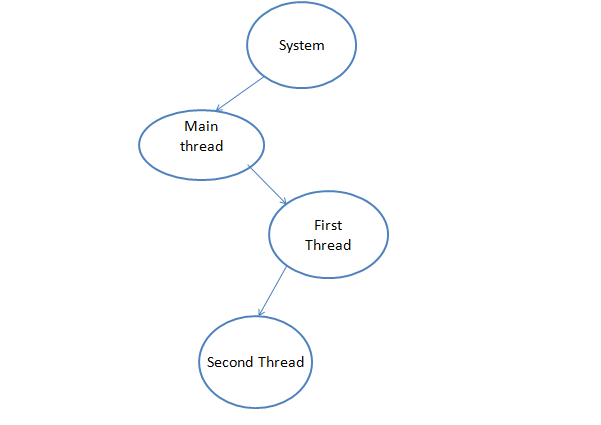
System.***out***.println(g1.getParent().getName());

ThreadGroup g2=**new** ThreadGroup(g1,"SecondGroup");

System.***out***.println(g2.getName());

System.***out***.println(g2.getParent().getName());

}

}

### Important methods ThreadGroup class

1. **String getName()** :returns name of the threadGroup
2. **Int getMaxPriority()** : returns max priority of ThreadGroup
3. **Void setMaxPriority(int p):** to set maximum priority of threadGroup

The default max priority is 10

Threads in the threadgroup that have already have higher priority won’t be affected but newly added threads this max priority will applicable.

ThreadGroup g1=**new** ThreadGroup("FirstGroup");

Thread t1=**new** Thread(g1,"Thread1");

Thread t2=**new** Thread(g1,"Thread2");

g1.setMaxPriority(3);

Thread t3=**new** Thread(g1,"Thread3");

System.***out***.println(t1.getPriority()); //5

System.***out***.println(t2.getPriority()); //5

System.***out***.println(t3.getPriority()); //3

1. **ThreadGroup getParent()**: returns parent group of current thread
2. **void list** : it prints information about threadgroup to the console.
3. **int activeCount** : returns number of active thread presents in the threadgroup
4. **int active groupCount()** : returns number of active group presents in the current thread group.
5. **int enumerate(thread[] t)** : to copy all active thread of this threadgroup into provided thread array.in this subthread group thread will be considered.
6. **Int enumerate(ThreadGroup[] g)** : to copy all active subthread groups into threadgroup array.
7. **Boolean(isDaemon)** : to check threadGroup is daemon or not
8. **Void setDaemon(Boolean b)**: to set daemon
9. **Void interrupt** : to interrupt all sleeping/waiting thread present in the threadGroup
10. **Void destroy** : to destroy threadgroup and it’s subthread groups.

ThreadGroup system=Thread.*currentThread*().getThreadGroup().getParent();

Thread[] t=**new** Thread[system.activeCount()];

system.enumerate(t);

**for**(Thread t1:t){

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

}

Output :

Reference Handler....true

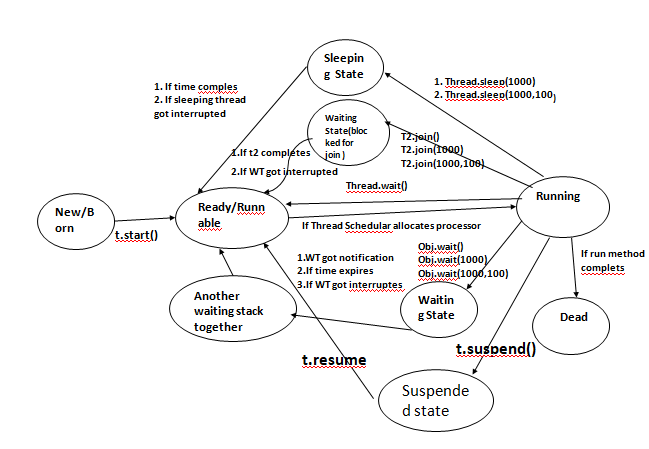
Finalizer....true

Signal Dispatcher....true

Attach Listener....true

main....false

## Thread Life Cycle



In above diagram in running to suspend state t.suspend in place of t.start().

## Java Util.concurrent Package

The problems with traditional synchronizer keyword

1. We are not having any flexibility to try for a lock without waiting.
2. There is no way to specify maximum waiting time for a thread to get lock so that thread will wait until getting the lock which may creates performance problems which may cause deadlock.
3. If a thread releases the lock then which waiting thread will getting that lock we are not having any control on this.
4. There is no API to list out all waiting thread for a lock.
5. Synchronizer keyword compulsory we have to use either at a method or within a method and it is not possible to use across multiple methods.

To overcome this problems Sun people introduce java.util.concurrent.locks package in java 1.5 version. It also provide several enhancement to the programmer to provide more control on concurrency.

### lock interface

lock object is similar to implicit lock acquired by a thread to execute synchronizer method or synchronizer block. Lock implementation provide more extensive operation then traditional implicit locks.

### Important methods of Lock Interface

1. **void lock()** : we can use this method to acquire a lock if the lock is already available then immediately current thread will get that lock, if the lock is not already available then it will wait until getting the lock it is exactly same behavior of traditional synchronized keyword.
2. **boolean tryLock():** to acquire the lock without waiting ,if the lock is available then thread acquires that lock and returns true if the lock is not available then this method returns false and continue its execution without waiting,in this case thread never be enter into waiting state.

If(l.tryLock()){

Perform safe operation

}

else {

Perform alternative operation

}

1. **boolean tryLock(long time,TimeUnit unit)** : If lock is available then thread will get the lock and continue its execution. If the lock is not available then thread will wait until specified amount of time still if lock is not available then thread can continue it’s execution.

**TimeUnit**

Time unit is a enum present in java.util.concurrent package.

enum TimeUnit

{

NANOSECONDS,

MICROSECONDS,

MILLISECONDS,

SECONDS,

MINUTS,

HOURS,

DAYS

}

1. **Void lockInterruptibly()**

Acquire a lock if it is available and returns immediately if lock is not available then it will wait, while waiting if thread is interrupted then thread won’t get the lock.

1. **void unlock()** : to release a lock.

To call this method compulsory current thread should be owner of the lock otherwise we will get runtime exception saying illegalMonitorState exception.

## ReentrantLock()

It is the implementation class of lock interface and it is the direct child class of object.

ReentrantLock means a thread can acquire same lock multiple times without any issue.internally ReentrantLock lock increment thread’s personal count whenever we call lock method and decrement counter whenever thread call unlock method and lock will be release whenever count reaches 0.

### Constructor :

ReentrantLock l=new ReentrantLock ();

1. Create an instance on ReentrantLock lock.
2. ReentrantLock l=new ReentrantLock (boolean fairness)

Creates ReentrantLock with given fairness policy if the fairness is true then longest waiting thread can acquire the lock if it is available i.e it follow FCFS policy.

If fairness is false then which waiting thread will get the chance we can’t expect.

Note : The default value of for fairness is false.

Which of the following declaration are equal

* 1. ReentrantLock l=new ReentrantLock ()
  2. ReentrantLock l=new ReentrantLock (true)
  3. ReentrantLock l=new ReentrantLock (false)
  4. all the above

Ans : a and c

### Important method of ReentrantLock

1. **void lock()**
2. **boolean tryLock()**
3. **boolean tryLock(long l,TimeUnit**)
4. **void lockInterruptibly()**
5. **void unlock()**
6. **getHoldCount ()** : returns number of Hold on this lock by current thread.
7. **boolean isHeldByCurrentThread**: return true if and only if lock is hold by current thread
8. **int getQueueLength()** : returns number of thread waiting for the lock
9. **Collection getQueuedThreads()** : it returns collection of thread which are waiting to get the lock.
10. **boolean hasQueuedThreads()** : returns true if any thread is waiting to get the lock.
11. **boolean isLocked():** returns true if the lock is acquire by some thread.
12. **boolean isFair():** returns true if the fairness policy is set true value.
13. **Thread getOwner():** returns the thread which acquired the lock.

public class ReentrantLockDemo1 {

public static void main(String[] args) {

ReentrantLock l=new ReentrantLock();

l.lock();

l.lock();

System.out.println(l.isLocked());

System.out.println(l.isHeldByCurrentThread());

System.out.println(l.getQueueLength());

l.unlock();

System.out.println(l.getHoldCount());

System.out.println(l.isLocked());

l.unlock();

System.out.println(l.isLocked());

System.out.println(l.isFair());

}

}

Program

==

**package** oops.thread.synchronization;

**import** java.util.concurrent.locks.ReentrantLock;

**class** Display15{

ReentrantLock l=**new** ReentrantLock();

**publicvoid** wish(String name){

l.lock(); // line1

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

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

**try** {

Thread.*sleep*(2000);

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

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

}

l.unlock(); //line 2

}

}

**class** MyThread15 **extends** Thread{

Display15 d;

String name;

**public** MyThread15(Display15 d, String name) {

**super**();

**this**.d = d;

**this**.name = name;

}

**publicvoid** run(){

d.wish(name);

}

}

**publicclass** SynchronizationDemo4 {

**publicstaticvoid** main(String[] args) {

Display15 d=**new** Display15();

MyThread15 t1=**new** MyThread15(d,"binay");

MyThread15 t2=**new** MyThread15(d,"binit");

t1.start();

t2.start();

}

}

If we comment line1 and line2 then thread will be executed simulentiously and we will be get irregular output if we are not commenting line1 and line2 then threads will be executed one by one and we will get regular output.

### Demo program for try lock method

**package** oops.thread.ReentrantLock;

**import** java.util.concurrent.locks.ReentrantLock;

**class** MyThread16 **extends** Thread{

**static** ReentrantLock *l*=**new** ReentrantLock();

MyThread16(String name){

**super**(name);

}

**publicvoid** run(){

**if**(*l*.tryLock()){

System.***out***.println(Thread.*currentThread*().getName()+"..got lock and performing safe operations");

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

*l*.unlock();

}**else**{

System.***out***.println(Thread.*currentThread*().getName()+"...unable to get lock and trying to perform alternate operations");

}

}

}

**publicclass** TryLockDemo {

**publicstaticvoid** main(String[] args) {

MyThread16 t1=**new** MyThread16("First Thread");

MyThread16 t2=**new** MyThread16("Second Thread");

t1.start();

t2.start();

}

}

### Program program for Trylock with timeunit

**package** oops.thread.ReentrantLock;

**import** java.util.concurrent.TimeUnit;

**import** java.util.concurrent.locks.ReentrantLock;

**class** MyThread17 **extends** Thread{

**static** ReentrantLock *l*=**new** ReentrantLock();

MyThread17(String name){

**super**(name);

}

**publicvoid** run(){

**do**{

**try** {

**if**(*l*.tryLock(3000, TimeUnit.***MILLISECONDS***)){

System.***out***.println(Thread.*currentThread*().getName()+"..got the lock performing safe operation");

Thread.*sleep*(30000);

*l*.unlock();

System.***out***.println(Thread.*currentThread*().getName()+"Releasing the lock");

**break**;

}**else**{

System.***out***.println(Thread.*currentThread*().getName()+"...unable to get lock and will trying again ...");

}

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

}**while**(**true**);

}

}

**publicclass** TryLockWithUnitTimeDemo {

**publicstaticvoid** main(String[] args) {

MyThread17 t1=**new** MyThread17("First Thread");

MyThread17 t2=**new** MyThread17("Second Thread");

t1.start();

t2.start();

}

}

## Thread Pools(Executor Framework)

Creating a new thread for every job may create performance and memory problems, to overcome this we should go for thread pool. **Thread pool is pool of already created thread ready to do our job**. java 1.5 version introduces threadpool framework to implement thread pools. Thread pool framework also known as **Executor Framework**.

We can create a thread pool as follows

**ExecutorService service=Executors.newFixedThreadPool(3);**

We can submit a runnable job by using submit() method

**service.submit(job)**

we can shut down executor service by using shutdown() method

**service.shutdown()**

### Demo program of executor Framework

**package** oops.thread.ExecutorFramework;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**class** PrintJob **implements** Runnable{

String name;

PrintJob(String name){

**this**.name=name;

}

**Public void** run(){

System.***out***.println(name+"...Job started by Thread: "+Thread.*currentThread*().getName());

**try**{

Thread.*sleep*(2000);

}**catch**(InterruptedException e){}

System.***out***.println(name+"..job completed by Thread:"+Thread.*currentThread*().getName());

}

}

**publicclass** ExectorFrameworkDemo1 {

**publicstaticvoid** main(String[] args) {

PrintJob[] jobs={**new** PrintJob("Binit"),

**new** PrintJob("Binay"),

**new** PrintJob("Abhay"),

**new** PrintJob("Manoj"),

**new** PrintJob("Raja"),

**new** PrintJob("Lalan Singh")

};

**ExecutorService service=Executors.*newFixedThreadPool*(3);**

**for**(PrintJob job:jobs){

service.submit(job);

}

service.shutdown();

}

}

In above example 3 thread are responsible to execute 6 job so that a single thread can be reused for multiple job.

**Note** : while developing webserver/application server you can use thread pool concept.

## Callable and Future

In the case of **runnable** job thread won’t return anything after completing the job if a thread is required to return some result after execution then we should go for **callable**.

Callable interface contain only one method **call**

**public object call() throws exception**

ifwe submit callable object to executor then after completing the job thread returns an object of the type **Future**.that is **Future** object can be used for retrieve the result from callable job.

Program

**package** oops.thread.ExecutorFramework;

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**class** MyCallable **implements**Callable{

**int**num;

**public** MyCallable(**int**num) {

**this**.num=num;

}

@Override

**public** Object call() **throws** Exception {

System.***out***.println(Thread.*currentThread*().getName()+"is responsible to find sum of "+num+"numbers");

**int**sum=0;

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

sum=sum+i;

}

Thread.*sleep*(2000);

**return**sum;

}

}

**publicclass** CallableDemo {

**publicstaticvoid** main(String[] args) **throws** InterruptedException, ExecutionException {

MyCallable[] jobs={ **new** MyCallable(10),

**new** MyCallable(20),

**new** MyCallable(30),

**new** MyCallable(40),

**new** MyCallable(50),

**new** MyCallable(60),

};

ExecutorService service=Executors.*newFixedThreadPool*(3);

**for**(MyCallable job:jobs){

Future f=service.submit(job);

System.***out***.println(f.get());

}

service.shutdown();

}

}

#### Difference between Runnable and Callable

|  |  |
| --- | --- |
| **Runnable** | **Callable** |
| 1. If a thread is not required to return anything after completing the job then we should go for runnable | 1. If a thread required to return something after completing the job then we should go for callable. |
| 1. Runnable interface contain only one method **run()**method | 1. Callable interface contain onle one **call()** method. |
| 1. Runnable not return anything and hence return type of run() method is **void** | 1. Callable job is required return something and hence return type of call() method is **object** |
| 1. Within run method if there is any chance of checked exception compulsory we should handle by using **try-catch** we can’t used throws keyword for run method. | 1. Within call method if there is any chance of rising checked exception we are not required to handle by using try catch because call method already throws exception |
| 1. Runnable interface present in **java.lang** package | 1. Callable interface present In **java.util.concurrent** package |
| 1. Introduce in **1.0 version** | 1. Introduce in **1.5 version** |

## ThreadLocal

Threadlocal class provides thread local variables .threadlocal class maintain values per thread basis.

Each threadlocal object maintain a separate value like userid,transactionid etc.

For each thread that access that object.Thread can access its local value can manipulate its value even can remove its value.In every part of the code which is executed by the thread we can access its local variable.

Consider a servlet which invokes some business methods.we have a requirement to generate a unique transaction id for each and every request and we have to pass this transaction id to the business methods for this requirements we can use threadlocal to maintain a separate transactionid for every request id for every thread.

Note :

1. Thread local class introduced in 1.2 version and enhanced in 1.5 version
2. ThreadLocal can be associated with threadscope.
3. Total code which is executed by the thread has access to the corresponding local variable.
4. A thread can access its own local variables and can’t access other threads local variables.
5. Once thread enter into dead state all its local variable are by default available for eligible for garbase collection.

#### Methods :

1. **object get()** : returns the value of threadlocal variable associated with current thread
2. **object initalValue()** : returns initial value of threadlocal variable associated with current thread

the default implementation of this method returns null.

To customize our own initial value we have to override this method.

1. **void set(object newValue)** : to set a new value.
2. **void remove()** : to remove the value of threadlocal variable associated with current thread.

It is newly added method in 1.5 version.

After removal we are trying to access it will reinitialize by invoking it is initial value method.

**program**

**package** oops.thread.threadlocal;

**publicclass** ThreadLocalDemo1 {

**publicstaticvoid** main(String[] args) {

// ThreadLocal tl=new ThreadLocal();

// overriding initialValue() method

ThreadLocaltl=**new**ThreadLocal()

{

**public** Object initalValue()

{

**return**"abc";

}

};

System.***out***.println(tl.get());

tl.set("Binit");

System.***out***.println(tl.get());

tl.remove();

System.***out***.println(tl.get());

}

}

**Program 2**

**package** oops.thread.threadlocal;

**class** CustomerThread **extends** Thread{

**static** Integer *custId*=0;

**privatestatic**ThreadLocal*tl*=**new**ThreadLocal()

{

**protected** Integer initialValue()

{

**return** ++*custId*;

}

};

CustomerThread(String name)

{

**super**(name);

}

**publicvoid** run(){

System.***out***.println(Thread.*currentThread*().getName()+"executing with Customer id :"+*tl*.get());

}

}

**publicclass** ThreadLocalDemo2 {

**publicstaticvoid** main(String[] args) {

CustomerThread t1=**new** CustomerThread("CustomerThread--1");

CustomerThread t2=**new** CustomerThread("CustomerThread--2");

CustomerThread t3=**new** CustomerThread("CustomerThread--3");

CustomerThread t4=**new** CustomerThread("CustomerThread--4");

t1.start();

t2.start();

t3.start();

t4.start();

}

}

Output :

CustomerThread--2executing with Customer id :1

CustomerThread--4executing with Customer id :3

CustomerThread--3executing with Customer id :2

CustomerThread--1executing with Customer id :1

In above program for every thread a separate customer id will be maintain by ThreadLocal object.

#### ThreadLocal vs Inheritance

Parent threadlocal variable by default not available to the child thread.

If you want to make parent threads threadlocal value available to the child thread then we should go for **inheritable threadlocal class**.

By Default child thread value is exactly same as parent thread value but we can customize value for thread by overriding child value method.

#### Constructor

**InheritableThreadLocal tl=new InheritableThreadLocal();**

#### method

1. Inheritable threadlocal is the child class of threalocal and hence all method present in threadlocal by default available to inheritable threadlocal , in addition to this method it contain only one method.

**public object childValue(object parentValue)**

**Program**

**package** oops.thread.threadlocal;

**class** ParentThread **extends** Thread{

**publicstatic**InheritableThreadLocal*tl*=**new**InheritableThreadLocal()

{

**public** Object childValue(Object p){

**return**"cc";

}

};

**publicvoid** run(){

*tl*.set("pp");

System.***out***.println("Parent Thread Value .."+*tl*.get());

ChildThread ct=**new** ChildThread();

ct.start();

}

}

**class** ChildThread **extends** Thread{

**publicvoid** run(){

System.***out***.println("childThread value .."+ParentThread.*tl*.get());

}

}

**publicclass** ThreadLocalDemo3 {

**publicstaticvoid** main(String[] args) {

ParentThread pt=**new** ParentThread();

pt.start();

}

}

**In the above program if we replace inheritable threadlocal with threadlocal if we are not overriding child value method then output is**

Parent Thread Value .. pp

Childthread value .. null

In the program if we are maintaining inheritable threadlocal and if we are not overriding child value method

Parent Thread Value .. pp

Childthread value .. pp

## Java Multithreading Interview Questions and Answers

#### What is multithreading?

**Multithreading in java** is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Its main advantage is:

* Threads share the same address space.
* Thread is lightweight.
* Cost of communication between process is low.

#### What are the benefits of multi-threaded programming?

* More efficient CPU use
* Better system reliability
* Improved performance on multiprocessor computers

#### What is thread?

A thread is a lightweight sub process is a separate path of execution. It is called separate path of execution because each thread runs in a separate stack frame.

### *What is the difference between Process and Thread?*

|  |  |
| --- | --- |
| **Process** | **Thread** |
| An executing instance of a program is called a process | A thread is a subset of the process. |
| Heavy weight | Light weight |
| Process can contain threads | But threads can’t contain process |
| processes run in separate memory spaces. | Thread run in a shared memory space |
| communication between process requires some Time | Where as thread requires less time. |

### *What does join() method?*

If a thread wants to wait until completing some other thread, then we should go for join method.

### *What is difference between wait() and sleep() method?*

|  |  |
| --- | --- |
| Wait | Sleep |
| 1) The **wait**() method is defined in Object class. | The **sleep**() method is defined in Thread class. |
| 2.**wait**() method releases the lock. | The **sleep**() method doesn't releases the lock. |

### *Is it possible to start a thread twice?*

No, there is no possibility to start a thread twice. If we do, it throws an exception.

### *Can we call the run() method instead of start()?*

yes, but it will not work as a thread rather it will work as a normal object so there will not be context-switching between the threads.

### *What about the daemon threads?*

The daemon threads are basically the low priority threads that provides the background support to the user threads. It provides services to the user threads.

### *Can we make the user thread as daemon thread if thread is started?*

No, if you do so, it will throw **IllegalThreadStateException**

### *What is shutdown hook?*

The shutdown hook is basically a thread i.e. invoked implicitely before JVM shuts down. So we can use it perform clean up resource.

### *When should we interrupt a thread?*

A thread can interrupt a sleeping thread or waiting thread by using interrupt of thread class

### *What is synchronization?*

* synchronizer is a modifier applicable only for method and block but not for classes and variable
* **If a multiple thread trying to operate simultaneously at the same java object then may be a chance of data inconsistency problem. To overcome this problem we should go for synchronizer keywords.**
* If a method or block declared as synchronizer 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 resolve.

### *What is the purpose of Synchronized block?*

* If very few lines of code required synchronization then it is not recommended declaring entire method synchronized 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 thread and improves performance of the system**.

### *Can Java object be locked down for exclusive use by a given thread?*

Yes. You can lock an object by putting it in a "synchronized" block. The locked object is inaccessible to any thread other than the one that explicitly claimed it.

### *What is static synchronization?*

If you make any static method as synchronized, the lock will be on the class not on object.

### *What is the difference between notify() and notifyAll()?*

* We can use **notify** method to the give the notification for only one waiting thread if multiple threads are waiting then only one thread will be notified and the remaining thread have to wait for further notifications.
* We can use **notifyAll** to give the notification for all waiting threads of a particular object even though multiple threads notified but execution will be perform one by one because thread requires lock and only one lock is available.

### *What is deadlock?*

Deadlock is a situation when two threads are waiting on each other to release a resource.

### *What is difference between user Thread and daemon Thread?*

|  |  |
| --- | --- |
| **User Thread** | **Daemon Thread** |
| User threads are high priority threads which always run in foreground | Whereas Daemon threads are low priority threads which always run in background |
| User threads are designed to do some specific task | whereas daemon threads are used to perform some supporting tasks |
| User threads are created by the **application** (user) to perform some specific task | Whereas daemon threads are mostly created by the **JVM** to perform some background tasks like garbage collection |
| JVM will wait for user threads to finish their tasks | JVM will not wait for daemon threads to finish their tasks. It will exit as soon as all user threads finish their tasks. |
| JVM will not force the user threads to terminate It will wait for user threads to terminate themselves | JVM will force the daemon threads to terminate if all the user threads have finished their task |

### *What is Thread Scheduler and Time Slicing?Difference b/w preemptive scheduling and time slicing*

Thread scheduler in java is the part of the JVM that decides which thread should run.

**Preemptive scheduling:** -the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence.  
**Time slicing**:-a task executes for a predefined slice of time and then reenters the pool of ready tasks.

### *What is context-switching in multi-threading?*

**Context switch** is the process of storing the state of a [process](https://en.wikipedia.org/wiki/Process_(computing)) or of a [thread](https://en.wikipedia.org/wiki/Thread_(computing)), so that it can be restored later and [execution](https://en.wikipedia.org/wiki/Execution_(computing)) resumed from the same point later. This allows multiple processes to share a single [CPU](https://en.wikipedia.org/wiki/CPU), and is an essential feature of a [multitasking operating system](https://en.wikipedia.org/wiki/Multitasking_operating_system).

Context switch is the switching of the CPU from one process/thread to another process/thread.

### *Why thread communication methods wait(), notify() and notifyAll() are in Object class?*

**Wait ,notify, notifyAll** method present in object class but not in thread class because thread can call these methods on any java object.

In Java every Object has a monitor and wait, notify methods are used to wait for the Object monitor or to notify other threads that Object monitor is free now. There is no monitor on threads in java and synchronization can be used with any Object.

### 23.*Why wait(), notify() and notifyAll() methods have to be called from synchronized method or block?*

When a Thread calls wait() on any Object, it must have the monitor on the Object that it will leave and goes in wait state until any other thread call notify() on this Object. Similarly when a thread calls notify() on any Object, it leaves the monitor on the Object and other waiting threads can get the monitor on the Object. Since all these methods require Thread to have the Object monitor, that can be achieved only by synchronization, they need to be called from synchronized method or block.

### *Why Thread sleep() and yield() methods are static?*

Thread **sleep() and yield()** methods work on the currently executing thread. So there is no point in invoking these methods on some other threads that are in wait state.

### *How can we achieve thread safety in Java?*

There are several ways to achieve [thread safety in java](https://www.journaldev.com/1061/thread-safety-in-java) – **synchronization, atomic concurrent classes, implementing concurrent Lock interface**, using **volatile keyword**, using **immutable classes** and Thread safe classes.

### *What is Thread Group?*

Based on functionality we can group thread into a single unit which is nothing but ThreadGroup contains a group of threads.

ThreadGroup is a class which was intended to provide information about a thread group.

ThreadGroup API is weak and it doesn’t have any functionality that is not provided by Thread.

Two of the major feature it had are to get the list of active threads in a thread group and to set

the uncaught exception handler for the thread. But Java 1.5 has addedsetUncaughtExceptionHandler(UncaughtExceptionHandler eh)

 method using which we can add uncaught exception handler to the thread.

So ThreadGroup is obsolete and hence not advised to use anymore.

Java provides a convenient way to group multiple threads in a single object.

In such way, we can suspend, resume or interrupt group of threads by a single method call.

### *What is Java Thread Dump, How can we get Java Thread dump of a Program?*

Thread dump is list of all the threads active in the JVM, thread dumps are very helpful in analyzing bottlenecks in the application and analyzing deadlock situations. There are many ways using which we can generate Thread dump – Using Profiler, Kill -3 command, jstack tool etc.

### *What is Thread Pool?*

A thread pool manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed.

### *What is volatile keyword in Java*

Reading and writing of volatile variables causes the variable to be read or written to main memory.

The Java **volatile** keyword is used to mark a Java variable as "**being stored in main memory**". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

### *What is ThreadLocal variables in Java.*

The **ThreadLocal** class in Java enables you to create variables that can only be read and written by the same thread.

Thus, even if two threads are executing the same code, and the code has a reference to a ThreadLocal variable,

then the two threads cannot see each other's ThreadLocal variables.

ThreadLocal gives us the ability to store data individually for the current thread.

The TheadLocal construct allows us to store data that will be accessible only by a specific thread.

**ThreadLocal** in Java had been introduced on JDK 1.2 but was later generified in JDK 1.5 to introduce type safety on ThreadLocal variable.

**ThreadLocal can be associated with Thread scope**, all the code which is executed by Thread has access to ThreadLocal variables but two thread can not see each others ThreadLocal variable.

Each thread holds an exclusive copy of ThreadLocal variable which becomes eligible to Garbage collection after thread finished or died, normally or due to any Exception, Given those ThreadLocal variable doesn't have any other live references.

**ThreadLocal** variables in Java are generally private static fields in Classes and maintain its state inside Thread.

### *How to create daemon thread in java*

You can make any java thread as daemon thread. Daemon threads acts like service providers for other threads running in the same process.

Daemon threads will be terminated by the JVM when there are none of the other threads running, it includes main thread of execution as well.

To specify that a thread is a daemon thread, call the **setDaemon** method with the argument true.

To determine if a thread is a daemon thread, use the accessor method isDaemon.

class DaemonThreadDemo1 extends Thread {

     public DaemonThreadDemo1(){

      setDaemon(true);

     }

     public void run(){

      System.out.println("is this thread daemon ==>"+isDaemon());

     }

      public static void main(String[] args) {

      DaemonThreadDemo1 d=new DaemonThreadDemo1();

      d.start();

}

}

### *What is Lock*

This is the base interface for Lock API. It provides all the features of synchronized keyword with additional ways to create

different Conditions for locking, providing timeout for thread to wait for lock. Some of the important methods are **lock()** to acquire the lock,

**unlock()** to release the lock, **tryLock()** to wait for lock for a certain period of time, newCondition() to create the Condition etc.

### *Different between Lock vs Synchronized*

1. Java Lock API provides more visibility and options for locking, unlike synchronized where a thread might end up waiting indefinitely for the lock, we can use tryLock() to make sure thread waits for specific time only.
2. Synchronization code is much cleaner and easy to maintain whereas with Lock we are forced to have try-finally block to make sure Lock is released even if some exception is thrown between lock() and unlock() method calls.
3. synchronization blocks or methods can cover only one method whereas we can acquire the lock in one method and release it in another method with Lock API.
4. synchronized keyword doesn’t provide fairness whereas we can set fairness to true while creating ReentrantLock object so that longest waiting thread gets the lock first.
5. We can create different conditions for Lock and different thread can await() for different conditions.

The Lock has additional method such as tryLock which is not an option for a synchronized block.

### *Program for Producer and Consumer*

import java.util.Vector;

class Producer extends Thread {

    static final int MAXQUEUE = 5;

    private Vector messages = new Vector();

    public void run() {

        try {

            while ( true ) {

                putMessage();

                sleep( 1000 );

            }

        }

        catch( InterruptedException e ) { }

    }

    private synchronized void putMessage()

        throws InterruptedException

{

        while ( messages.size() == MAXQUEUE )

            wait();

        messages.addElement( new java.util.Date().toString() );

        notify();

    }

    // Called by Consumer

    public synchronized String getMessage()

        throws InterruptedException

{

        notify();

        while ( messages.size() == 0 )

            wait();

        String message = (String)messages.firstElement();

        messages.removeElement( message );

        return message;

    }

}

class Consumer extends Thread {

    Producer producer;

    Consumer(Producer p) {

        producer = p;

    }

    public void run() {

        try {

            while ( true ) {

                String message = producer.getMessage();

                System.out.println("Got message: " + message);

                sleep( 2000 );

            }

        }

        catch( InterruptedException e ) { }

    }

    public static void main(String args[]) {

        Producer producer = new Producer();

        producer.start();

        new Consumer( producer ).start();

    }

}

[***https://dzone.com/articles/threads-top-80-interview***](https://dzone.com/articles/threads-top-80-interview)

[***https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers***](https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers)