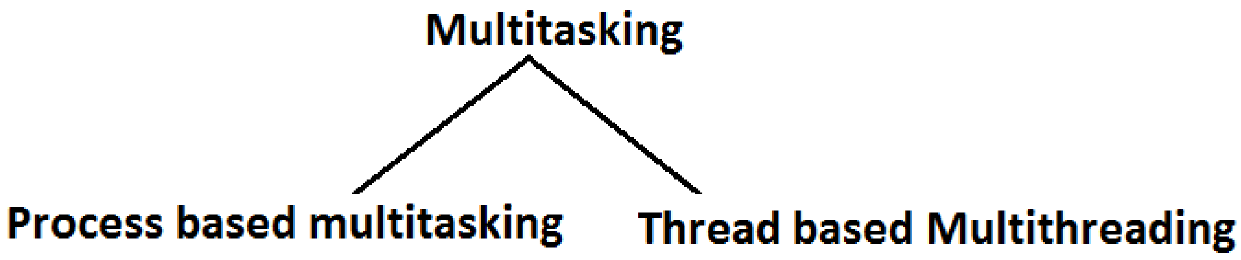
THREADS

*__Single tasking__: Executing a single task at a time is called as single tasking. Here much of the processor time is wasted.
__Ex__: DOS
*__Multi tasking__: Executing multiple tasks at the same time is called as multitasking. Here the processor time is utilized in an optimum way. This multitasking will improve the performance by reducing the response times. Executing several tasks simultaneously is the concept of multitasking. There are two types of multitasking's.
__Ex__: windows



*__Time Slice__: It is a small amount of processor time given to a process for execution. Multitasking is of two types. They are given below
1. Process based multitasking
2. Thread based multitasking
1. __Process based multitasking__: Executing different processes simultaneously at the same time which are independent of each other and every process contains its own memory. This multitasking is an operating system approach.
__Ex__: writing java program, down loading s/w.
Listen to music, copying s/w etc.
2.__Thread based multitasking__: Executing different parts of the same process simultaneously at the same time. Where those different parts have common memory, which may be dependent of independent. This multitasking is a programmatic approach.
__Ex__: Games, web applications
*__Multithreading__: Executing multiple threads at the same time is a called as multi threading or thread based multitasking.
*__Thread__: A separate piece of code which is executed separately is called as thread. Program to get currently executing thread information.
__Ex__:
```
class ThreadInfo{
    Public static void main(String[] args){
    Thread t = thread.currentThread();
    System.out.println("Thread:" +t);
            }
      }
```

$$\begin{tikzcd} & & \text{T.priority} & \\ \text{Output: Thread Info: Thread[} & \text{main,} \arrow[d] & 5, \arrow[u] & \text{main]} \arrow[d] \\ & \text{T.name} & & \text{T.groupname} \end{tikzcd}$$

Current thread method will provide the information of the currently executing thread and it provides information like thread name, thread priority and the thread group name.
Every java program by default contains one thread called as main thread.
The user thread can be created in two ways.

1. By extending Thread class
2. By implementing Run able interface

1.**By extending Thread class**:
***Procedure**:
1.Create a class as sub class thread class.
Syntax: class Myclass extends Thread
2.Write the functionally of user thread with in the run method.
Syntax: public void run(){}
3.Create the object of the class that is extending Thread class
Syntax: Myclass mc = new Myclass();
4.Attach the above created object to the thread calss
Syntax: Thread t = new Thread(mc);
5.Execute the user thread by invoking start();
Syntax: t.start();

```
//User Thread(extending)
class ThreadDemo extends Thread{
//functionality of user Thread
Public void run(){
for(int i= 1; i<=10; i++){
System.out.println(“user Thread:” +i);
        }
}
public static void main(String[] args){
//creating the object
ThreadDemo td = new ThreadDemo();
//attaching the user thread
Thread t = new Thread(td);
//executing the user thread
t.start();
        }
}
```

<u>Difference between t.start() and t.run() methods.</u>

- In the case of t.start() a new Thread will be created which is responsible for the execution of run() method.
- But in the case of t.run() no new Thread will be created and run() method will be executed just like a normal method by the main Thread.

2.**<u>By implementing Run able interface</u>**:
***<u>Procedure</u>**:
1.Create a class implementing Run able interface.
<u>Syntax</u>: class Myclass implements Run able
2.Write the functionality of user thread with in the run method.
<u>Syntax</u>: public void run(){}
3.Create the object of class implements Run able interface.
<u>Syntax</u>: MyClass mc = new MyClass();
    MyClass mc = new Myclass();
4.Attach the above created object to the thread class
<u>Syntax</u>: Thread t = new Thread(mc);
5.Execute the user thread by invoking start().
<u>Syntax</u>: t.start();

```
//user implementing run able interface
class RunableDemo implements Runable{
//functionality of user Thread
public void run(){
for(int i = 1; I <= 15; i++){
System.out.println("user Thread:" +i);
        }
}
Public static void main(String[] args){
//creating the object
RunnableDemo rd = new RunableDemo();
//attaching the user thread
Thread t = new Thread(rd);
//executing the user thread
t.start();
        }
}
```

*****<u>Different b/w extends Thread & implementsRunable</u>**:
When we create thread by extending thread class we do not have a chance to extend from another class.

When we create thread by implementing Runable we have a chance to extends from another class.
Note: It is recommended to use implements Runable to create a ‘thread’.

```
// creating a user multiple Threads acting on multiple objects.
class MultiThread implements Runnable{
String name;
MultiThread(String name){
this.name = name;
        }
Public void run(){
for(int i = 1; i <= 10; i++){
System.out.println(“name value:” + i );
                }
        }
}
// un other program
class MultiThreadDemo{
public static void main(String[] args){
MultiThread mt1 = new MultiThread(“Thread1”);
MultiThread mt2 = new MultiThread(“Thread2”);
Thread t1 = new Thread(mt1);
Thread t2 = new Thread(mt2);
t1.start();
t2.strat();
for(int i = 1; i <= 10; i++){
System.out.println(“main value:” + i);
                }
        }
}
// Program creating multiple Threads
class College implements Runnable{
int seats’
college(int seats){
this.seats = seats;
        }
public void run(){
Thread t = Thread.currentThread();
String name = t.getName();
System.out.println(name + “No.of seats before allotment” + seats);
if(seats > o){
try{
Thread.sleep(2000);
System.out.println(“seat allotted to :” + name);
Seats = seats -1;
```

```
        }
catch(Interrupted Exception ie){
ie.printStackTrace();
        }
}
else{
System.out.println("seat not allotted to: " + name); }
System.out.println(name + "No.of seats after allotment:" +seats);        }
}
class Allotment{
public static void main(String[] args){
College c = new College(60);
Thread t1 = new Thread(c);
Thread t2 = new Thread(c);
t1.setName("student1");
t2.setName("student2");
t1.start();
t2.start();
        }
}
```

When multiple threads are acting on the same object there is a chance of data inconsistency problem occurring in the application.
Data inconsistency problem occurring when one of the thread is updating the value when other thread is truing to read the value at same time.
To avoid the data inconsistency problem we have to the synchronies the threads that are the acting on the same object.
**<u>Thread Synchronization</u>: When multiple threads wants to access the object at the same time avoiding multiple threads to access the same and giving access to one of the thread is called as thread synchronization. Thread synchronization can be done into two ways.

1. Synchronized Block
2. synchronized Method

1.**<u>Synchronized Block</u>**: Synchronizing a group of statements or part of a code is called as Synchronized Block.
<u>Syntax</u>: Synchronized(object){
            Statements;   }
2.**<u>Synchronized Method</u>**: when we want to Synchronized all the statements in a method we go for synchronized method.
<u>Syntax</u>: Synchronized returnType methodName(){
            Statements;
                    }

Note: In the previous program multiple threads acting on the same object leading to data in consistency, to avoid the data inconsistency problem. We have to synchronize the threads acting on the same object.

Ex: public Synchronized void run(){
        Same code previous programs;
                }

When multiple threads are acting on synchronized objects then there is chance of other problems like 'Deadlock' occurring in the application.

***Deadlock**: When a thread holds a resource and waits for another resource to be realized by second thread, the second thread holding a resource and waiting for realized by first thread. Both the threads will be waiting in defiantly and they never execute this switching is called as "Deadlock".

In java there is no mechanism to avoid deadlock situation, it is the responsibility of the programmer to write proper logic to avoid deadlock situation.

***Creation of a Thread**:

Syntax: Thread t = new Thread();

The above syntax will create a thread having default names. The default names will be Thread – 0, Thread – 1, Thread – 2, ………………..

Syntax: Thread t = new Thread(String name);

        The above syntax will create a thread with the specified name.

Syntax: Thread t = new Thread(Object obj);

The above syntax will create a thread which is attached to the specified object.

Syntax: Thread t = new Thread(Object obj, String name);

        The above syntax will create a thread with the specified name and attached to the specified object.

***Methods of Thread class**:

1. **Current_Thread():** This method is used to provide the information of currently executing Thread.
2. **Start():** This method is used to execute the user thread, that is use to execute the logic of Run method.
3. **Sleep(milli seconds):** This method is used to suspend the execution of a thread for amount of time specified. This method throws and exception called interrupted exception which must be handeled.
4. **getName():** This method returns the name of the thread.
5. **SetName(String name):** This method used to assigned a name to a thread.
6. **getpriority():** This method returns the priority of a thread.
7. **Set priority():** This method is used to change the priority of a thread. When we want to change the thread priority it is always recommended to take the support of the constant declared in the thread class.

        * MIN_PRIORITY
        *NORM_PRIORITY
        *MAX_PRIORITY

Ex: t.SetPriority(8);
     t.SetPriority(Thread.MAX_PRIORITY-2);   //(recommended)

8. **iSAlive():** This method returns true if the thread is true. Otherwise is false. A thread is said to be Alive as long as Thread is executing “run()”.
9. **join():** This method is used to make a thread wait until another thread dice.

*****Methods of object class related to Threads**:
1.**wait()**: This method is used to suspend the execution of a thread until it receives a notification.
2.**notity()**: This method is used to send a notification to one of the waiting threads.
3.**notifyAll()**: This method is used to send a notification to All the waiting threads.
Note: The above three methods are used for making communication among the threads.
***Types of Threads**:
1.**Orphan Thread**: A thread which is executed without the help of a parent is called as Orphan Threads. Orphan threads can be created ‘join()’.
2.**Helper Threads**: when multiple threads having a same priority are competing for executing, allowing one of those threads to execute. Depending upon the requirement and the remain threads are called as helper threads. Helper threads give chance for other threads to execute.
3.**Selfish Thread**: A thread which takes lot of resources are execute for longer time periods or until completion are called as Selfish Threads.
4.**Starving Thread**: A thread that is waiting for longer time periods are called as Starving Threads.
5.**Green Thread**:(JVM level treads) These threads are also called as JVM level threads. These threads are used for allocating resource to the user thread. Here the allocation of the resources may not be efficient.
6.**Native threads**: These threads are also called as ‘operating System level threads. These threads are responsible for allocating resource to user threads. Here the allocating of resources of resource is efficient.
7.**Deamon Thread**: These threads are also called as background threads. These threads will execute where no other threads are under execution.

```
//program for inter thread communication
class Showroom{
int value;
boolean flag = true;
public Synchronized void produce(int i){
if(flag == true){
value = i;
System.out.println(“produce value:” +i);
notify();
flag = flase;
        }
try{
```

```
wait();
          }
catch(Interrupted Exception ie){
          }
}
Public Synchronized int consume(){
if(flag == true){
try{
wait();
          }
catch(Interrupted Exception ie){
ie.printStackTrace();
          }
}
notify();
flag = true;
return value;
          }
}//show Room class
class producer extends Thread{
ShowRoom s;
Producer(ShowRoom s){
this.s = s;
          }
Public void run(){
int i = 1;
while(true){
s.produce(i);
i = i + 1;
try{
Thread.Sleep(2000);
          }
catch(Interrupted Exception ie){
System.out.println(ie);
                              }
                    } //while
          } //run
} //producer
class Consumer extends Thread{
ShowRoom s;
Consumer(ShowRoom s){
this.s = s;
          }
Public void run(){
```

```
while(true){
int x = s.consume();
System.out.println("Consumed value:" + x);
try{
Thread.sleep(2000);
        }
catch(interrupted Exception ie){
System.out.println(ie);
                        }
                } //while
        } //run
} // consumer
class producerConsumer{
public static void main(String[] args){
ShowRoom s = new ShowRoom();
Producer p = new Producer();
Consumer c = new Consumer(s);
Thread t1 = new Thread(p);
Thread t2 = new Thread(c);
        t1.start();
        t2.start();
        }
}
```

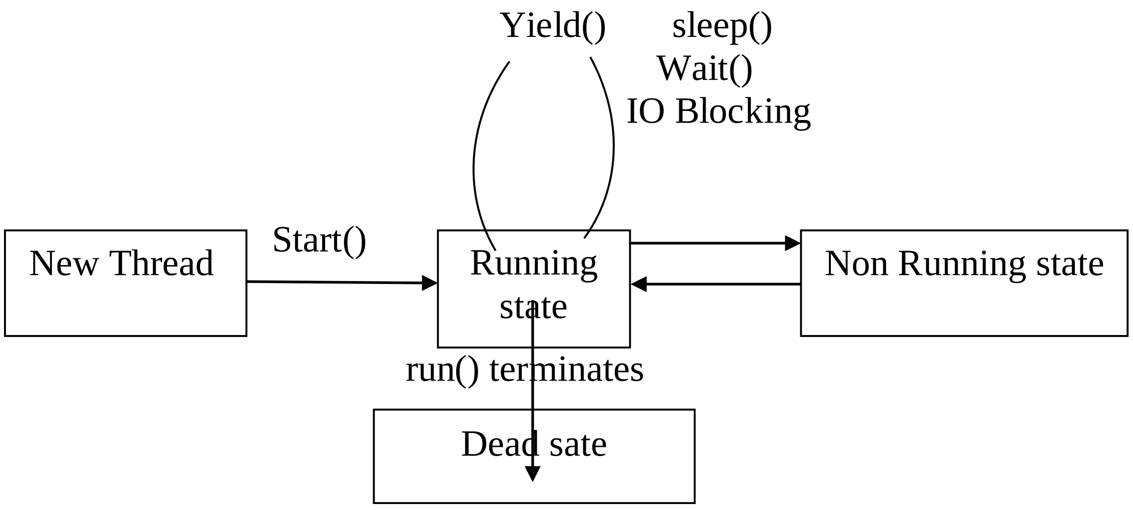
Note: The wait(), the notify() and notifyAll() must be called with in a Synchronized block otherwise we get a run time error called Illegal monitor State Exception.

```
// Another Example
class MyThread extends Thread{
static int total = 0;
public Synchronized void run(){
System.out.println("user thread started calculation");
for(int i = 1; i <= 10; i++){
total = total +i;
        }
System.out.println("user thread sending notification");
notifyAll();
System.out.println("user total = " + total);
        }
Public static void main(String[] args) throws Interrupted Exception{
MyThread mt = new MyThread();
Thread t = new Thread(mt);
System.out.println("main thread calling user thread");
t.start();
Synchronized(mt){
```

```
mt.wait();
        }
System.out.println("main thread got notification");
System.out.println("main Total = " + mt.total);
        }
}
```

The wait(), notify() and notifyAll() are available in Object class. So that we can use those methods directly in our logic to make communication without the reference of thread class.

<u>**THREAD LIFE CYCLE**</u>


Yield()
sleep()
Wait()
IO Blocking
New Thread
Start()
Running state
Non Running state
run() terminates
Dead sate