**Multithreading in Java**

Multithreading in java is a process of executing multiple threads simultaneously. The aim of multithreading is to achieve the concurrent execution.

**Thread**

Thread is a lightweight components and it is a flow of control. In other words a flow of control is known as thread.

**State or Life cycle of thread**

State of a thread are classified into five types they are

1. New State
2. Ready State
3. Running State
4. Waiting State
5. Halted or dead State

**New State**

If any new thread class is created that represent new state of a thread, In new state thread is created and about to enter into main memory. No memory is available if the thread is in new state.

**Ready State**

In ready state thread will be entered into main memory, memory space is allocated for the thread and 1st time waiting for the CPU.

**Running State**

Whenever the thread is under execution known as running state.

**Halted or dead State**

If the thread execution is stoped permanently than it comes under dead state, no memory is available for the thread if its comes to dead state.

**Note:**If the thread is in new or dead state no memory is available but sufficient memory is available if that is in ready or running or waiting state.

**Achieve multithreading in java**

In java language multithreading can be achieve in two different ways.

1. Using thread class
2. Using Runnable interface

**Using thread class**

In java language multithreading program can be created by following below rules.

1. Create any user defined class and make that one as a derived class of thread class.

**class** Class\_Name **extends** Thread

{

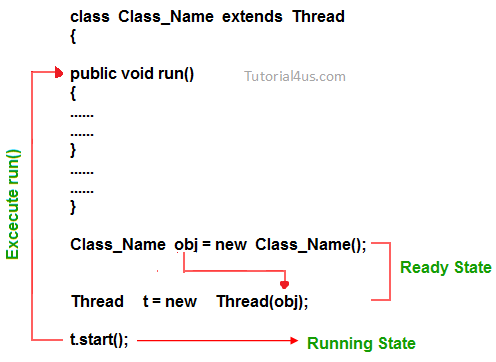
........

}

1. Override run() method of Thread class (It contains the logic of perform any operation)
2. Create an object for user-defined thread class and attached that object to predefined thread class object.

Class\_Name obj=new Class\_Name Thread t=new Thread(obj);

1. Call start() method of thread class to execute run() method.
2. Save the program with filename.java



**Example of multithreading using Thread class**

Thread based program for displaying 1 to 10 numbers after each and every second.

// Threaddemo2.java

**class** Th1 **extends** Thread

{

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

{

**try**

{

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

{

System.**out**.println("value of i="+i);

Thread.sleep(1000);

}

}

**catch**(InterruptedException ie)

{

System.err.println("Problem in thread execution");

}

}

}

**class** Threaddemo2

{

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

{

Th1 t1=**new** Th1();

System.**out**.println("Execution status of t1 before start="+t1.isAlive());

t1.start();

System.**out**.println("Execution status of t1 before start="+t1.isAlive());

**try**

{

Thread.sleep(5000);

}

**catch**(InterruptedException ie)

{

System.**out**.println("Problem in thread execution");

}

System.**out**.println("Execution status of t1 during execution="+t1.isAlive());

**try**

{

Thread.sleep(5001);

}

**catch**(InterruptedException ie)

{

System.**out**.println("problem in thread execution");

}

System.**out**.println("Execution status of t1 after completation="+t1.isAlive());

}

}

Output

Execution status of t1 before start=false //new state

Execution status of t1 after start=true //ready state

1

2

3

4

5

6

Execution status of t1 during execution=true //running state

7

8

9

10

Execution status of t1 after completation=false //halted state

**Thread class properties**

Thread class contains constant data members, constructors, predefined methods.

**Constant data members**

* MAX-PRIORITY
* MIN-PRIORITY
* NORM-PRIORITY

**MAX-PRIORITY**

Which represent the minimum priority that a thread can have whose values is 10.

Syntax:

public static final int MAX-PRIORITY=10

**MIN-PRIORITY**

Which represents the minimum priority that a thread can have.

Syntax:

public static final int MIN-PRIORITY=0

**NORM-PRIORITY**

Which represent the default priority that is assigned to a thread.

Syntax:

public static final int NORM-PRIORITY=5

**Constructors of Thread class**

* Thread()
* Thread(String name)
* Thread(object)
* Thread(object, String name)

**Thread()**

Which will be execute to set the predefined name for newly created thread, these names are generally in the form of thread -0, thread -1, ....  
Syntax to call constructor:

Syntax

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

**Thread(String name)**

Which can be used to provide user defined name for newly created thread.

Syntax

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

**Thread(object)**

Which can be used to provide default name for newly created user defined thread.

Syntax

UserdefinedThreadclass obj=**new** UserdefinedThreadclass();

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

**object, String name**

Which will be used to provide user defined name for the newly created user defined thread.

Syntax

UserdefinedThreadclass obj=**new** UserdefinedThreadclass();

Thread t=**new** Thread(**object**, "secondthread");

**Methods of Thread class**

* getPriority()
* setPriority()
* getName()
* setName()
* isDeamon()
* run()
* start()
* sleep()
* suspend()
* resume()
* stop()
* isAlive()
* currentThread()
* join()
* getState()
* yield()

**getPriority()**

This method is used to get the current priority of thread.

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

**int** x=t.getPriority();

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

**setPriority()**

This method is used to set the current priority of thread.

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

t.setPriority(any priority number between o to 10)

**or**

t.setPriority(Thread.MAX-PRIORITY)

**getName()**

This method is used to get the current executing thread name.

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

String s=t.getName();

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

**setName()**

This method is used to set the userdefined name for the thread.

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

t.setName("mythread");

**isDeamon()**

Which returns true if the current thread is background thread otherwise return false.

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

**boolean** b=t.isDeamon();

**run()**

Which contains the main business logic that can be executed by multiple threads simultaneously in every user defined thread class run method should be overridden.

**public** Class\_Name **extends** Thread

{

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

{

.....

.....

}

}

**start()**

Used to convert ready state thread to running state.

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

t.start();

**sleep()**

Used to change running state thread to ready state based on time period it is a static method should be called with class reference.

**public** **static** **final** sleep(**long** milisecond)**throws** InterruptedException

{

**try**

{

Thread.sleep(3000);

}

**catch**(InterruptedException ie)

{

........

........

}

}

Once the given time period is completed thread state automatically change from waiting to running state.

**suspend()**

Used to convert running state thread to waiting state, which will never come back to running state automatically.

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

t.suspend();

**resume()**

Used to change the suspended thread state(waiting state) to ready state.

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

t.resume();

**Note:**Without using suspend() method resume() method can not be use.

**What is the difference between sleep() and suspend()**

Sleep() can be used to convert running state to waiting state and automatically thread convert from waiting state to running state once the given time period is completed. Where as suspend() can be used to convert running state thread to waiting state but it will never return back to running state automatically.

**stop()**

This method is used to convert running state thread to dead state.

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

t.stop();

**isAlive()**

Which is return true if the thread is in ready or running or waiting state and return false if the thread is in new or dead state.

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

t.isAlive();

**currentThread()**

Used to get the current thread detail like thread name thread group name and priority

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

t.currentThread();

**Note:**

* The default thread name is thread-0, (if it is a main thread default name is main)
* The default thread group name is main
* Default thread priority is "5" is normal priority.

**join()**

Which can be used to combined more than one thread into a single group signature is public final void join()throws InterruptedException

**try**

{

t.join();

t2.join();

.....

.....

}

**getState()**

This method is used to get the current state of thread.

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

t.getState();

**yield()**

Which will keep the currently executing thread into temporarily pass and allows other threads to execute

**Using Runnable Interface**

Runnable is one of the predefined interface in java.lang package, which is containing only one method and whose prototype is " Public abstract void run "

The run() method of thread class defined with null body and run() method of Runnable interface belongs to abstract. Industry is highly recommended to override abstract run() method of Runnable interface but not recommended to override null body run() method of thread class.

In some of the circumstance if one derived class is extending some type of predefined class along with thread class which is not possible because java programming never supports multiple inheritance. To avoid this multiple inheritance problem, rather than extending thread class we implement Runnable interface.

**Rules to create the thread using Runnable interface**

* Create any user defined class and implements runnable interface within that
* Override run() method within the user defined class.
* call start() method to execute run() method of thread class
* Save the program with classname.java

**class** Class\_Name implement Runnable

{

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

{

........

}

}

Class\_Name obj=**new** Class\_name();

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

t.start();

**Note:**While implementing runnable interface it is very mandatory to attach user defined thread class object reference to predefined thread class object reference. It is optional while creating thread by extending Thread class.

**Thread Synchronization**

Whenever multiple threads are trying to use same resource than they may be chance to of getting wrong output, to overcome this problem thread synchronization can be used.

**Definition:**Allowing only one thread at a time to utilized the same resource out of multiple threads is known as thread synchronization or thread safe.

In java language thread synchronization can be achieve in two different ways.

1. Synchronized block
2. Synchronized method

**Note:**synchronization is a keyword(access modifier in java)

**Synchronized block**

Whenever we want to execute one or more than one statement by a single thread at a time(not allowing other thread until thread one execution is completed) than those statement should be placed in side synchronized block.

**class** Class\_Name implement Runnable **or** **extends** Thread

{

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

{

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

{

.......

.......

}

}

}

**Synchronized method**

Whenever we want to allow only one thread at a time among multiple thread for execution of a method than that should be declared as synchronized method.

**class** Class\_Name implement Runnable **or** **extends** Thread

{

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

{

**synchronized** **void** fun()

{

.......

.......

}

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

{

fun();

....

}

}

**Interthread Communication**

The process of execution of exchanging of the data / information between multiple threads is known as Interthread communication or if an output of first thread giving as an input to second thread the output of second thread giving as an input to third thread then the communication between first second and third thread known as Interthread communication.

In order to develop Interthread communication application we use some of the methods of java.lang.Object class and these methods are known as Interthread communication methods.

**Interthread communication methods**

1. public final void wait(long msec)
2. public final void wait()
3. public final void notify()
4. public final void notifyAll()

**public final void wait(long msec)**

public final void wait (long msec) is used for making the thread to wait by specifying the waiting time in terms of milliseconds. Once the waiting time is completed, automatically the thread will be interred into ready state from waiting state. This methods is not recommended to used to make next thread to wait on the basis of time because java programmer may not be able to decide or determine the CPU burst time of current thread and CPU burst time is decided by OS but not by the programmer.

**public final void wait()**

public final void wait() is used for making the thread to wait without specifying any waiting time this method is recommended to use to make the next thread to wait until current thread complete its execution.

**public final void notify()**

public final void notify() is used for transferring one thread at a time from waiting state to ready state.

**public final void notifyAll()**

public final void notifyAll() is used for transferring all the threads at a time from waiting state to ready state.