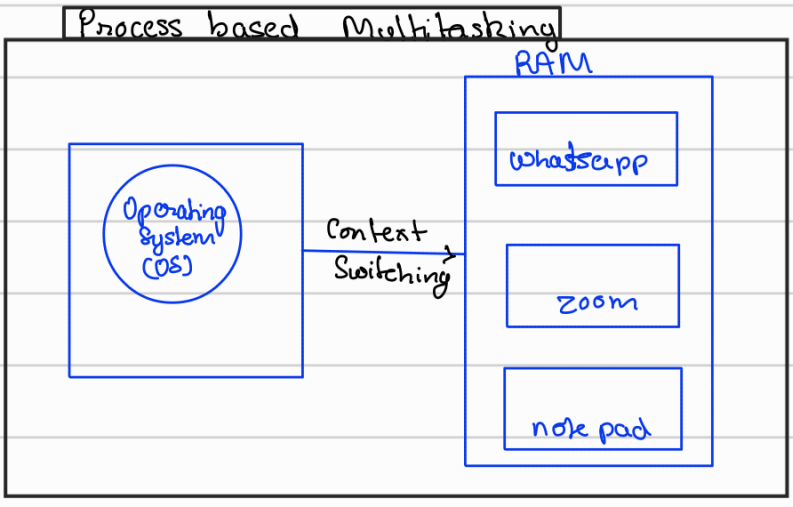
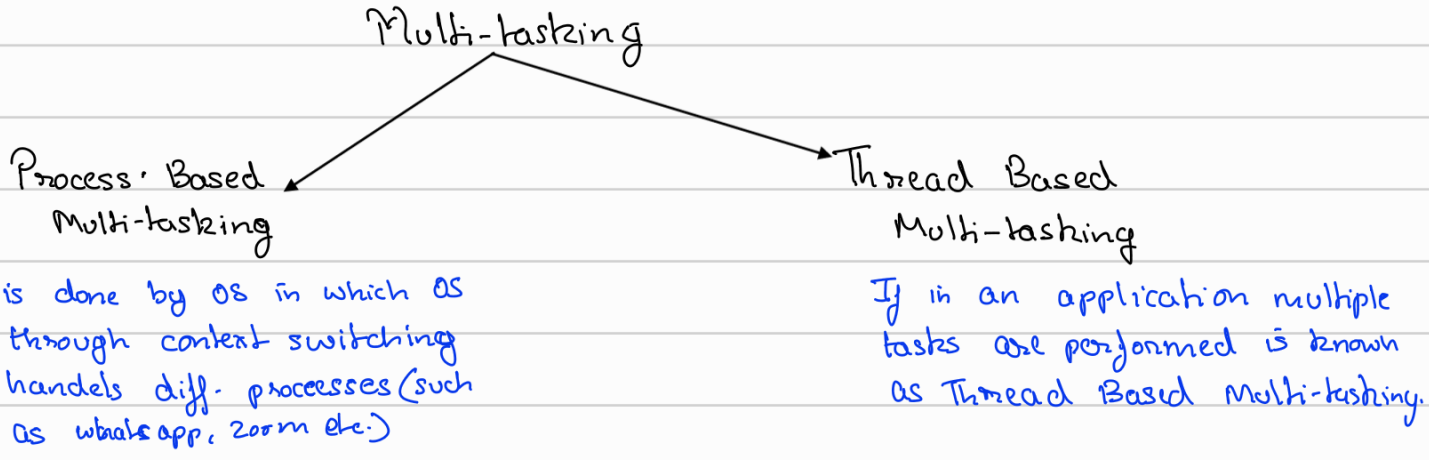
Task :- it is an Activity or a piece of work

#=> Task is executed by the processor on ram i.e 1st the task is loaded into ram then the processor executes the task.





is done by OS in which OS through context switching handels diff. processes (such as whatsapp, zoom etc.)

If in an application multiple tasks are performed is known as Thread Based Multi-tasking.

List of Concepts Involved:

- What is an Operating System?
- Need of Multiple thread
- What is Thread?
- How to create Threads
- run() method
- Multiple task within single run()
- Different states of Thread
- synchronised keyword
- Deadlock
- Producer - Consumer problem

What is an Operating System?

It is a system software which runs in the background and helps the user to run other applications.

MultiTasking

Executing several tasks simultaneously is the concept of multitasking.
There are 2 types of Multitasking.
a. Process based multitasking
b. Thread based multitasking.

Process based multitasking

Executing several tasks simultaneously where each task is a separate independent process such type of multitasking is called "**process based multitasking**".
Example
typing a java program
listening to a song
downloading the file from internet
Process based multitasking is best suited at "os level".

Thread based multitasking

Executing several tasks simultaneously where each task is a separate independent part of the same Program, is called "Thread based MultiTasking".
Each independent part is called a "Thread".

1. This type of multitasking is best suited at "Programmatic level".
The main advantages of multitasking is to reduce the response time of the system and to improve the performance.

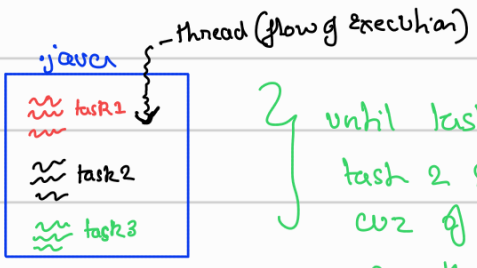
2. The main important application areas of multithreading are
   a. To implement multimedia graphics
   b. To develop web application servers
   c. To develop video games

3. Java provides inbuilt support to work with threads through API called
   Thread,Runnable,ThreadGroup,ThreadLocal,...

4. To work with multithreading, java developers will code only for 10% remaining
   90% java API will take care..

Java provides a line of execution / flow of execution also c/a Thread for execution of a program.

Suppose there is a java program containing 3 independent task so having only one flow of execution (thread) will prevent the other independent task from executing as the thread will be busy executing the 1st task, wasting CPU's time.



} until task 1 is finished executing task 2 & task 3 will not be executed cuz of only one flow of execution an thus wasting CPU's timecycle.

Main Thread JAVA :-

main thread is created by default in java.

default name of main thread : main

default priority of main thread : 5

WAP to print the current thread name and priority and also change the main thread's name & priority

Changin name and priority of current thread using inbuit class Thread and its inbuit methods

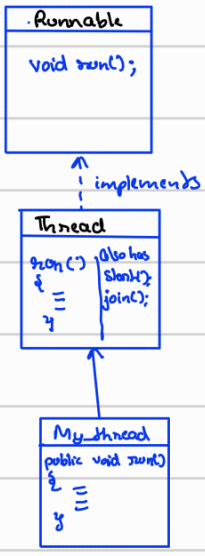
```java
public class Multithreading2 {
    Run | Debug
    public static void main(String[] args) {
        System.out.println("Main Thread");
        System.out.println("Before change: name of current thread= "+Thread.currentThread().getName());
        //or
        // String name  = Thread.currentThread().getName();
        // System.out.println(name);



        //Priority can also be given in case we are using more than 1 thread
        //Priority of main thread
        System.out.println("Before changing: Main Thread Priority= "+Thread.currentThread().getPriority());


        System.out.println("Changing main thread name and Priority");
        //create Thread class object
        Thread t = Thread.currentThread();
        t.setName(name:"Thread1");
        t.setPriority(newPriority:1);
        System.out.println("name and priority changed");

        System.out.println("After change: name of current thread= "+Thread.currentThread().getName());
        System.out.println("After change: Main Thread Priority= "+Thread.currentThread().getPriority());

    }
}
```

How to create Threads?

There are 2 ways for creating a Thread

1. By extending Thread Class (write your own class and extend it to Thread class)
2. By implementing runnable interface

Runnable Interface in Java :- it is an inbuilt interface inside java the contains run() abstract method. So the inbuilt Thread class implements this interface and overrides the run method



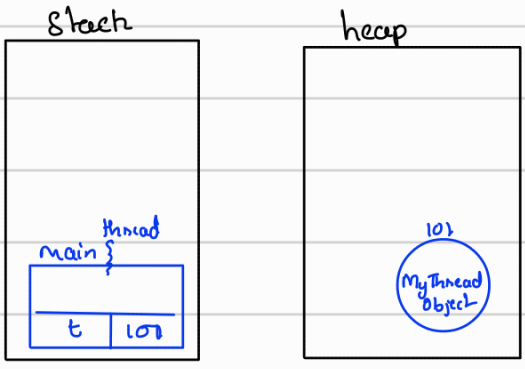
① Creating thread by extending Thread class :-

```java
/*Creating a Thread by extending the Thread class */
class MyThread extends Thread{
    public void run()
    {
        System.out.println("Child Thread");
        System.out.println(Thread.currentThread().getName());
    }
}
public class Multithreading3 {
    Run | Debug
    public static void main(String[] args) {
        System.out.println("Main Thread");
        System.out.println(Thread.currentThread().getName());

        MyThread t = new MyThread();
        t.start();
    }
}
```

execution :- When the execution starts the main method's stack frame will come inside stack area along with a thread. the will start executing line by line then, an object is create for MyThread class in the heap area. and its reference will be stored by a variable 't' inside stack area.

Now t.start() method registers the thread with a "Thread Scheduler" inside JVM. any thread weather main of custom thread it

is managed by thread scheduler. After registering the thread through start() method the Thread Scheduler will automatically calls the run() method in MyThread class. (so no need to call run method in main block)




```
import java.util.Scanner;
/**
 * Multithreading4:
 * Creating a Thread by extending the Thread class 2
 */

class Calc extends Thread{
   public void run()
   {
      System.out.println(Thread.currentThread().getName()+"starting");
      try (Scanner scan = new Scanner(System.in)){
         System.out.print("Enter A= ");
         int a = scan.nextInt();     //Until user input is not provided the other independent task
will not be executing as there is only one line of flow(thread) thus wasting CPU's timecycle.
         System.out.print("Enter B= ");
         int b = scan.nextInt();
         int res = a+b;
         System.out.println("A+B= "+res);
      } catch (Exception e) {

         e.printStackTrace();

      }
      System.out.println(Thread.currentThread().getName()+"ended");
   }
}
class Msg extends Thread{
```

```
    public void run()
    {
        System.out.println(Thread.currentThread().getName()+"starting");
        System.out.println("Displaying msg");
        for(int i=0;i<4;i++)
        {
            System.out.println(i+1+") Important msg");
            try {
                Thread.sleep(2000);  //checked Exception so inside try-catch block
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } //sleep for 2 second
        }
        System.out.println("Task3 ended");
        System.out.println(Thread.currentThread().getName()+"ended");
    }
}
class PatternPrint extends Thread{
    public void run()
    {
        System.out.println(Thread.currentThread().getName()+"starting");
        for(int i=0; i<4 ; i++)
        {
            for(int j=0;j<4;j++)
            {
                System.out.print("*");
            }
            System.out.println();
        }
        System.out.println(Thread.currentThread().getName()+"ended");
    }
}
public class Multithreading4 {
    public static void main(String[] args) {
        System.out.println("Main Thread started: "+Thread.currentThread().getName());

        Calc t1 = new Calc();

        Msg t2 = new Msg();

        PatternPrint t3 = new PatternPrint();

        t1.start();
        t2.start();
```

```
        t3.start();

    }
}
```

In above case as classes created extends to Thread class therefore their object creation is same as creating a thread object

But if the the classes implement Runnable class then then we have to create seperate thread objects and pass the class objects as argument

② Creating thread by implementing Runnable class :-

```
class Calc implements Runnable {
    public void run()
    {
        System.out.println(Thread.currentThread().getName()+"starting");
        try (Scanner scan = new Scanner(System.in)){
            System.out.print("Enter A= ");
            int a = scan.nextInt();    //Until user input is not provided the other independent task will not be executing as there is only one line of flow(thread) thus wasting CPU's timecycle.
            System.out.print("Enter B= ");
            int b = scan.nextInt();
            int res = a+b;
            System.out.println("A+B= "+res);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName()+"ended");
    }
}
class msgg implements Runnable{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("Imp Message");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class Multithreading5 {
    public static void main(String[] args) {
        //creating objects of above class
        msgg c1 = new msgg();
        Calc c2 = new Calc();

        //creatingThread class object with class reference as argument
        Thread t1 = new Thread(c1);
        Thread t2 = new Thread(c2);
```

```
        //starting both threads so that Threas Scheduler register them and manage them
automatically
        t1.start();
        t2.start();
    }
}
```

Extend Thread Vs Implements Runnable

① Extends Thread:-

If we extends a class to Thread any object created of that class is similar to creating a thread (i.e Thread class object) but Suppose we have to extend this class to another class it can't be done cause multiple inheritence not possible in extend

② Implements Runnable:-

In this case there is scope of extending and inheriting the class.

∴ Therefore Implements Runnable is the best approach for creating threads cuz using this methods we have a scope of extending the class.

Multiple tasks with single run();

```java
/**
 * Multithreading8: Multiple task within single run()
 * (with extends Thread)
 */
import java.util.*;

class MyThread1 extends Thread{

    public void run()
    {
        String tName=Thread.currentThread().getName();
        if(tName.equals(anObject:"CALC"))
        {
            calc();
        }
        else{
            importantMesg();
        }
    }
    public void calc()
    {
        System.out.println(x:"Calculation Task Started");

        Scanner sc=new Scanner(System.in);
        System.out.println(x:"Please enter first number");
        int num1=sc.nextInt();

        System.out.println(x:"Please enter 2nd number");
        int num2=sc.nextInt();

        int res=num1+num2;

        System.out.println(res);
        System.out.println(x:"Calculation Task Ended");

        System.out.println(x:"**************************************************");
    }
    public void importantMesg()
    {
        System.out.println(x:"Displaying important message task");
        try{
            for(int i=0;i<3;i++)
            {
                System.out.println(x:"Focus is important to master skills");
                Thread.sleep(millis:2000);
            }
        }
        catch(Exception e)
        {
            System.out.println(x:"Some problem");
        }


        System.out.println(x:"Displaying import message task ended");
    }
}
public class Multithreading8 {
    Run | Debug
    public static void main(String[] args) {
        System.out.println(x:"Main Thread started");

        MyThread1 thread1=new MyThread1();

        MyThread1 thread2 =new MyThread1();

        thread1.setName(name:"CALC");
        thread2.setName(name:"PRINT");

        thread1.start();
        thread2.start();
    }
}
```

States of Threads :- A thread is a path of execution in a program that goes through following states of thread. The 5 states are as follows :-

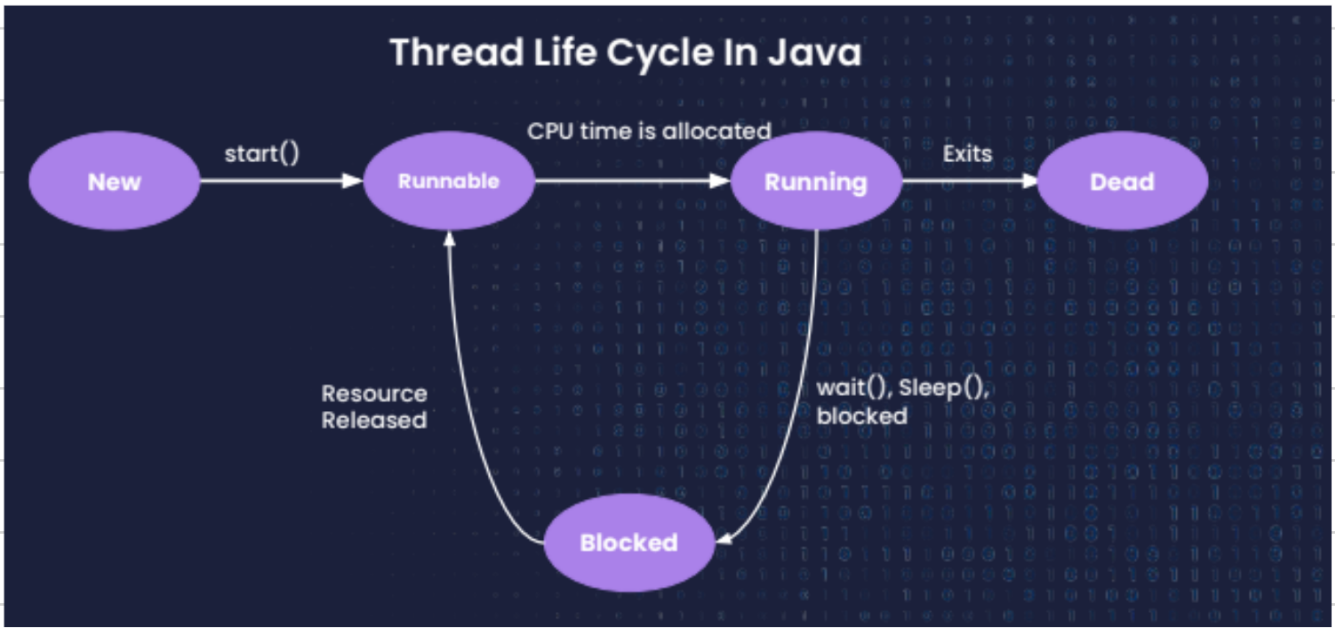
① New // when thread object is created it is said to be in new/born state

② Runnable // when thread Scheduler registers a thread

③ Running // when thread Scheduler executes a thread

④ Blocked (Non-Runnable state) // state where scheduler encounters sleep or join or suspend method

⑤ Dead // thread after executing task.



Q. Explain a method to bring a thread back to Runnable state from the Running state?

Sol^n :→ Yield() method is used for such purpose

join() and isAlive() method :-

```java
import java.util.Scanner;
/*join() and isAlive() method */
class Printing implements Runnable{

    public void run()
    {
        try(Scanner scan = new Scanner(System.in)){
            System.out.print(s:"Enter number of message to print");
            int ip = scan.nextInt();
            for(int i=0;i<ip;i++)
            {
                System.out.println(i+1+") Focus is important");
                Thread.sleep(millis:2000);
            }

        }
        catch(Exception e)
        {
            System.out.println(x:"Some problem");
        }
    }
}

public class Multithreading9 {
    Run | Debug
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()+" Thread Started");

        Printing p1 = new Printing();

        Thread t1 = new Thread(p1);

        System.out.println("is "+t1.getName()+" alive?: "+t1.isAlive());
        t1.start();
        System.out.println("is "+t1.getName()+" alive?: "+t1.isAlive());
        try {
            t1.join(); //Checked Exception: InterruptedException shown which needs to be inside try-catch
            //.join used to wait until t1 thread ends
        } catch (InterruptedException e) {

            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName()+" Thread ended");
    }
}
```

interrupt() method

```java
/**
 * Multithreading10: Thread interruption: interrupt() method
 * Thread is only interrupted when its in Blocked/waiting state (i.e when either sleep() or join() method is envoked)
 */
class JoinisAlive implements Runnable{
    public void run()
    {

        for(int i=0; i<3; i++)
        {
            try{
                System.out.println(i+1+") Focus is imp");
            Thread.sleep(millis:2000);
            }
            catch(InterruptedException e){
                System.out.println("Thread interrupted: "+e.getMessage());
            }

        }

    }
}
public class Multithreading10 {
    Run | Debug
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()+" Thread started");

        JoinisAlive c1 = new JoinisAlive();

        Thread t1 = new Thread(c1);

        t1.start();
        t1.interrupt();
        System.out.println(Thread.currentThread().getName()+"Thread ended");
    }
}
```

Synchronization of Threads

```java
/**
 * Multithreading11 : Synchronization in java
 * using synchronizzed keyword
 */
class Car implements Runnable{
    public void run()
    {
        try{
            System.out.println(Thread.currentThread().getName()+" has entered parking lot.");
            Thread.sleep(millis:2000);
            System.out.println(Thread.currentThread().getName()+" got into the car");
            Thread.sleep(millis:2000);
            System.out.println(Thread.currentThread().getName()+" started to drive the car");
            Thread.sleep(millis:2000);
            System.out.println(Thread.currentThread().getName()+" came back and parked the car");
            Thread.sleep(millis:2000);
        }
        catch(Exception e)
        {
            System.out.println("Some problem: "+e.getMessage());
        }
    }
}
public class Multithreading11 {
    Run | Debug
    public static void main(String[] args) {
        Car c1 = new Car();

        Thread t1 = new Thread(c1);
        Thread t2 = new Thread(c1);
        Thread t3 = new Thread(c1);

        t1.setName(name:"Son-1");
        t2.setName(name:"Son-2");
        t3.setName(name:"Son-3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output

```
Son-1 has entered parking lot.
Son-2 has entered parking lot.
Son-3 has entered parking lot.
Son-3 got into the car
Son-2 got into the car
Son-1 got into the car
Son-2 started to drive the car
Son-1 started to drive the car
Son-3 started to drive the car
Son-3 came back and parked the car
Son-2 came back and parked the car
Son-1 came back and parked the car
PS C:\Users\Euphor\Documents\JAVA\14_MultiThreading>
```

How can 3 diff. persons be driving the same car at the same time

Race condition Problem :- this condition arises when more then 1 threads try to access a single resource and the threads starts racing/competing for their task completion doesn't matter the output.

Ex:

In above case 3 threads are trying to access Car Class object such they they are racing for the task completion ingnoring the output which does not makes sense as How can 3 ppl be driving sigle car at the same time

To solved this "race condition" problem "Synchronized" keyword is used. In a synchronized environment only 1 thread is allowed at a time.

Synchronized keyword applied to a method :-

```java
/**
 * Multithreading11 : Synchronization in java
 * using synchronizzed keyword
 */
class Car implements Runnable{
    synchronized public void run()
    {
        try{
            System.out.println(Thread.currentThread().getName()+" has entered parking lot.");
            Thread.sleep(millis:2000);
            System.out.println(Thread.currentThread().getName()+" got into the car");
            Thread.sleep(millis:2000);
            System.out.println(Thread.currentThread().getName()+" started to drive the car");
            Thread.sleep(millis:2000);
            System.out.println(Thread.currentThread().getName()+" came back and parked the car");
            Thread.sleep(millis:2000);
        }
        catch(Exception e)
        {
            System.out.println("Some problem: "+e.getMessage());
        }
    }
}
public class Multithreading11 {
    Run | Debug
    public static void main(String[] args) {
        Car c1 = new Car();

        Thread t1 = new Thread(c1);
        Thread t2 = new Thread(c1);
        Thread t3 = new Thread(c1);

        t1.setName(name:"Son-1");
        t2.setName(name:"Son-2");
        t3.setName(name:"Son-3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

output

```
Son-1 has entered parking lot.
Son-1 got into the car
Son-1 started to drive the car
Son-1 came back and parked the car
Son-3 has entered parking lot.
Son-3 got into the car
Son-3 started to drive the car
Son-3 came back and parked the car
Son-2 has entered parking lot.
Son-2 got into the car
Son-2 started to drive the car
Son-2 came back and parked the car
```

So here by applying synchro-nized keyword only one thread can access the run() method at a time.

But suppose, we have to synchronize only a particular block of code in a method. for example no need to synchronize "entering parking lot" cuz 3 son can enter it simultaneously synchronizing rest of the block.

```java
/**
 * Multithreading11 : Synchronization in java
 * using synchronizzed keyword
 */
class Car implements Runnable{
    public void run()
    {
        try{
            System.out.println(Thread.currentThread().getName()+" has entered parking lot.");
            Thread.sleep(millis:2000);

            synchronized(this)
            {
            System.out.println(Thread.currentThread().getName()+" got into the car");
            Thread.sleep(millis:2000);
            System.out.println(Thread.currentThread().getName()+" started to drive the car");
            Thread.sleep(millis:2000);
            System.out.println(Thread.currentThread().getName()+" came back and parked the car");
            Thread.sleep(millis:2000);
            }
        }
        catch(Exception e)
        {
            System.out.println("Some problem: "+e.getMessage());
        }
    }
}
public class Multithreading11 {
    Run | Debug
    public static void main(String[] args) {
        Car c1 = new Car();

        Thread t1 = new Thread(c1);
        Thread t2 = new Thread(c1);
        Thread t3 = new Thread(c1);

        t1.setName(name:"Son-1");
        t2.setName(name:"Son-2");
        t3.setName(name:"Son-3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

output

```
Son-1 has entered parking lot.
Son-3 has entered parking lot.
Son-2 has entered parking lot.
Son-2 got into the car
Son-2 started to drive the car
Son-2 came back and parked the car
Son-1 got into the car
Son-1 started to drive the car
Son-1 came back and parked the car
Son-3 got into the car
Son-3 started to drive the car
Son-3 came back and parked the car
```

☆ Synchronized driving and parking.

Dead Lock()

Dead Lock is a scenario in java where 2 or more threads are stuck in Blocked state cuz each one is busy acquiring a resource in a synchronized state which the other threads want to access

Example:- Suppose there are two threads t1 , t2 and 3 resources r_1, r_2, r_3. Now whichever threads goes into running state will aquire a resource and lock it in a synchronized enviornment. Suppose t1 get a chance into running state it locks r_1 resource in a synchronized enviornment here it encounters sleep or join method So in order to utilize CPU's timecycle efficiently in that moment Thread Scheduler will schedule t_2 and t_2 will lock r_3 and starts executing. Now suppose after finishing r_1 t_1 aquires r_2 and encounters block there ; and if t_2 also encounters bloch state and again both comes back to running and try to aquire each other state like t_2 trying to aquire r_2 and t_1 trying to aquire r_3 but it will not happen cuz t_1 has locked r_2 and t_2 has locked r_3. due to this they will be permanently stuck in blocked state because:-

Threads only comes out of Blocked/waiting state only if they aquire a resource to execute but, if all the resources are already aquired it will be stuck in Blocke-state unhill a resource is available.

```java
/**
 * Multithreading12: Dead Lock
 */
class Book implements Runnable
{
    String res1 = new String(original:"Book1");
    String res2 = new String(original:"Book2");
    String res3 = new String(original:"Book3");
    public void run()
    {
        String name = Thread.currentThread().getName();
        if(name.equals(anObject:"Student1"))
        {
            try {
                Thread.sleep(millis:3000);
                synchronized(res1){
                    System.out.println(name+" aquired "+res1);
                    Thread.sleep(millis:3000);
                    synchronized(res2){
                        System.out.println(name+" aquired "+res2);
                        Thread.sleep(millis:3000);
                        synchronized(res3){
                            System.out.println(name+" aquired "+res3);
                            Thread.sleep(millis:3000);
                        }
                    }
                }
            } catch (Exception e) {
                // TODO: handle exception
                System.out.println(x:"Something went wrong");
            }
        }
        else{
            try {
                Thread.sleep(millis:3000);
                synchronized(res3){
                    System.out.println(name+" aquired "+res3);
                    Thread.sleep(millis:3000);
                    synchronized(res2){
                        System.out.println(name+" aquired "+res2);
                        Thread.sleep(millis:3000);
                        synchronized(res1){
                            System.out.println(name+" aquired "+res1);
                            Thread.sleep(millis:3000);
                        }
                    }
                }
            } catch (Exception e) {
                // TODO: handle exception
                System.out.println(x:"Something went wrong");
            }
        }
    }
```

```java
public class Multithreading12 {
    Run | Debug
    public static void main(String[] args) {
        Book b = new Book();

        Thread t1 = new Thread(b);
        Thread t2 = new Thread(b);

        t1.setName(name:"Student1");
        t2.setName(name:"Student2");

        t1.start();
        t2.start();
    }
}
```

```
Student2 aquired Book3
Student1 aquired Book1
Student1 aquired Book2
```

t_1 waits 3 sec
t_2 locks res 3

t_2 waits 3sec

t_1 locks res 1

t_1 waits 3sec

t_1 locks res 2

t_2 fails to aquire res2
remain in Blocked state

t_1 fails to aquire res 3
cuz t_2 has lock on
res 3 ∴ remains in
Blocked state

=> Dead lock occurred

What is thread?

A. Separate flow of execution is called "Thread".
if there is only one flow then it is called "SingleThread" programming.
For every thread there would be a seperate job.

B. In java we can define a thread in 2 ways
a. implementing Runnable interface
b. extending Thread class

ThreadScheduler

If multiple threads are waiting to execute,then which thread will execute 1st is decided by ThreadScheduler which is part of JVM.
In the case of MultiThreading we can't predict the exact output, only possible output we can expect.
Since jobs of threads are important,we are not interested in the order of execution; it should just execute such that performance should be improved.

diff b/w t.start() and t.run()

if we call t.start() and seperate thread will be created which is responsible to execute the run() method.
if we call t.run(), no separate thread will be created, rather the method will be called just like a normal method by main thread.

Importance of Thread class start() method

For every thread, required mandatory activities like registering the thread with Thread Scheduler will be taken care by Thread class
start() method and the programmer is responsible for just doing the job of the Thread inside run() method.
start() acts like an assistance to programmers.

```
start()
   {
register thread with ThreadScheduler
          All other mandatory low level activities
 invoke or call the run() method.
   }
```

We can conclude that without executing the Thread class start() method there is no chance of starting a new Thread in java.
Due to this, start() is considered the **heart of MultiThreading.**

If we are not overriding run() method

If we are not Overriding run() method then Thread class run() method will be executed which has an empty implementation and hence we won't get any output.

eg::

```
class MyThread extends Thread{}
class ThreadDemo{
 public static void main(String... args){
  MyThread t=new MyThread();
  t.start();
 }
}
```

Overloading of run() method

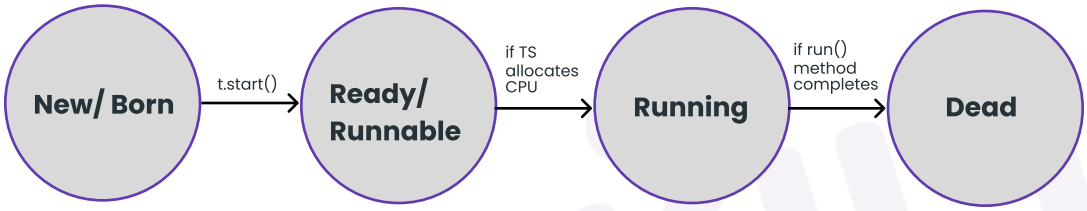
We can overload the run() method but Thread class start() will always call run() with zero argument.
if we overload run method with arguments, then we need to explicitly call argument based run method and it will be executed just like normal method.

Life cycle of a Thread

- if Thread scheduler allocates CPU time then we say thread entered into Running state.
- if run() is completed by thread then we say thread entered into dead state.
- Once we create a Thread object then the Thread is said to be in a new state or born state.
- Once we call start() method then the Thread will be entered into Ready or Runnable state.
- If Thread Scheduler allocates CPU then the Thread will be entered into running state.
- Once the run() method completes then the Thread will enter into dead state.

Life Cycle Of Thred

MyThread t=new My Thread();



Different approach for creating a Thread?

A. extending Thread class
B. implementing Runnable interface

Which approach is the best approach?

- implements Runnable interface is recommended becoz our class can extend another class through which inheritance benefit can be brought into our class.
- Internally performance and memory level is also good when we work with interfaces.
- if we work with extended features then we will miss out inheritance benefit because already our class has inherited the feature from "Thread class", so we normally don't prefere extended approach rather implements approach is used in real time for working with "MultiThreading".

Various Constructors available in Thread class

Thread t=new Thread()
Thread t=new Thread(Runnable r)
Thread t=new Thread(String name)

Names of the Thread

Internally for every thread,there would be a name for the thread.
a. name given by jvm
b. name given by the user.

ThreadPriorities

- For every Thread in java has some priority.
- The valid range of priority is 1 to 10,it is not 0 to 10.
- if we try to give a different value then it would result in "IllegalArgumentException".
- Thread.MIN_PRIORITY = 1
- Thread.MAX_PRIORITY = 10
- Thread.NORM_PRIORITY = 5
- If both the threads have the same priority then which thread will get a chance as a program we can't predict becoz it is vendor dependent.

We can set and get priority values of the thread using the following methods
a. public final void setPriority(int priorityNumber)
b. public final int getPriority()
The allowed priorityNumber is from 1 to 10,if we try to give other values it would result in **"IllegalArgumentException".**
System.out.println(Thread.currentThread().
setPriority(100);//IllegalArgumentException.

DefaultPriority

The default priority for only the main thread is "5",whereas for other threads priority will be inherited from parent to child.
Parent Thread priority will be given as Child Thread Priority.

We can prevent Threads from Execution
a. yield()
b. sleep()
c. join()

yield()

- It causes the current executing Thread to give a chance for waiting Threads of the same priority.
- If there is no waiting Threads or all waiting Threads have low priority then

the same Thread can continue its execution.

- If all the threads have the same priority and if they are waiting then which thread will get a chance we can't expect, it depends on ThreadScheduler.
- The Thread which is yielded, when it will get the chance once again depends on the mercy of "ThreadScheduler" and we can't expect exactly.

public static native void yield()

join()

If the thread has to wait until the other thread finishes its execution then we need to go for join().
if t1 executes t2.join() then t1 should wait till t2 finishes its execution.
t1 will be entered into waiting state until t2 completes, once t2 completes then t1 can continue with its execution.

eg#1.

venue fixing ======> t1.start()
wedding card printing ======> t2.start()======> t1.join()
wedding card distribution ======> t3.start()======> t2.join()

Waiting of Child Thread until Completing Main Thread

We can make the main thread wait for the child thread as well as we can make the child thread also wait for the main thread.

sleep()

If a thread doesn't want to perform any operation for a particular amount of time then we should go to sleep().

Signature

```
public static native void sleep(long ms) throws InterruptedException
public static void sleep(long ms,int ns) throws InterruptedException
```

Every sleep method throws InterruptedException, which is a checked exception so we should compulsorily handle the exception using
try catch or by throws keyword otherwise it would result in a compile time error.

```
Thread t=new Thread(); //new or born state
 t.start() // ready/runnable state
```

=> If T.S allocates cpu time then it would enter into running state.
=> If run() completes then it would enter a dead state.
=> If running thread invokes sleep(1000)/sleep(1000,100) then it would enter into Sleeping state
=> If time expires/ if sleeping thread got interrupted then thread would come back to "ready/runnable state".

synchronization

1. synchronized is a keyword applicable only for methods and blocks
2. if we declare a method/block as synchronized then at a time only one thread can execute that method/ block on that object.
3. The main advantage of synchronized keywords is we can resolve data inconsistency problems.
4. But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and effects performance of the system.
5. Hence if there is no specific requirement then never recommended to use synchronized keyword.
6. Internally synchronization concept is implemented by using lock concept.
7. Every object in java has a unique lock. Whenever we are using synchronized keyword then only the lock concept will come into the picture.
8. If a Thread wants to execute any synchronized method on the given object 1st it has to get the lock of that object. Once a Thread gets the lock of that object then it's allowed to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock.
9. While a Thread executing any synchronized method the remaining Threads are not allowed to execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented based on object but not based on method].

Note: Every object will have 2 area[Synchronized area and NonSynchronized area]
Synchronized Area => write the code only to perform update,insert,delete
NonSynchronized Area => write the code only to perform select operation

```
class ReservationApp{
 checkAvailablity(){
  //perform read operation
        }
 synchronized bookTicket(){
  //perform update operation
 }
}
```

class level lock

1. Every class in java has a unique level lock.
2. If a thread wants to execute static synchronized method then the thread requires "class level lock".
3. While a Thread executing any static synchronized method the remaining Threads are not allowed to execute any static synchronized method of that class simultaneously.
4. But remaining Threads are allowed to execute normal synchronized methods, normal static methods, and normal instance methods simultaneously.
5. Class level lock and object lock both are different and there is no relationship between these two.

synchronized block

```
synchronized void m1(){
 ...
 ...
 ...
 ...
 ...
 ======
 ======
 ======
 ======

 ...
 ...
 ...
 ...
 ...
}
```

if a few lines of code is required to get synchronized then it is not recommended to make the method only as synchronized.
If we do this then for threads performance will be low, to resolve this problem we use "synchronized block", due to synchronized block performance will be improved.

DeadLock

If 2 Threads are waiting for each other forever(without end) such type of situation (infinite waiting) is called dead lock.
There are no resolution techniques for deadlock but several prevention(avoidance) techniques are possible.
Synchronized keyword are the cause for deadlock hence whenever we are using synchronized keyword we have to take special care.

DeadLock vs starvation

- Long waiting of a thread, where waiting never ends is termed "deadlock".
- Long waiting for a thread, where waiting ends at a certain point is called "starvation".

eg:: Assume we have 1cr threads, where all 1cr threads have priority is 10,but one thread is there which has priority 0,now the thread with a priority-0 has to wait for long time but still
it gets a chance,but it has to wait for a long time, this scenario is called "Starvation".

Note: Low priority thread has to wait until completing all priority threads but ends at a certain point which is nothing but starvation.

Daemon Threads

The thread which is executing in the background is called "DaemonThread".
eg: AttachListener,SignalDispatcher,GarbageCollector,....

remember the example of movie
1. producer
2. director
3. music director
4. ....
5. ....
6. ....

MainObjective of DaemonThread

The main objective of DaemonThread, to provide support for Non-Daemon threads(main thread).
eg:: if main threads run with low memory then jvm will call GarbageCollector thread, to destroy the useless objects,so that no of bytes of free memory will be improved with this free memory the main thread can continue its execution.

Usually Daemon threads have low priority,but based on our requirement daemon threads can run with high priority also.
JVM => creates 2 threads
a. Daemon Thread(priority=1,priority=10)
b. main (priority=5)
while executing the main code, if there is a shortage of memory then immediately jvm will change the priority of Daemon thread to 10, so Garbage collector activates Daemon thread and it frees the memory after doing it immediately it changes the priority to 1, so the main thread will continue.

How to check whether the Thread is Daemon or not?

public boolean isDaemon() => To check whether the thread is "Daemon"
public void setDaemon(boolean b) throws IllegalThreadStateException
b=> true,means the thread will become Daemon,before starting the Thread we need to make the thread as "Daemon" otherwise it would result in "IllegalThreadStateException".

What is the default nature of the Thread?

Ans. By default the main thread is "NonDaemon ".for all remaining thread Daemon nature is inherited from Parent to child, that is If the parent thread is "Daemon" then the child thread will become "Daemon" and if the parent thread is "NonDaemon" then automatically the child thread is also "NonDaemon".

Is it possible to change the NonDameon nature of Main Thread?

Ans. Not possible, becoz the main thread starting is not in our hands, it will be started by "JVM".

How to stop a thread ?

As for how to start a method we have a method called t.start().
Similarly to stop a thread we have a method called t.stop().
This method will kill a thread and it makes the thread enter into a dead state.
This is not a good approach to kill a thread, so this method is "deprecated".

CompleteLifeCycle of a Thread

```
MyThread t=new MyThread();//"new or born state"
t.start();//"ready or runnable state"
```

a. if T.S allocates CPU time, then thread will be in "running state".
b. if running thread calls
    1. Thread.yield() it enters into ready/runnable state(it is just pausing)
    2. t2.join()
        t2.join(1000)
        t2.join(1000,100);//Thread enters into waiting state
            a. if t2 finishes the execution
            b. if time expires
            c. if thread gets interruption [it comes back to ready/runnable state]
    3. Thread.sleep(1000,100);
        Thread.sleep(100);//Thread enters into sleeping state
        a. if time expires
            b. if thread gets interruption [it comes back to ready/runnable state]
    4. obj.wait()
        obj.wait(1000);
        obj.wait(1000,100); //Thread enters into waiting state
            a. if thread gets notification
            b. if time expires
            c. if thread gets interruption [it comes back to ready/runnable state]
    5. t.suspend()
            a. it enters into suspended state
                a. if it calls t.resume()
                    it enters into ready/runnable state
    6. t.stop()
            a. it enters into dead state
b. if run() is completed by a thread, then the thread enters into "deadstate".

InterThreadCommunication

Two threads can communicate each other with the help of
a. notify()
b. notifyAll()
c. wait()

notify()=> Thread which is performing updates should call notify(),so the waiting thread will get notification so it will continue with its execution with the updated items.
wait() => Thread which is expecting notification/updation should call
wait(), immediately the Thread will enter into a waiting state.

wait(),notifyAll(),notify() is present in Object class, but not in Thread class why?

- Thread will call wait(),notify(),notifyall() on any type of objects like Student,Customer,Engineer.
- If a thread wants to call wait(),notify()/notifyall() then compulsorily the thread should be the owner of the object otherwise it would result in "IllegalMonitorStateException".
- We say thread to be owner of that object if thread has lock of that object.
- It means these methods are part of synchronized block or synchronized method, if we try to use outside the synchronized area then it would result in a RunTimeException called "IllegalMonitorStateException".
- if a thread calls wait() on any object, then first it immediately releases the lock on that object and it enters into waiting state.
- if a thread calls notify() on any object,then he may or may not release the lock on that object immediately.

Method prototype of wait(),notify(),notifyAll()

1. public final void wait()throws InterruptedException
2. public final native void wait(long ms)throws InterruptedException
3. public final void wait(long ms,int ns)throws InterruptedException
4. public final native void notify()
5. public final void notifyAll()

Difference b/w notify and notifyAll()

notify() => To give notification only for one waiting thread
notifyAll() => To give notification for many waiting thread

=> We can use notify() method to give notification for only one Thread. If multiple Threads are waiting then only one Thread will get the chance and remaining Threads have to wait for further notification. But which Thread will be notified(inform) we can't expect exactly it depends on JVM.
=> We can use notifyAll() method to give the notification for all waiting Threads of a particular object. All waiting Threads will be notified and will be executed one by one, because they require lock.

Note: On which object we are calling wait(), notify() and notifyAll() methods that corresponding object lock we have to get but not other object locks.