MultiThreading:

***Introduction:***

Executing multiple tasks simultaneously is called multi-tasking. This total multi-tasking is divided into two parts.

1. Process based
2. Thread based

***Process based:*** Executing multiple tasks simultaneously as a separate independent process is called process based multi-tasking. This type of multi-tasking is related to OS level and not programmatic level.

***Thread based:*** This type of multi-tasking is related to the programmatic level. Multiple threads execute with in one program. Threads are independent but they share common memory area (Address space). In this only one program is their but contains many independent parts.

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

1. It is used to develop multimedia graphics.
2. To develop animations.
3. To develop video games.
4. To develop web servers, application servers.

Java provides inbuilt support for multi-threading with the help of several rich API like Thread, Runnable, ThreadGroup.

Thread is an independent job, separate flow of execution,

There are two ways to define a thread:

1. By extending Thread class.
2. By implementing Runnable interface.

Every java program contains by-default one thread i.e, main thread and several daemon threads.

**public** **class** Thread Demo {

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

***//main thread creates child thread myThread***

MyThread myThread = **new** MyThread(); ***//Thread instantiation //only one thread***

myThread.start(); ***// Now there are two threads***

}

}

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

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

System.***out***.println("Thre

ad is started");

}

}

***Thread scheduler:*** it is a part of JVM. If there are multiple threads, the order in which they must execute is controlled by Thread scheduler. But there is not any guarantee that order is same every time, it varies from JVM to JVM.

***Q. What is the importance of start() method.?***

Several internal activities are to be performed that is done by start() method.

Register the thread with the Thread Scheduler.

Invoke run() method.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Q. Can we overload run() method.?***

Yes, but JVM only call no-argument run() method. The other overloaded run() method is to be called explicitly.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Q. If we do not override run() method?***

The program is valid but the Thread class run() method will be called which has empty implementation, hence we won’t get any output.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Q. If we define start() (i.e, override start()) method in the child thread class, what will happen?***

The overridden start() method will be executed just like a normal start() method and the new thread won’t be created.

**public** **class** Thread Demo {

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

***//main thread creates child thread myThread***

MyThread myThread = **new** MyThread(); ***//Thread instantiation //only one thread***

myThread.start(); ***// Now there are two threads***

}

}

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

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

**super**.start(); ***//this will create new thread, otherwise no thread will be created in this case***

System.***out***.println("override start method is executed");

}

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

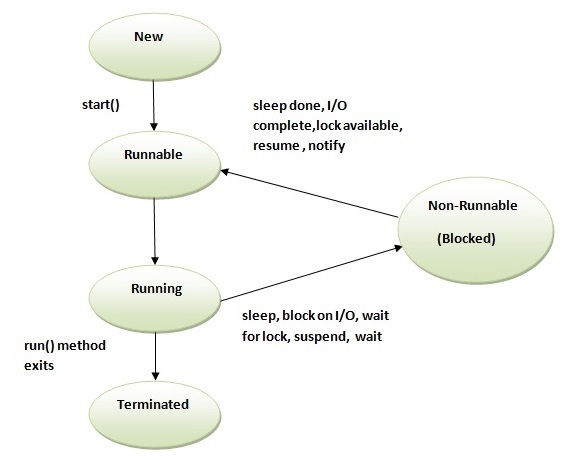
System.***out***.println("Thread is started");

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Thread Life Cycle:**



***Q. Can we start start() method twice.?***

No, we will get IllegalThreadStateException.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Implementing Runnable is recommended than extending Thread as by extending Thread we cannot inherit any other class, so inheritance benefit is missing and also performance wise is also good in case of Runnable interface.

***Thread class constructors:***

Thread t1 = new Thread(); Thread t2 = new Thread(Runnable r); Thread t3 = new Thread(String name); Thread t4 = new Thread(Runnable r, String name); Thread t5 = new Thread(ThreadGroup g, String name); Thread t6 = new Thread(ThreadGroup, Runnable r); Thread t7 = new Thread(ThreadGroup, Runnable r, String name); Thread t8 = new Thread(ThreadGroup, Runnable r, String name, long stackSize);

***Name of thread:***The name of the thread is provided by the JVM or the customized name must be provided.

**public** **class** ThreadDemo **extends** Thread {

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

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

Thread.*currentThread*().setName("MAIN:p");

System.***out***.println(Thread.*currentThread*().getName()); // MAIN:p

ThreadDemo t1 = **new** ThreadDemo();

System.***out***.println(t1.getName()); // Thread-0

t1.setName("Tanuj Thread");

System.***out***.println(t1.getName()); // Tanuj Thread

System.***out***.println(10 / 0); // Exception in thread "MAIN:p"

}

}

***Priority of Thread:*** The priority of the thread is provided by the JVM or the customized priority must be provided.

Priority or the thread is from 1 to 10.

Thread.MIN\_PRIORITY = 1

Thread.NORM\_PRIORITY = 5

Thread.MAX\_PRIORITY = 10

If the two threads have the same priority, then the order of their execution is not preserved.

If the parent has the thread priority 10 then the child will also have the priority of 10. It mean that the default priority is inherited from the parent to the child.

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

}

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

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

Thread.*currentThread*().setPriority(7);

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

MyThread myThread = **new** MyThread();

System.***out***.println(myThread.getPriority());

}}

In the above we have change the priority of the main class to 7. As main thread is the parent thread for executing Mythread thread, so the priority of the MyThread thread will also get changed to 7.

NOTE: parent class of MyThread is Thread but parent Thread for the MyThread is main Thread

Some platform won’t provide proper support for thread priority.

***We can prevent the thread execution by using:***

1. yield(): static method
2. join(): non-static method
3. sleep(): static method
4. **yield()**

public static native void yield()

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

If multiple threads are waiting with the same priority we can’t expect and it depends on thread scheduler.

The thread which is yielded the it will get the chance again depends on thread scheduler.

Some platform won’t provide proper support for yield method.

1. **Join:**

public final void join() throws InterruptedException

public final void join(long ms) throws InterruptedException

public final void join(long ms, int nano) throws InterruptedException

All methods are final.

1. **sleep()**

public static **native** void sleep(long milliseconds) throws InterruptedException

public static void sleep(long milliseconds, int nanoseconds) throws InterruptedException

one method is native and other is not.

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

**static** Thread *mainThread*;

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

**try** {

*mainThread*.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**for** (**int** i = 0; i < 2; i++)

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

}

}

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

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

MyThread.*mainThread* = Thread.*currentThread*(); // Assigning main thread to *mainThread*

// Intantiating thread

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

t.start();

***// t.join() // DEADLOCK***

// main thread performing action

Thread.*sleep*(2000);

**for** (**int** i = 0; i < 2; i++) {

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

}

}

}

***OUTPUT:***

Main Thread

Main Thread

child

child

As we have called Thread.sleep(2000), so output should have start from child. As we have called join() method, main thread will complete it execution first no matter what. If the child thread also calls join method then there will be deadlock.

A Thread can interrupt a sleeping thread or waiting thread with the help of the method named interrupt(). If there is no sleep() method then interrupt() method will not get wasted because JVM handles it i.e, if and only if JVM finds sleep() , wait() methods, it calls interrupt() method.

***Synchronization:***

Synchronized is a modifier that is used for the synchronization.

It is applicable for only methods and blocks.

If multiple threads are trying to operate simultaneously on the same java object then data inconsistency arises. To overcome this problem synchronization is done.

Internally synchronization concept is implemented by lock. Every object in java has a unique lock. Whenever we are using synchronized keyword, lock concept comes into the picture. If the thread wants to execute synchronized method on the given object, first it has to get the lock of that object. Once it gets that intrinsic lock, it is allowed to execute that method. Once method execution completes, that lock is released by JVM.

If multiple threads are operating on same java Object then there may be a chance of data inconsistency problem. This is called race condition. We can overcome this problem with the help of synchronized keyword.

**class** MyThread {

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

**while**(**true**)

{

}

}

**public** **synchronized** **void** m2() {

System.***out***.println("m2 method");

}

**public** **void** m3() {

System.***out***.println("m3 method");

}

}

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

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

MyThread m = **new** MyThread();

Thread t1 = **new** Thread(() -> {

m.m1();

});

Thread t2 = **new** Thread(() -> {

***// Either m1(), m2(), m3()***

m.m3();

m.m1();

m.m2();

});

t1.start();

t2.start();

}

}

As there are two threads, and method m1() and m2() are synchronized. m1() method contains infinite loop i.e, it won’t get complete. Here, the thread t1 calls the method m1() so t1 acquires the intrinsic lock of MyThread class Object which won’t get release as method m1() cannot get complete. Meanwhile, there is another thread t2 which can do the following:

1. **It can try to execute m1():** It is not possible because for that it must acquire the lock of MyThread object, that is acquires by the thread t1 and won’t release it.
2. **It can try to execute m2() method which is an another synchronized method:** Again it can’t happen as synchronization is based on Object lock, and in this case the lock is acquired by the thread t1.
3. **It can try and execute non-synchronized m3() method:** It can happen.

If the method is static synchronized then synchronization is based on class level lock. Once thread got class level lock then it is allowed to execute any static synchronized method. Once method execution completes automatically threads releases the lock.

**Inter-Thread Communication**

Two threads can communicate with each other with wait(), notify() and notifyAll(). The thread which is expecting updation is responsible to call wait() method then immediately the thread will enter into the waiting state. The thread which is responsible updation, after performing updation is responsible to call notify() method then waiting thread will get the notification and continue its execution with those updated items.

wait(), notify() and notifyAll() are to be called within synchronized area.

***Q. Why the above methods are defined in Object class and not Thread class?***

Because thread can call these methods on any java objects.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

If a thread call notify() method on any object, it releases the lock of that object immediately.

Except wait(), notify(), notifyAll() there is no other method that releases the lock.

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

**public** **int** sum = 0;

@Override

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

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

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

sum += i;

}

**this**.notify();

}

}

}

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

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

MyThread m = **new** MyThread();

m.start();

**synchronized** (m) {

m.wait();

}

System.***out***.println(m.sum);

}

}

***Producer Consumer Problem***

class Producer class Consumer

{ {

produce() consume()

{ {  
synchronized(q) synchronized(q)

{ {  
produce items to the queue if(q is empty)

q.notify() q.wait()

}}} else

Consume items from q}}}

***Daemon Thread:***

The threads which are executing the background are called daemon threads. Example: Garbage collector, signal dispatcher, Attach Listener.

**Advantage of daemon thread**.

To provide support for non-daemon threads(main thread). For example if main thread run with low memory then JVM calls 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.

Daemon runs with the low priority.

At the beginning main thread and gc thread are there, but gc thread have low priority hence main thread runs. If main run low memory then the priority of gc thread is increased.

# Daemon thread can run as low priority thread as well as high priority thread.

We can check the daemon nature of thread by using isDaemon() method of Thread class.

Public Boolean isDaemon();

We can change the nature of the thread by using setDaemon().

But changing Daemon nature is possible before starting of a Thread only. After starting the thread if we are changing the Daemon nature we get IllegalThreadStateException.

By default, main thread is always non-daemon and for all remaining thread daemon nature will be inherited from parent to child i.e, if parent thread is daemon then automatically child thread is daemon and vice versa.

Whenever last non-Daemon Thread terminates, all daemon threads will be terminates irrespective of their positions.

***Q. Can we change the nature of main thread to daemon nature?***

It is impossible as it is already started by the JVM in the beginning.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

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

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

Thread.*currentThread*().setDaemon(**true**); **// java.lang.IllegalThreadStateException**

Thread t = **new** Thread(() -> {});

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

t.setDaemon(**true**);

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

}

}

***Green thread:***

Multithreading concept is implemented by using two models:

1. Green thread model
2. Native os model

Green thread is managed completely by only JVM with out taking underlying OS support.

Very few Os supports Green thread model. Therefore, it is deprecated.

***DeadLock:***

Two threads are waiting for each other forever. Such type of infinite waiting is called deadlock.

Synchronized keyword is responsible for this type of situation. Order of the lock is also responsible for the deadlock.

Synchronized keyword is the only reason for deadlock situation. Hence while using synchronized keyword we have to take special care. Several prevention techniques are responsible but no resolution technique is available.

**class** A {

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

b.last(); // trying to call the method of B

}

**// The above d1(B b) method is synchronized so it will hold the intrinsic lock of A**

**// In this method it is calling last() method of class B which is also synchronized and having intrinsic lock of B**

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

System.***out***.println("A");

}

}

**class** B {

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

a.last(); // trying to call the method of A

}

**// The above d2(A a) method is synchronized so it will hold the intrinsic lock of B**

**// In this method it is calling last() method of class A which is also synchronized and having intrinsic lock of A**

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

System.***out***.println("B");

}

}

**public** **class** Durga {

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

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

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

Thread t1 = **new** Thread(() -> {

a.d1(b);

});

Thread t2 = **new** Thread(() -> {

b.d2(a);

});

t1.start();

t2.start();

}

}

**The above is the example of deadlock.**

**Long waiting of a thread where waiting doesn’t end is called DeadLock.**

**Long waiting of a thread where waiting must end is called starvation.**

**# EXAMPLE 2**

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

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

***Thread.currentThread().join(); //DEADLOCK ARISES***

}

}

**If a thread calls on the same thread itself, this is something like deadlock.**

**Above is also kind of deadlock as the main thread will wait infinitely waiting of the completion of itself.**

**#EXAMPLE:3**

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

**static Thread *mainThread*;**

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

**try** {

*mainThread*.join(); **// main thread must complete fully**

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

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

}

}

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

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

MyThread.*mainThread* = Thread.*currentThread*(); // Assigning main thread to *mainThread*

// Intantiating thread

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

t.start();

t.join(); **// child thread must complete fully**

// main thread performing action

Thread.*sleep*(2000);

**for** (**int** i = 0; i < 4; i++) {

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

}

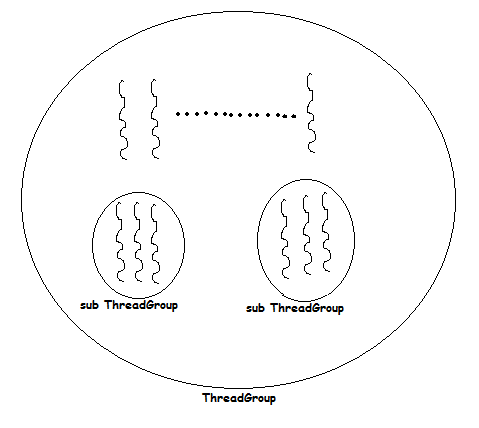
}

}

**In the above main thread as well as the child thread are calling join() method, the condition of deadlock arises.**

***ThreadGroup:***

Based on the functionality we can group threads into a single unit which is nothing but ThreadGroup i.e, ThreadGroup contains a group of threads. In addition to thread, ThreadGroup can also contains sub thread groups.



Every ThreadGroup is a child of System. So every program has one main thread and one main ThreadGroup.

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

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

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

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

}}

System Group contains several System level threads like

1. Finalize
2. ReferenceHandler
3. AttachListener

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

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

ThreadGroup t1 = **new** ThreadGroup("First thread group");

System.***out***.println(t1.getParent().getName()); **// main**

ThreadGroup t2 = **new** ThreadGroup(t1, "Second thread group");

System.***out***.println(t2.getParent().getName()); **// First thread group**

}

}

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

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

ThreadGroup g = **new** ThreadGroup("First thread group");

Thread t1 = **new** Thread(g, "First Thread");

Thread t2 = **new** Thread(g, "Second Thread");

g.setMaxPriority(3);

Thread t3 = **new** Thread(g, "Third thread");

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

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

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

}

}

To copy all active threads of Thread group into provided Thread array. Sub Thread group will also be considered.

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

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

ThreadGroup parent = **new** ThreadGroup("Parent Group");

ThreadGroup child = **new** ThreadGroup(parent, "Second Thread");

Thread t1 = **new** Thread(parent, () -> {

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

}, "childThread1");

Thread t2 = **new** Thread(parent, () -> {

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

}, "childThread2");

t1.start();

t2.start();

System.***out***.println("active count is " + parent.activeCount()); // 2

System.***out***.println("active group count is " + parent.activeGroupCount()); // 1

parent.list();

}

}

**Q. Write a program to display all the active threads of System ThreadGroup.?**

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

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

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

//gets the System ThreadGroup

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

//creates Thread array with the size of a all active threads.

system.enumerate(threads); // All active threads are copied into threads array

**for** (Thread thread : threads) {

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

}

}

}

Output:

Reference Handler true

Finalizer true

Signal Dispatcher true

Attach Listener true

Monitor Ctrl-Break:true

main false

Here, enumerate() method will enumerate all the thread in system group and copy them into threads array.

**java.util.Concurrent**

In this package the problem of DeadLock is solved.

The problems with the traditional synchronized 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 a lock, so a thread will wait until getting a lock which may cause deadlock and performance problem.
3. If a thread releases a lock then which waiting thread will get that lock, we are not having any control on this.
4. There is no API to list out all waiting threads for a lock.
5. Synchronized compulsory we have to use either in method level or within a method and it is not possible to use across multiple methods.

To overcome this problem sun people introduced **java.util.Concurrent.**

Lock:

Lock object is similar to implicit lock acquired by a thread to execute synchronized method or synchronized block. It provides more extensive operation than traditional implicit locks.

Important method of lock interface:

void lock(): we can use this method to acquire a lock. If lock is already available

ThreadLocal:

It provides thread local variable. It maintains values per thread basis.

Each local threadLocal object maintains a separate value like usedId, transactionId etc for each thread that accesses that object.

Thread can access its local value, manipulate its value and even can remove its value. In every part of the code which is executed by the Thread, we can access its local variable. Example: consider a servlet which invokes some business methods.

We have a requirement to generate unique transaction Id for each and every request and we have to pass this transactionId to the business methods. For this requirement we can use thread local to maintain a separate transaction Id i.e, for every thread.

ThreadLocal class is introduced in 1.3v and enhanced in 1.5 version. ThreadLocal can we associated with thread scope. Total code that is executed by the thread has access to the corresponding ThreadLoacal variable. A thread can access its own local variable and can’t access other threads local variables.

Once thread enter into dead state all its local variable are by default available for garbage collection.

Constructors:

ThreadLocal tl = new ThreadLocal();

Methods:

Object get() : return the value of threadLocal variable associated with current thread.

Object initialValue(): return the initial value of ThreadLocal variable associated with current thread. The default implementation of this method return null. To customize our own initial value we have to override this method.

Void set(Object newValue): to set a new value.

Void remove(): remove the value of threadLocal variable associated with current thread. This method is added in 1.5v. after removal if we are trying to access, it will be reinitialized once again by invoking its initialValue() method.

ThreadLocal<String> tl = **new** ThreadLocal<>();

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

tl.set("tanuj");

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

tl.remove();

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

**If you don’t want null as the initial value we have to override the initialValue() method.**

ThreadLocal<String> tl = new ThreadLocal<String>() {

public String initialValue() {

return "abc";

}

};

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

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

**static** Integer *cust* = 0;

**private** **static** ThreadLocal<Integer> *tl* = **new** ThreadLocal<Integer>() {

**public** Integer initialValue() {

**return** ++*cust*;

}

};

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

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

Thread t1 = **new** Thread(() -> {

l.lock();

System.***out***.println(Thread.*currentThread*().getName() + " " + *tl*.get());

l.unlock();

}, "first");

Thread t2 = **new** Thread(() -> {

l.lock();

System.***out***.println(Thread.*currentThread*().getName() + " " + *tl*.get());

l.unlock();

}, "second");

Thread t3 = **new** Thread(() -> {

l.lock();

System.***out***.println(Thread.*currentThread*().getName() + " " + *tl*.get());

l.unlock();

}, "third");

Thread t4 = **new** Thread(() -> {

l.lock();

System.***out***.println(Thread.*currentThread*().getName() + " " + *tl*.get());

l.unlock();

}, "fourth");

t1.start();

t2.start();

t3.start();

t4.start();

}

}

Output:

first 1

second 2

third 3

fourth 4

ThreadLocal:

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

**static** ThreadLocal<String> *tl* = **new** ThreadLocal<>();

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

*tl*.set("pt");

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

Child c = **new** Child();

c.start();

}

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

MyThread m = **new** MyThread();

m.start();

}

}

**class** Child **extends** Thread {

@Override

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

System.***out***.println("child thread " + MyThread.*tl*.get());

}

}

parent value pt

child thread null

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

**static** InheritableThreadLocal<String> *tl* = **new** InheritableThreadLocal<>();

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

*tl*.set("pt");

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

Child c = **new** Child();

c.start();

}

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

MyThread m = **new** MyThread();

m.start();

}

}

**class** Child **extends** Thread {

@Override

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

System.***out***.println("child thread " + MyThread.*tl*.get());

}

}

parent value pt

child thread pt

Multi processing is related to several independent program running at the same time but multi threading is related to the same program having independent processes running together at the same time.

Multi threading is used to develop animations, video games, etc.

**Lock Interface:**

Lock object is similar to implicit lock acquired by a thread to execute synchronized method or synchronized block. It provides more extensive operations than traditional implicit locks.

Important method of lock interface

1. ***Void lock():***

* we use this method to acquire a lock. If a lock is available then immediately current thread will get a lock. If a lock is not already available then it will wait until getting the lock.

1. **Boolean tryLock():**

* to acquire the lock without waiting. If the lock is available then the thread acquires the lock and return true. If lock is not available then this method returns false and can continue its execution without waiting. In this case thread never be entered into waiting state.

If(l.tryLock())

{  
perform safe operations}

else

{  
perform alternate operations}

1. ***boolean tryLock(long time, TimeUnit unit):***

* If the lock is not available then the thread will wait till the specified amount of time.
* TimeUnit is an enum which is present in java.util.concurrent.TimeUnit.

Enum TimeUnit{

NANOSECONDS,

MICROSECONDS,

MILLISECONDS,

SECONDS,

MINUTES,

HOURS,

DAYS,

}

If(l.tryLock(1000, TimeUnit.NANSECONDS))

{

Perform safe operations{

Else

{

Perform alternate operations

}

4. ***void lock interruptibly():*** Acquires a lock if it available immediately. If a lock is not available it will wait. While waiting if it is interrupted it will not wait.

4. ***void unlock():*** to call this method current thread should be owner of the lock otherwise we will get runtime exception saying IllegalMonitorStateException.

**Reentrant lock:**

It is an implementation class of Lock interface and it is the direct child class of Object.

Reentrant means a thread can acquire same lock multiple times without any issue. Internally reentrant lock implements threads personal count whenever we call lock method and decrements unlock method and lock will be released whenever the count reaches to zero.

Constructors

1. ReentrantLock l = new ReentrantLock();
2. ReentrantLock l = new ReentrantLock(Boolean fairness);

* Creates reentrant lock with the given fairness policy. If the fairness is true then the longest waiting thread will get the chance if available. So it follows FCFS. If fairness is false which waiting thread will get a chance we can’t guess. Default is false.

Important methods of Reentrant lock.

1. int getHoldCount()
2. Boolean isHeldByCurrentThreead()
3. Int getQueueLength()
4. Collection getQueuedThreads()
5. Boolean hasQueuedThreads()
6. Boolean isLocked()
7. Boolean isFair()
8. Thread getOwner();

**public** **class** RentrantLock {

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

ReentrantLock lock = **new** ReentrantLock();

lock.lock(); // first lock

lock.lock(); // second lock

System.***out***.println(lock.isLocked()); // true

System.***out***.println(lock.isHeldByCurrentThread()); // true

System.***out***.println(lock.getHoldCount()); // 2

System.***out***.println(lock.getQueueLength()); // 0

lock.unlock();

System.***out***.println(lock.isLocked()); // true

System.***out***.println(lock.isHeldByCurrentThread()); // true

System.***out***.println(lock.getHoldCount()); // 1

System.***out***.println(lock.getQueueLength()); // 0

lock.unlock();

System.***out***.println(lock.isLocked()); // false

System.***out***.println(lock.isFair()); // false

}

}

**class** Display {

ReentrantLock lock = **new** ReentrantLock();

**public** **void** display(String s) {

lock.lock();

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

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

}

lock.unlock();

}

}

**public** **class** RentrantLock {

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

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

Thread t1 = **new** Thread(() -> {

d.display("tanuj");

});

Thread t2 = **new** Thread(() -> {

d.display("vikhyat");

});

t1.start();

t2.start();

}

}

**class** Display {

ReentrantLock lock = **new** ReentrantLock();

**public** **void** display(String s) {

**if** (lock.tryLock()) {

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

lock.unlock();

} **else**

System.***out***.println("For " + Thread.*currentThread*().getName() + "lock is not available");

}

}

**public** **class** RentrantLock {

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

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

Thread t1 = **new** Thread(() -> {

d.display("tanuj");

}, "first thread");

Thread t2 = **new** Thread(() -> {

d.display("vikhyat");

}, "second thread");

t1.start();

t2.start();}}

**class** Display {

ReentrantLock lock = **new** ReentrantLock();

**public** **void** display(String s) **throws** InterruptedException {

**do** {

**if** (lock.tryLock(1000, TimeUnit.***MILLISECONDS***)) {

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

Thread.*sleep*(5000);

lock.unlock();

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

**break**;

} **else**

System.***out***.println("For " + Thread.*currentThread*().getName() + "lock is not available");

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

}

}

**public** **class** RentrantLock {

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

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

Thread t1 = **new** Thread(() -> {

**try** {

d.display("tanuj");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}, "first thread");

Thread t2 = **new** Thread(() -> {

**try** {

d.display("vikhyat");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}, "second thread");

t1.start();

Thread.*sleep*(1);

t2.start();

}

}

**ThreadPool Executor framework:**

Creating a new thread for every job creates a memory problem. To overcome this we should go for thread pool. Thread pool is the pool of already created threads. Java 1.5 introduced thread pool 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*(5);**

**class** Display **implements** Runnable {

**private** String name;

**public** Display(String name) {

**this**.name = name;

}

@Override

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

System.***out***.println(Thread.*currentThread*().getName() + "....." + name);

}

}

**public** **class** RentrantLock {

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

ExecutorService service = Executors.*newFixedThreadPool*(5);

Display[] d = { **new** Display("tanuj"),

**new** Display("vikhyat"),

**new** Display("wani"),

**new** Display("aviral"),

**new** Display("srishti") };

**for** (Display display : d) {

service.submit(display);

}

service.shutdown();

}

}

#NOTE: While designing web servers and application servers we use ThreadPool concept.

***Callable and Future Interface:***

Introduced in 1.5v and present in java.concurrent package while Runnable is present in java.lang package.

If we use Runnable interface then we have void run() method that don’t return anything. In some cases we want to return something then in that case we have to use Callable interface instead of Runnable interface.

Callable interface contains call() method.

public Object call() //throws Exception we can use throws with it.

When the value is returned then Future interface to going to collect it.

**class** Display **implements** Callable<Integer> {

**private** Integer number;

**public** Display(Integer number) {

**this**.number = number;

}

@Override

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

**return** number;

}

}

**public** **class** MyClass {

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

ExecutorService service = Executors.*newFixedThreadPool*(5);

Display[] d = { **new** Display(1), **new** Display(2), **new** Display(3), **new** Display(4), **new** Display(5) };

**for** (Display display : d) {

Future<Integer> f = service.submit(display);

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

}

service.shutdown();

}

}

**Producer\_Consumer**

**Producer consumer with Blocking queue**

class ProducerConsumer {  
  
 public static void main(String[] args) {  
  
 BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(1);  
  
 Thread producer = new Thread(() -> {  
 int i = 1;  
 while (i != 0) {  
 try {  
 Thread.*sleep*(2000);  
 queue.put(i);  
 System.*out*.println("Produced:: " + i);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 i++;  
 }  
 }, "producer");  
  
 Thread consumer = new Thread(() -> {  
 try {  
 while (true) {  
 System.*out*.println("consumed " + queue.take());  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }, "consumer");  
  
 producer.start();  
 consumer.start();  
 }  
}

**Producer consumer with wait, notify queue**

class ProducerConsumer\_With\_wait\_notify {  
  
 public static void main(String[] args) {  
  
 Queue<Integer> queue = new LinkedList<>();  
  
 */\*\*  
 \* This Thread produces element in every 1 second  
 \* Once it puts the element in a queue, it notifies other Threads  
 \*/* Thread producer = new Thread(() -> {  
 int element = 1;  
 while (element != 0) {  
 try {  
 // Produces element in every 1 second  
 Thread.*sleep*(1000);  
  
 synchronized (queue) {  
 System.*out*.println("Produced:: " + element);  
 // Adds element and notify other Threads  
 queue.add(element);  
 queue.notifyAll();  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 element++;  
 }  
 }, "producer");  
  
 */\*\*  
 \* This Thread consumes the element produced by producer Thread  
 \* First it checks if the queue is empty, if its empty then this thread waits  
 \*/* Thread consumer = new Thread(() -> {  
 while (true) {  
 synchronized (queue) {  
 while (queue.isEmpty()) {  
 try {  
 queue.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("consumed ::" + queue.remove());  
 queue.notifyAll();  
 }  
 }  
 }, "consumer");  
  
 producer.start();  
 consumer.start();  
 }  
}