MULTI –THREADING

MultiTasking:

Processing/Executing more than one task /several tasks simultaneously is called MultiTasking.

There two types of multitasking. They are:

→Process based multitasking.

→Thread based multitasking.

Processed Based Multitasking:

Processing multiple tasks simultaneously based on address/process is called process Based Multitasking.

Here each job can be done by separate process.

Example:

The best example of Process Based Multitasking is when we reading a book first we see the word after we are reading/speaking the word.

Here seeing the word is one task and reading word is another task.

These two tasks can be done by simultaneously.

In the Process Based Multitasking switching one context /area to another context/ area is take more time.

Each and every process having its own address/context, so switching from one context to another context takes more time. It is heavy weight process.

Resources consuming is more.

Thread Based Multitasking:

Processing multiple jobs/tasks simultaneously based on Thread is call Thread Based Multitasking.

Here each task is separately individual part.

This is best suitable for programming.

Thread uses less resources when compare Process.

We simply say this; collection of threads is called process.

Here all threads are placed in a one context/ area, so switching the control from one context to context is not take that much of time when compare to Process switching.

That's why comparing process based multitasking, thread based multitasking best suitable for application performance.

We can also say thread based multitasking as Multithreading.

In the both cases of multitasking we should get performance, the reason is reducing the time that means complete the task quickly. This is light weight process. Resource consuming is less. Java provides huge libraries to work with multi threading in the form of API like Runnable, Thread, ThreadGroup.

Thread:

A thread represents a separate path of execution of a group of statements.

In java program, we have group of statements, these are executed one by one statements by JVM. We can also specify Thread is a stack which is created in Java Stacks Area. In our program the default thread is main, which is executed by JVM.

(Q) Why should we go for Multithreading?

Threads are mainly used in server-side programming.

Threads can give the response to all the clients with in the same time.

Threads can also used at developing games, animation.

In java simply we can say JVM is a process, which is by default having two Threads for each and every program. They are:

main: Used to execute the methods.

garbage collector: Deleting the objects, which are not used by using mechanism called Mark and Sweep.

When we working with threads we can see two types of execution, they are:

Sequential execution:

A single thread can executes each and every method one by one. In sequential execution the time for the completion of work is more.

Concurrent execution: Methods are executed simultaneously. The completion of the work is very fast when compare to sequential execution.
Note: At a time a thread can execute only one time.

In concurrent execution the tasks will be processed in the bellow manner.

Start→suspend→resume→end.

In this process user should create multiple threads to work with multiple tasks.

```
public class Demo {
	void m1(){
		System.out.println("m1-method: "+
				Thread.currentThread().getName());
	}
	static void m2(){
		System.out.println("m2-method: "+
				Thread.currentThread().getName());
	}
	public static void main(String[] args) {
		System.out.println("main-method: "+
					Thread.currentThread().getName());
		Demo d = new Demo();
		d.m1();
		m2();
	}
}
```

In the above program we unable to get multithreading concept.

To develop multithreading concept we required multiple threads.

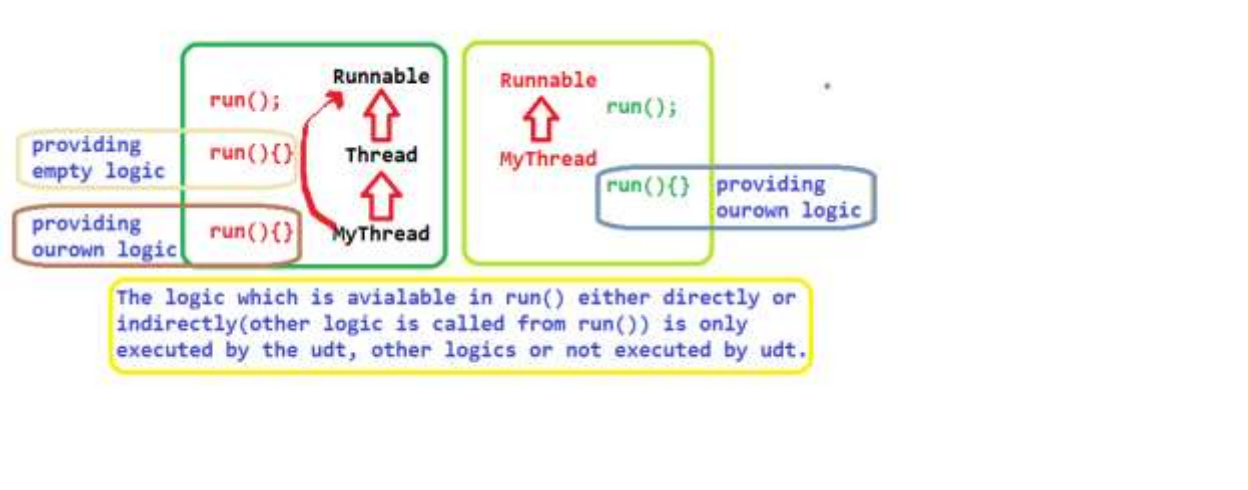
In java every code(static block,instance lock,static method, non-static method, constructor) will be executing by "main thread" only.

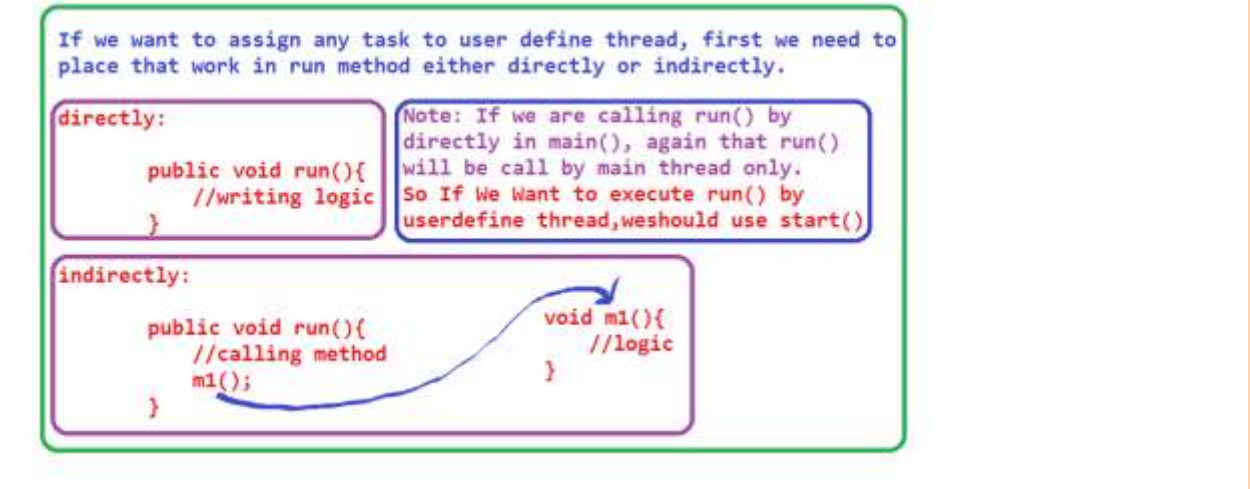
Creation of UserDefine/Custom Thread:

If we want to create user define thread in java we have two ways.

a. By extends java.lang.Thread class

b. By implements java.lang.Runnable interface.





By extends java.lang.Thread:

package thread;

```
public class MyThread extends java.lang.Thread{
	public void run(){
		System.out.println("varun/userdefine thered" +
				" priority: "+Thread.currentThread().getPriority());
		for(int i =0; i< 10; i++){
			System.out.println(Thread.currentThread().getName()+
				".."+i);
		}
	}
	public static void main(String[] args) {
		MyThread mt = new MyThread();
		//mt.run();
		mt.setName("varun");
		//mt.setPriority(10);
		//mt.setPriority(-11);
		mt.start();
		//mt.start();
		//mt.stop();
		System.out.println("main thered priority: "+
		Thread.currentThread().getPriority());
		for(int i =0; i< 10; i++){
			System.out.println(Thread.currentThread().getName()+
					..."+i);
			//Thread.currentThread().stop();
```

```
            }
            System.out.println(Thread.MIN_PRIORITY);
            System.out.println(Thread.NORM_PRIORITY);
            System.out.println(Thread.MAX_PRIORITY);
        }
}
```

First we should take one class, that class must be extends

java.lang.Thread class. The reason if we want develop any

User define thread class, that class must be need java.jang.Thread class functionalaties.

```
class MyThread extends java.lang.Thread{
}
```

If we want execute user define logic with the help of userdefine thread, that code must be write within the run().

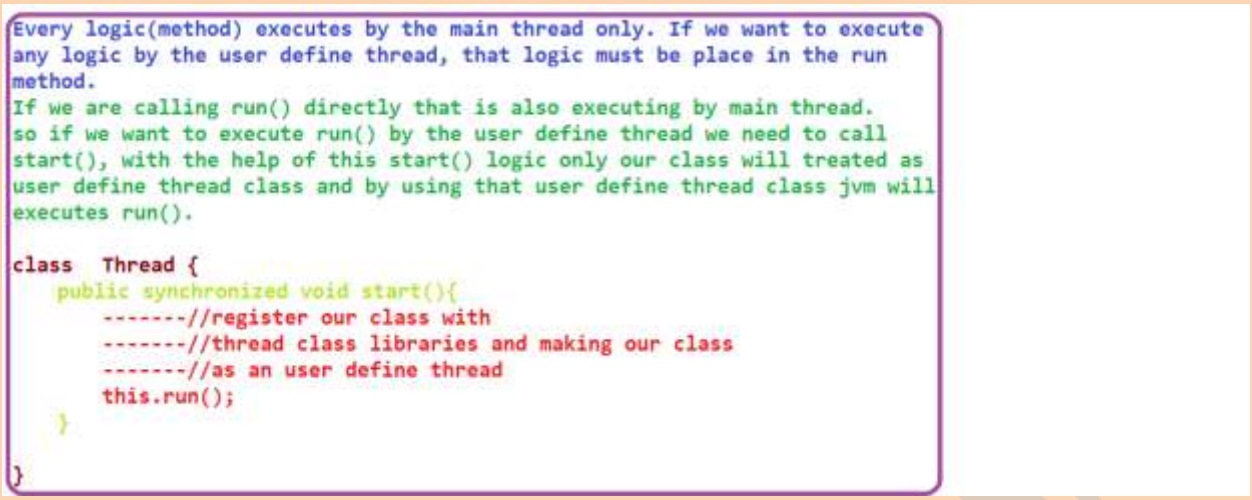
All the main method statements execute by the main thread.

If we are calling any method from main(), that called method statements also execute by the main thread only.

So if we are calling run () directly, then main thread will be execute that method but not user define thread.

If we want execute run () by the user define thread then weshould we start().

start () of Thread class automatically calls user define override run(). That means that run () internally execute by user define thread only.



```
Every logic(method) executes by the main thread only. If we want to execute
any logic by the user define thread, that logic must be place in the run
method.
If we are calling run() directly that is also executing by main thread.
so if we want to execute run() by the user define thread we need to call
start(), with the help of this start() logic only our class will treated as
user define thread class and by using that user define thread class jvm will
executes run().

class  Thread {
    public synchronized void start(){
        -------//register our class with
        -------//thread class libraries and making our class
        -------//as an user define thread
        this.run();
    }

}
```

If we want start the thread we need start().

If we want stop the thread we have stop().

If we want to know the thread name

Then we should bellow syntax.

Thread.currentThread().getName().

If we want to give name to thread, we have method like

setName(String name).

Thread priorities are start from 1 to 10.

Main thread priority is 5. Whatever the threads which arecreated by the main thread those thread priority is also 5.

Java provides three constants to know the priority of an threads those are

MIN_PRIORITY

NORM_PRIORITY

MAX_PRIORITY

Thread minimum priority is 1.

Normal priority is 5

Maximum priority is 10.

If we want setting priority and getting priorities of a thread we have two predefine methods.

1.setPriority()

2.getPrioity()

If we are setting thread priorities more than 10 and Less than 1 then we will get runtime exception this is Java.lang.IllegalArgumentException.

We cannot call start () more than one time on one single thread object.

If we are calling start () more than one time, then we will get one runtime exception java.lang.IllegalThreadStateException.

Thread execution is not reliable, thread execution entirely depend upon JVM.

Q) Can we override start () in our class?

A) Yes. We can.

```
public class Demo extends Thread {

	@Override
	public void run(){
		System.out.println("run_method: "+
				Thread.currentThread().getName());
	}

	@Override
	public void start(){
		System.out.println("start_method: "+
				Thread.currentThread().getName());
		run();//this.run()
	}

	public static void main(String[] args) {
		System.out.println("main_method: "+
					Thread.currentThread().getName());
		Demo d = new Demo();
		//d.run();
		d.setName("kit");
		d.start();
	}
}
```

If we calling start () on user define thread object, then control not goes to start () of Thread class. User define Start () will be execute. In this time run () not executed.

If we are overriding the start () method, that method will be treated as a normal method by the JVM.

Note: If we want to call the run () by the user define thread, then we should forward our control to java.lang.Thread class start ().

How to execute block and method level logic by the udt?

```
package threads;

public class MyThread extends java.lang.Thread{
	@Override
	public void run(){
		System.out.println("-----------------");
		System.out.println("run : "+
	Thread.currentThread().getName());
		new MyThread();
		m1();
	}
	{
		System.out.println("block: "
	+Thread.currentThread().getName());

	}
	void m1(){
		System.out.println("m1: "+
	Thread.currentThread().getName());
	}
	public static void main(String[] args) {
		System.out.println("main: "+
				Thread.currentThread().getName());
		MyThread mt = new MyThread();
		System.out.println("----------------------");
		//mt.run();
		mt.setName("UDT");
		mt.start();
	}
}
```


Creating user define thread by implementing java.lang.Runnable interface

Program: ---------need to add----

Q)Difference between extends Thread implements Runnable?

A)If we extends Thread, we are unable to extends some other class functionalities, the reason java does not support multiple inheritance through classes.

But if we go for Runnable interface we can access Runnable functionalaties and some other class funcationalities.

If we extends Thread class, we are unable use same object more than one time. If we call start() on same object more than one time we will one exception that is java.lang.IllegalThreadStateException.

But if we implements Runnable, we can same object more than one with the help of Thread class reference.

If we extends Thread class we will get all functionalites, but if we go for Runnable we will get required functionalty.

extends Thread	implements Runnable
If we extends Thread, our class unable extends some other class, the reason java doesn't support multiple inheritance. If we extends Thread, our reference can not be reusable. If we extends Thread, we will all functionalities.	If we implements Runnable, we can make our class as an thread class and also we will some other class funcaitonalities. If we implements Runnable, we reuse our user define thread reference. If we want access limited functionalities, we should implements Runnable

How to execute different logics by different thread from the same run():

```
package jdbc;

class Test extends Thread{
    @Override
    public void run(){
        String s= Thread.currentThread().getName();
        if(s.equalsIgnoreCase("bus")){
            m1();
        }
```

```
        else
            if(s.equalsIgnoreCase("car")){
                m2();
            }
    }
    public void m1(){
        for(int i=1;i<=10;i++){
System.out.println("m1:
"+Thread.currentThread().getName()+ "..."+i);
        }
    }
    public void m2(){
        for(int i=101;i<=110;i++){
System.out.println("m2:
"+Thread.currentThread().getName()+ "..."+i);
        }
    }
    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = new Test();
        t1.setName("bus");
        t2.setName("car");
        t1.start();
        t2.start();
        System.out.println("program ends");
    }
}
```

Controlling the threads:

1. yield ():

If we want forwarding the control from one thread to another thread based on priority then we can go for yield().

Before giving the control to waiting thread, currently executing thread checks priority of waiting thread.

If waiting thread priority is greater than or equal to current thread then automatically control goes to waiting thread.

2. join(): If we want push the waiting thread into execution state then we can use join().If one thread execution depends on another thread response, in that scenario we should use join().

Java provide threetypes of join methods.

join()

join(-)

join(-,-)

Q) What is difference between join() and join(-)?

A) If we are using join(), then joined thread will execute entire work. Whereas join(-) will gives chance to thread only some period of time.

Within the period of time if work is completethen control goes to waiting thread, otherwise in the middle of the work only control goes to waiting thread.

3.sleep():

This method is use to placing the current execution thread in sleep mode for particular time.We have two different sleep methods.

sleep(-) and sleep(-,-)

Note: Both join() and sleep(-) are throwing compile time Exception. That is java.lang.InterruptedException

Differences between join and sleep and yield methods.

	Yield	join	sleep
Number of method:	1	3	2
Static	yes	no	yes
Native	yes	no	yes
Final	no	yes	no
Interrupted	no	yes	yes
Exception			
Synchronized	no	2—syn(join(-)	no
		Join(-,-)	
		Join()(not synt)	

Synchronization:

It is a mechanism, which is used to allow only one thread to execute the entire resource.

If more than one thread is executing the only one common resource, then we should allow only one thread at a time. After complete the work by the allowed thread, then remaining threads will get chance to execute.

If we want to get these type of functionalities, we should go for synchronization.

If we want to achieve synchronization in java, we have one keyword that is "synchronized".

"synchronized" keyword we can be applied on top of methods and blocks not on classes and variables.

If more than one thread executing the single resource we will get inconsistency output. (Race condition)

If any one thread executing the synchronized method, first that thread will get lock on that object, once that method execution completed thread will releases the lock. Getting the lock and releasing the lock is taken care by jvm only.

Jvm will give the chance to only one thread to execute the synchronized method. Once thread is executing the synchronized method, remaining threads can not execute same synchronized method, but other threads can execute other non-synchronized methods.

```
class Cinema{
	synchronized public void cinemaTalk(String name){
		for(int i=0;i<10;i++){
			try {
				Thread.sleep(100);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println("good: "+name);
		}
		System.out.println("==================");
	}


	void m1()throws InterruptedException{
		for(int i=0;i<10;i++){
			Thread.sleep(100);
			System.out.println("m1 method"+
		Thread.currentThread().getName());
		}
	}
}
class Person extends Thread{
	Cinema c;
	String name;
	Person(Cinema c , String name ){
		this.c = c;
		this.name = name;
	}
	public void run(){
		c.cinemaTalk(name);

	}
}
public class Synchronization extends Thread{
	public static void main(String[] args)
			throws InterruptedException{
		Cinema c = new Cinema();//1010
```

```
            Person p1 = new Person(c,"ram");
            p1.start();

            Person p2 = new Person(c,"sam");
            p2.start();
            c.m1();
        }
}
```

Ex:

```
synchronized void m1 () {
        //synchronized method
}
Void m2 () {
        synchronized (MyThread.class){
                //synchronized block
        }
}


class Display{
         synchronized void wish(String msg){
                for(int i=0;i<10;i++){
                        System.out.println("good morning  "       +msg);
                }
        try{
                Thread.sleep(5000);
        }catch(Exception e){}
}
}
```

```
class MyThread1 extends Thread{
	Display d1;
	String msg;
	MyThread1(Display d1, String msg){
		this.d1 = d1;
		this.msg=msg;
	}
	public void run(){
		d1.wish(msg);
	}
}
public class SynchronizedDemo {
	public static void main(String[] args) {
		Display d = new Display();
		MyThread1 m1 = new MyThread1(d,"ram");
		MyThread1 m2 = new MyThread1(d,"sam");
		m1.start();
		m2.start();
	}
}
```

In the following scenario we have two threads on two different resources, that means every thread has it own resource, in that time if we are applying synchronization there is no usage.

```
class Display{
 synchronized void wish(String msg){
            for(int i=0;i<10;i++){
            System.out.println("good morning  "       +msg);
            }
            try{
                  Thread.sleep(5000);
            }catch(Exception e){}
      }
}

class MyThread1 extends Thread{
      Display d1;
      String msg;
      MyThread1(Display d1, String msg){
            this.d1 = d1;
            this.msg=msg;
      }
      public void run(){
            d1.wish(msg);
      }
}
public class SynchronizedDemo {
      public static void main(String[] args) {
```

```
        Display d = new Display();
        Display d1 = new Display();
        MyThread1 m1 = new MyThread1(d,"ram");
        MyThread1 m2 = new MyThread1(d1,"sam");
        m1.start();
        m2.start();
    }
}
```

Difference between join() and synchronizaiton?

whenever we call join() on top of one thread,then join thread will executing the task, current thread will be in the stop mode.
If we are allowe two thread on synchronized method at a time one thread will execute task, remaining thread is not in stop mode, can execute other work.

Wehenever we call join() on top of any thread that thread will execute entire task(all statements), but with the help of synchronization block we execute some of the statements of the method by only one thread at a time.

performance: Both low, but comparing to join(), synchronization having little bit of more performance.

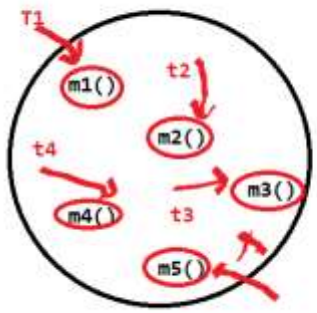
If any thread which executes static synchronized methods, remaining threads not execute static synchrnozed method but remaining thread can executes

static non-synchronized methods

non-static non-synchronized methods

non-static synchronized methods.

```
class SynDemo{
    static synchrnonized void m1(){}
    static void m2();
    synchronized void m3(){}
    void m 4();
    static synchrnonized void m5(){}

}
```



In this particular time we should go for locking mechanism.

We have two types locking.

1. Class level locking.
2. Object level locking.

```
class DemoClass{
    public void demoMethod(String str){
        //synchronized(this){  //object level locking
        synchronized(DemoClass.class){  //class level locking
            for(int i=0;i<10;i++){
                System.out.println("good morning"+"..."+str);
            }
            try{Thread.sleep(2000);
            }catch(Exception e){
                e.printStackTrace();
            }
        }
}
```

```
}
class Supported extends Thread{
	DemoClass d;
	String str;
	Supported(DemoClass d,String str){
		this.d =d;
		this.str = str;
	}
	public void run(){
		d.demoMethod(str);
	}

}
public class SynchronizedDemo1 {
	public static void main(String[] args) {
		DemoClass dc = new DemoClass();
		DemoClass dc1 = new DemoClass();
		Supported s =  new Supported(dc,"ram");
		Supported s1 =  new Supported(dc1,"sam");
		s.start();
		s1.start();
	}
}
```

Drawbacks: Performance will be decreases.

The reason is multiple threads are in the waiting mode, program execution will take more time, if execution time is increases automatically performance will be decreases.

But synchronization is very good at result. Provides accurate/reliable results.

resume() and suspend():

With the help of suspend() we can stop the particularThread.

With the help of resume() we can start the particular thread.

```
package jdbc;


public class MyThread  extends Thread{
	@Override
	public void run(){
		for(int i=1;i<=20;i++){
		System.out.println("run: "+i+" "
					+Thread.currentThread().getName());
			try {
				Thread.sleep(500);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
	public static void main(String[] args) throws InterruptedException {
		MyThread mt1 = new MyThread();
		mt1.setName("ram");
```

```
        MyThread mt2 = new MyThread();
        mt2.setName("NareshIT");
        mt1.start();
        mt2.start();
        Thread.sleep(2000);
        mt2.suspend();
        Thread.sleep(2000);
        mt2.resume();
        Thread.sleep(2000);
        mt1.suspend();
        Thread.sleep(2000);
        mt1.resume();
        Thread.sleep(2000);
    }
}
```

Inter Thread communication:

One thread is communicating with another for providing notifications about work completion is called inter thread communication.

Wait() and notify() and notifyAll() metods we should use within the synchronized methods.

Whenever we use wait(), thread will release the lock on top of that particular object. Wait() is not only placing thread into wait state and also releasing lock. Where as sleep(-) will placing thred into sleep state upto particular but thread is not releasing the lock.

Whenever we apply wait() on particular thread, that thread will goes waiting state upto life time.

Wait(-)/wait(-,-) will place the thread into wait state upto that particular time.

```
class Customer{
	int amount=10000;
	synchronized void withdraw(int amount){
			System.out.println("going to withdraw...");
			if(this.amount<amount){
				System.out.println("Less balance; waiting " +
					"for deposit...");
				try{
					//wait(5000);
					wait();
				//Thread.sleep(10000);
				}catch(Exception e){
				}
			}
			this.amount=this.amount-amount;
			System.out.println("After withdrawl: "+this.amount);
			System.out.println("withdraw completed...");
	}
	synchronized void deposit(int amount){
			System.out.println("going to deposit...");
			System.out.println("before deposit: "+this.amount);
```

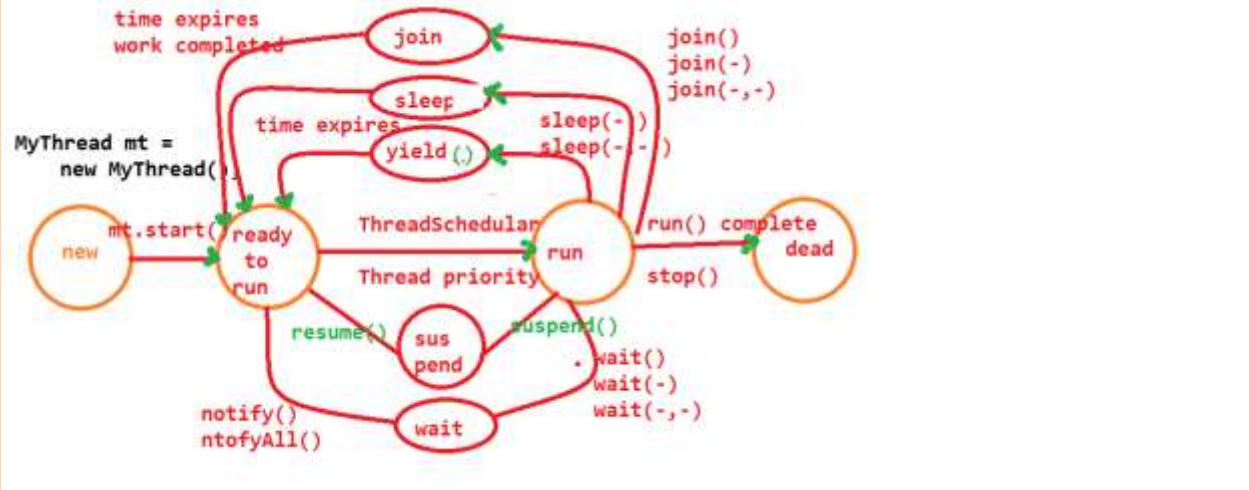
```
                this.amount =  this.amount+ amount;
                System.out.println ("depositcompleted... ");
                System.out.println("After Deposit: "+this.amount);
                notifyAll();
                //notify();
        }
}


public class WaitTest{
        public static void main(String args[]) throws InterruptedException{
                int a = 10/0;
                final Customer c=new Customer();
                new Thread()
                {
                        public void run(){
                                c.withdraw(15000);
                        }
                }.start();
                /*              new Thread()
                {
                        public void run(){
                                c.withdraw(15000);
                        }
```

```
        }.start();



        */
        new Thread(){
            public void run(){
                c.deposit(10000);
                }
            }.start();
        }
    }
```

Whenever Thread executes wait(), that thread automatically releases the lock. If there is any waiting thread that thread will get the lock on that particular object. Where as sleep(-) not releasing the lock by thread. Only it will pause the thread upto some particular time.

Sleep(-) we can use in both synchronized and non-synchronized methods, where as wait() we must and should use in synchronized methods only.

Thread Life Cycle:



Before executing run() by the userdefine thread, whatever priority to set that user define thread , that priority only applicable to that that thread.

```
class MyThread extends Thread{
	public void run(){

		try {
			Thread.sleep(3000);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
			System.out.println("run : "+
					Thread.currentThread().getName());
			System.out.println("run : "+
					Thread.currentThread().getPriority());
	}
	public static void main(String[] s) throws InterruptedException{
		MyThread mt = new MyThread();
		MyThread mt1 = new MyThread();

		mt.setName("ram");
		//mt.setPriority(11);//java.lang.IAE
		mt.setPriority(9);
		mt.setPriority(4);
		mt1.setName("yaane");
		mt1.start();

		mt1.join();
		mt.start();
		mt.setPriority(6);
		//mt.start();
```

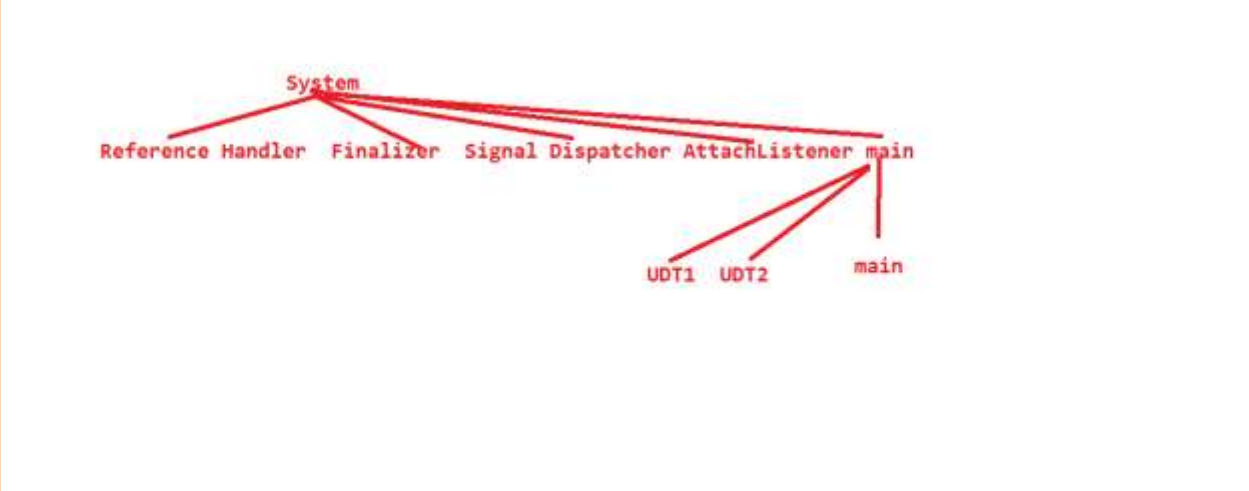
```
    }
}

public class Demo extends Thread{
    @Override
    public void run(){
        for(int i=1;i<=10;i++){
        System.out.println("run: "+i+"."+

    Thread.currentThread().getName()+"..."+Thread.cur
rentThread().isAlive());
        try{
            Thread.sleep(1000);
            if(i==5){
                Thread.currentThread().stop();
            }
        }catch(Exception e){
        }
        }
    }
    public static void main(String[] args)throws
InterruptedException {
        Demo d1 = new Demo();

        d1.setName("NIT");
        d1.start();
        Thread.currentThread().sleep(12000);
        System.out.println(d1.isAlive());

    }
}
```

Garbage Collection:

In java we have two types of threads.

1. Daemon Thread.
2. Non-Daemon Thread.

We have following 5 import threads.



Thread: Executing group of statement in the separate path.

Non-Daemon Thread:

These are high priority thread.

These threads must be executed.

Ex: main thread.

All user define threads also come under Non-Daemon thread.

Thre default value of Non-Daemon thread is '5'.

If we want to check whether the thread is daemon or not, in java we havae one method that is " isDaemon()".

If thread is Daemon then this method returns true, otherwise returns false.

we can change our thread from non-daemon to daemon, with the help of one method i.e "setDeamon".

Deamon Thread:

These are low priority threads.

We are not give gaurantee to execute.

ex: Garbage collector.

It is an one low priority thread.

It is executed in the background by the jvm.

```
public class MyThread extends Thread{
	public void run(){
		System.out.println("udt thread type: "+
				Thread.currentThread().isDaemon());
		try{
			for(int i=0;i<1000;i++){
				Thread.sleep(100);
				System.out.println("run: "+
			Thread.currentThread().getName()+"  "+i);
			}
		}
		catch(InterruptedException e){
		}
	}
	public static void main(String[] args) {
		System.out.println("main thread type: "+
	Thread.currentThread().isDaemon());
		MyThread mt = new MyThread();
		mt.setName("yaane");
		mt.setDaemon(true);
		mt.start();
		for(int i=0;i<1000;i++){
			System.out.println("main: "+
		Thread.currentThread().getName()+"  "+i);
		}
	}
}
```

Why should we go for GarbageCollection?

If we go to any other language like c++, if we want provide memory(allocating memory) our data programmer should write some code and also if we want to deallocating the memory in this time programmer should write some code.

If programmer forgot about deallocating memory code, some time later the memory will be filled, there may be chance of program is going to be stop .

To avoid above drawback java introduced one predefine service or functionalaty i.e memory deallocation with the help of Garbagecollector thread.

To cleanup the unused memory is called GarbageCollection.

```
public class Test {
        static void m1(){
                System.out.println("m1 method");
                Test t1 = new Test();
                Test t2 = new Test();
        }
    public static void main(String[] args) {
        Test t = new Test();
        Test t1 = new Test();
        Test t2 = new Test();
        //no object garbaged
        t1=null;
        //one object
        t=null;
        //two object
```

```
        t2= null;
        //three object
        m1();
        System.out.println("*****");
    }
}
```

Communicating with GarbageCollector:

In java we have two to communicate the garbage collector.

1. System.gc();
2. Runtime.getRuntime().gc();

GarbageCollector internally calls finalize() method to deallocate memory.

```
    public void finalize(){
    }
```

Another name of Garbage Collector is Finalizer.

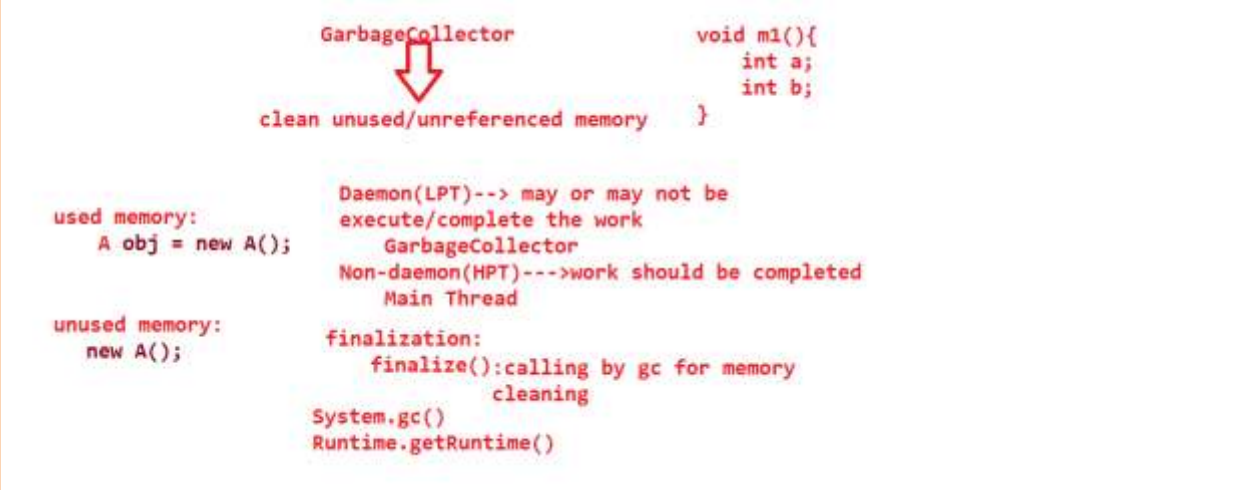
Executing the finalize method and deallocating the memory is called finalization.

```
public class Test {
    public void finalize(){
System.out.println(Thread.currentThread().getName()+" object garbaged");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t=null;
        System.gc();
        Test t1 = new Test();
```

```
        t1.finalize();
        Test t2 = new Test();
        t2 = null;
        //Runtime rt = new Runtime();  //wrong
        Runtime rt1 = Runtime.getRuntime();
        rt1.gc();
        //System.gc();
    }
}
```

If we are calling finalize() explacitly like object.finalize(), in this time Garbage collector not execute finalize method, the respective thread will be execute the finalize(). So it is just method calling not finalization.

```
final: keyword
       apply on variable,methods,class
       final variable not change
       final method not partispated in
                    Method overrding
       final class not partispated in
                        inheritance

finlly: block
       holding resource cleanup code
       closing db connection,closing file
           connection

finalize(): method
          it is calling by gc to clean the
memory
```



If finalize() internally calls by garbagecollector then only memory will be deallocated otherwise memory not deallocated.

If we are calling Stystem.gc() and Runtime.getRuntime().gc() repectively only one time memory will be garbaged.(not more than one time)

```
public class Test {

	int i = 111;

	public static void main(String[] args) {

		Test t1 = new Test();

		Test t2 = new Test();

		System.out.println("t1: "+t1.i);

		System.out.println("t2: "+t2.i);

		t2.i=333;

		t1=t2;

		System.out.println("t1 object is garbaged");

		System.out.println("t1: "+t1.i);

		System.out.println("t2: "+t2.i);

		t1.i=999;

		System.out.println("t1: "+t1.i);
```

```
            System.out.println("t2: "+t2.i);
        }
}
public class Test {
        int i = 111;
        static void m1(Test t1){
            Test t2 = new Test();
            Test t3 = new Test();
        }
        public static void main(String[] args) {
            Test t = new Test();
            System.out.println(t.i);
            m1(t);
            System.out.println(t.i);
        }
}
```

In the above program t2 and t3 memory will be deallocated and t1 reference will cancelled.

```
public class Test {
        public void finalize(){
            System.out.println(Thread.currentThread().getName()
                    +" is called");
            int a = 10/0;
        }
        public static void main(String[] args) {
```

```
        Test t = new Test();
        //t.finalize();
        t = null;
        System.gc();
    }
}
```

When ever Finalizer(GarbageCollector) calls the finalize(), if any

RuntimeExceptions are raised those are not thrown by finalize() that means unchecked exceptions are not thrown by finalize().

but if any thread call finalize() explacitly then finalize() will throws uncheckedexception messages.

```
public class MyThread extends Thread{
    static Thread mt;
    public void run(){
        try{
            mt.join();
        }
        catch(InterruptedException e){
            e.printStackTrace();
        }
        for(int i=0;i<10;i++){
            System.out.println("child thread");
        }
    }
```

```
	public static void main(String[] args) throws InterruptedException,
InstantiationException, IllegalAccessException {
		MyThread.mt = Thread.currentThread();
		MyThread mt1 = new MyThread();
		mt1.start();
		for(int i=0;i<10;i++){
			System.out.println("main thread");
			Thread.sleep(1000);
			mt1.join();
		}
	}
}
=====================================
public class MyThread extends Thread {
	@Override
	public void run(){
		for(int i=1;i<=10;i++){
		System.out.println("run: "+" "+i+" "+
				Thread.currentThread().getName());
		try {
				Thread.sleep(1000);
		} catch (InterruptedException e) {
				e.printStackTrace();
		}
		}
```

```
    }
    /*static{  //invalid
        System.out.println("static block");
        Thread t = Thread.currentThread();
        System.out.println("*********: "+t.getName());
        t.setDaemon(true);
    }*/
    public static void main(String[] args){
        /*Thread t = Thread.currentThread();  //invalid
        System.out.println("*********: "+t.getName());
        t.setDaemon(true);*/
        MyThread mt = new MyThread();
        mt.setName("UDT");
        System.out.println("check: "+mt.isDaemon());
        mt.setDaemon(true);
        System.out.println("check: "+mt.isDaemon());
        mt.start();
        //mt.setDaemon(true);
        //dont use setDaemon() after start() method
        for(int i=1;i<=10;i++)
            System.out.println("main: "+" "+i+" "+
                    Thread.currentThread().getName());
    }
}
```

```
==============================
ThreadGroup tg = Thread.currentThread().getThreadGroup().getParent();
		System.out.println(tg.getName());
		Thread[] t = new Thread[tg.activeCount()];
		System.out.println(t.length);
		tg.enumerate(t);
		for(Thread t1: t)
		{
			System.out.println(t1.getName());
		}
==========================================
package threads;
public class MyThread extends Thread {
	MyThread(ThreadGroup tg,String name){
		super(tg,name);
	}
	public void run(){
		System.out.println("chaild thread");
		try{
			Thread.sleep(5000);
		}
		catch(Exception e){
		}
```

```
	}
	public static void main(String[] args) throws InterruptedException{
		ThreadGroup tg = new ThreadGroup("ParentGroup");
		ThreadGroup cg = new ThreadGroup(tg,"chaildGroup");
		MyThread mt1 = new MyThread(tg,"chaildthread1");
		MyThread mt2 = new MyThread(tg,"chaildthread2");
		mt1.start();
		mt2.start();
		/*MyThread mt3 = new MyThread(cg,"chaildthread3");
		mt3.start();*/
		System.out.println("tg*****:"+tg.activeCount());
		System.out.println("tg1*****:"+tg.activeGroupCount());
		System.out.println("cg*****:"+cg.activeCount());
		System.out.println("cg1*****:"+cg.activeGroupCount());
	}
}
```

==========

```
package threads;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class BookTicket implements Runnable{
	String name;
	BookTicket(String name){
		this.name = name;
```

```
	}



	public void run(){
		System.out.println("Ticket is trying to book by: "+name+"...."+Thread.currentThread().getName());
		try{
			Thread.sleep(1000);
		}
		catch(Exception e){
		}
		System.out.println("Ticket was booked by: "+name+"....."+Thread.currentThread().getName());
	}
}
public class MyThread{
	public static void main(String[] args) throws InterruptedException{
		BookTicket bt1 = new BookTicket("ram");
		//new Thread(bt1).start();
		BookTicket bt2 = new BookTicket("sam");
		//new Thread(bt2).start();
		BookTicket bt3 = new BookTicket("kiran");
		//new Thread(bt3).start();
		BookTicket bt4 = new BookTicket("varun");
		//new Thread(bt4).start();
```

```
		BookTicket bt5 = new BookTicket("suji");
		//new Thread(bt5).start();
		BookTicket[] bt11 = {bt1,bt2,bt3,bt4,bt5};
		ExecutorService es = Executors.newFixedThreadPool(3);
		for(BookTicket bt12: bt11){
			es.submit(bt12);
		}
		es.shutdown();
		/*BookTicket[] bt1 = {new BookTicket("ram"),new
BookTicket("sam"),new BookTicket("kiran"),
		new BookTicket("varun"),new BookTicket("suji")};
		ExecutorService es = Executors.newFixedThreadPool(3);
		for(BookTicket bt2: bt1){
			es.submit(bt2);
		}
		es.shutdown();*/
	}
}
```

==========

```
package threads;
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 BookTicket implements Callable{


	int count;
	BookTicket(int count){
		this.count = count;
	}
	@Override
	public Object call() throws Exception {
		int sum=0;
		System.out.println(Thread.currentThread().getName()+
				": is trying to calculate sum ");
		for(int i=0;i<=count;i++){
			sum=sum+i;
			i++;
		}
		return sum;
	}
}
public class MyThread{
	public static void main(String[] args) throws InterruptedException,
ExecutionException{
		BookTicket bt1 = new BookTicket(10);
		BookTicket bt2 = new BookTicket(20);
		BookTicket bt3 = new BookTicket(30);
		BookTicket bt4 = new BookTicket(40);
```

```
            BookTicket bt5 = new BookTicket(50);
            BookTicket[] bt11 = {bt1,bt2,bt3,bt4,bt5};
            ExecutorService es = Executors.newFixedThreadPool(3);
            for(BookTicket bt12: bt11){
                  Future f =es.submit(bt12);
                  System.out.println(f.get());
            }
            es.shutdown();
      }
}
==================================
class ThreadLocalDemo{
      ThreadLocalDemo(){
            ThreadLocal tl = new ThreadLocal();
            System.out.println(tl.get());
            tl.set(123);
            System.out.println(tl.get());
            tl.remove();
            System.out.println(tl.get());
      }
}
class ThreadLocalDemo1{
      ThreadLocalDemo1(){
            ThreadLocal tl = new ThreadLocal(){
```

```
                public Object initialValue(){
                        return "ram";
                }
        };
        System.out.println(tl.get());
        tl.set("sam");
        System.out.println(tl.get());
        tl.remove();
        System.out.println(tl.get());
    }
}
public class MyThread{
    public static void main(String[] args){
        new ThreadLocalDemo();
        new ThreadLocalDemo1();
    }
}
================================
class ThreadLocalDemo extends Thread{
    static Integer id=0;
    ThreadLocal tl = new ThreadLocal(){
                protected Object initialValue(){
                        return ++id;
                }
```

```
        };
        ThreadLocalDemo(String name){
                super(name);
        }
        public void run(){
                System.out.println(Thread.currentThread().getName()+"  id  is:
"+tl.get());
        }
}
public class MyThread{
        public static void main(String[] args){
                ThreadLocalDemo tld1 = new ThreadLocalDemo("ram1");
                ThreadLocalDemo tld2 = new ThreadLocalDemo("ram2");
                ThreadLocalDemo tld3= new ThreadLocalDemo("ram3");
                ThreadLocalDemo tld4 = new ThreadLocalDemo("ram4");
                tld1.start();
                tld2.start();
                tld3.start();
                tld4.start();
        }
}
```