11/12/2017

* Multitasking:-

The process of performing multiple task execution simultaneously or concurrently is known as Multitasking.

To perform multitasking, two classifications is given.

1) Multiprocessing
2) Multithreading

1) MultiProcessing:-

Suitable at hardware level / system level.

ex- ① Playing music with vlc media player
② Developing java application with eclipse
③ Printing documents with printer sofware
④ Saving the data in MS Word
⑤ Painting operation in MS paint software

2) Multi Thread-

Suitable at software level. It is a light weight sub process concept.

ex-

Google Chrome
> Youtube
> Gmail
> facebook
> naukri.com

MS Word
> Spell correction
> Starting letter of statement will be capitalization
> Indentation
> Auto save option.

* For Application Execution- ** VVI

Java Lab 5

1) JVM → OS (memory collect/request)

2) OS → JVM (based on Ram size, memory allotment)

3) JVM memory will be created and JVM will initialize.

4) At the JVM startup, it is creating some thread group

   a) Main Thread Group > Main Thread

   b) System Thread Group > Finalizer Thread

5) After starting the main thread and finalizer thread, JVM task is finished and JVM will monitor the execution of java application.

   > Main Thread → Task is to execute the predefined main()

   > Finalizer Thread → Task is to call the finalizer() in GC process.

=> Main Thread task will start-

1) It will verify whether .class file name is there or not.

2) If it is not available, it will throw error.

3) If the class file is available then it will verify whether byte code file format is correct or not by using "BYTE CODE - VERIFIER TOOL".

4) If byte code file format is not correct and not understandable by JVM then it will throw error.

5) In case if byte code file format is correct then it will go for converting byte code to machine code.

before java 7

6) It is perform the class loading of the the "Lab" class to perform the execution by using "class Loader"

7) It will execute the static block by default at the time of class loading if there in the specified class.

8) It will verify whether the main method is there or not in class file. If it is there then that method has to invoke by the specified class name.

9) If main method is not there then it will throw error.

after java 7

6) First of all it will verify whether the main method definition is there or not in .class file

7) If main method is there then it will perform the class loading and if any static block is there then it will be executed first and after that it will invoke the main method by the specified class Name.

8) If main method is not there then it will not perform any class loading and there is no static block execution and it will give error as "Main method doesn't found".

While invoking the main method by the main thread by using specified class name if any input values are there after class name then it will be collected as command line orgument and stores those input value in String Array and it will be considered as the Dynamic input value for the application

Then the main thread will call the main method with String Array as the parameter with set of input values.

The main thread is responsible to execute the main method from starting line to end line

=> Until the main method execution is completed. The main thread will not be destroyed and after the main method completion only the main thread will be destroyed.

=> Anywhere in the application current Thread information can be given by one predefined method in thread class.

```
currentThread();
```

ex.

```
Thread th= Thread.currentThread();
System.out.println(th.getName());
System.out.println(th.getThreadGroup().getName());
```

Command line argument is collected by main thread.

NOTE -

=> Predefined standard main() is not calling by JVM and the main() is called by corresponding class name and initiated by predefined main thread.

=> The predefined thread and those task cannot be modified by the developer and while developing the custom application for multitasking, we have a chance of developing custom threads and we can assign custom task for execution.

=> The predefined main thread has created by JVM and started by JVM and assigned the task to execute predefined main method.

* Developing the custom Thread-

There are two ways to develop the custom thread as per the client requirement to assign custom task to achieve the multitasking of application.

1) By extending Thread class
2) By Implementing Runnable Interface.

1) By extending thread class-

a) Declare a java class with custom Thread Class Name and provide the inheritance relation by extending "Thread" class

b) Override the run() of the thread class in custom thread by following signature.

```
public void run() { }
```

c) Inside the run() provide the logic of the task which the custom thread has to perform.

d) Create the object of the custom thread class and start the thread by calling start() of thread class. (Any thread will perform its task by starting only)

2) By implementing runnable interface-

a) Declare a new java class with custom Thread class name and provide the inheritance relation by implementing Runnable interface.

b) Must and should override the abstract run() of Runnable interface in custom Thread class with following signature.

```
public void run() { }
```

c) Inside the run() provide the logic of the Task which the custom Thread has to perform.

d) Any type of thread will Start the execution by starting the Thread Object using Start().

e) Any custom thread object which is subtype of Runnable interface cannot call start() directly because start() is not a member of Runnable Interface. Then we need to create the predefined Thread class object by taking custom Thread Object as the constructor parameter and then with predefined Thread class Object we can call start().

#NOTE-

Thread is default super class for all types of Threads either custom Threads or predefined Threads

ex-

```
interface I1{
    void m1();
}

class A implements I1{
    public void m1(){}    //-> overriding m1();
    public void show(){}  //-> new method definition.
}

class B implements I1{
    public void m1(){}
}

class Lab{
    public static void main(String[] args){
        System.out.println("Main Started");
        B bref = new B();
        bref.show(); //-> Invalid as there is no inheritance relation
    }                    between A & B.
```

ex-

```
class Lab {
    public static void main (String [] args) {
        MyThread th = new MyThread();
        Thread t1 = new Thread(th);
        t1.show();
    }
}

Class MyThread implements Runnable {
        public void run() {
            System.out.println("Run() ===> MyThread");
        }
}
```

NOTE-

Developing the custom Thread by implementing Runnable interface is recomended option than by extending Thread class because

1) We can achieve multiple inheritance while developing the custom thread by implementing Runnable interface but not with extending Thread class.

2) While developing the custom thread class by implementing the Runnable interface compiler will force you to override the abstract run() signature of Runnable interface but it will not force by extending the Thread class.

3) When we develop the custom thread by ~~extending~~ implementing Runnable interface, there is only one issue we cannot call the start() directly and we need to call by using the predefined Thread class reference but by extending thread class we call start() directly.

NOTE-

=> if we call run method directly, then it will be called by main thread.

=> if we call start method, it will be called by JVM which will internally call run().

* Thread Name, Thread priority, Thread Group name-

Every custom thread will become a child Thread to the predefined main thread under the main thread group.

=> toString implementation of thread class-

```
class Thread {
    public string toString (){
        return "Thread [Thread Name, Thread Priority, Thread Group Name]";
    }
```

So, Thread Name, Thread Priority, Thread GroupName are called as member of Thread.

* ThreadName-

By default JVM provides the custom Thread names as

Thread-intvalue; -> Thread Name.

i.e Thread-0
Thread-1
Thread-2
:

Whenever we need to modify the custom Thread name then there are predefined constructors in Thread class.

public Thread (String); -> by extending Thread class

public Thread (Runnable, String); -> by implementing Runnable interface.

ex-

```
public class Lab{
    public static void main (String [] args){
    MyThread1 t1 = new MyThread1 ("Welcome Thread");
    System.out.println(t1);
    MyThread2 t2 = new MyThread2 ("Loan Thread");
    Thread t = new Thread(t2);        //-> or -> new Thread (t2,"Loan Thread");
    System.out.println(t);
    }
}

class MyThread1 extends Thread{
    public MyThread1 (String tname){
        super (tname);
        start();
    }
    public void run() { }        //-> not mandatory.
}

class MyThread2 implements Runnable{
    public MyThread2 (String tname){
        Thread th = new Thread (this, tname);
        th.start();
    }
    public void run(){
        Thread th = Thread.currentThread();
        System.out.println(th.getName());
    }
}
```

NOTE-

There are predefined method which are used to modify thread name after creating Object

1) getName() -> accessing the name of thread

2) setName(String name) -> modifying the thread name.

13/12/2017

* Thread Priority:-

It is a numerical value which is assigned for each and every thread by default. It is used for making high priority thread which needs to be executed first among the multiple threads.

Internally CPU schedular will use the thread priority value based on priority algorithm.

If CPU schedular is not following priority based algorithm then there is no benifit of Thread priority value.

While assigning the thread priority value for the custom thread, we need to use some constant which are provided in predefined thread class.

1) MIN PRIORITY - 1
2) NORM PRIORITY - 5 (default)
3) MAX PRIORITY - 10

Every custom Thread is the child Thread of predefined main Thread. So JVM has to assign the NORM PRIORITY VALUE to the PREDEFINED MAIN THREAD. Since main thread priority is assigning for the custom thread since it is a child Thread

We can modify the thread priority value to the custom thread by using predefined method.

> getPriority(); → accessing priority value of custom Thread
> setPriority(int val); → modifying priority value of custom Thread

Priority hierarchy:-

t_1	t_2	t_3	t_4	t_5	t_6
3	6	2	8	6	4
↓	↓	↓	↓	↓	↓
4th	2nd	5th	1st	2nd	3rd

based on Object creating order priority decided b/w these two

* ThreadGroup -

It is the predefined class which is used for collecting multiple threads. By default every custom thread which we are creating is placing in the main thread group Since these threads are the child thread to the main thread under Main Thread Group.

There is a chance of placing the custom Thread inside the custom Thread Group.

=> Constructor

```
public Thread (ThreadGroup, String);          //-> extends Thread
public Thread (ThreadGroup, Runnable, String); //-> implements Runnable
public Thread (ThreadGroup, Runnable);
```

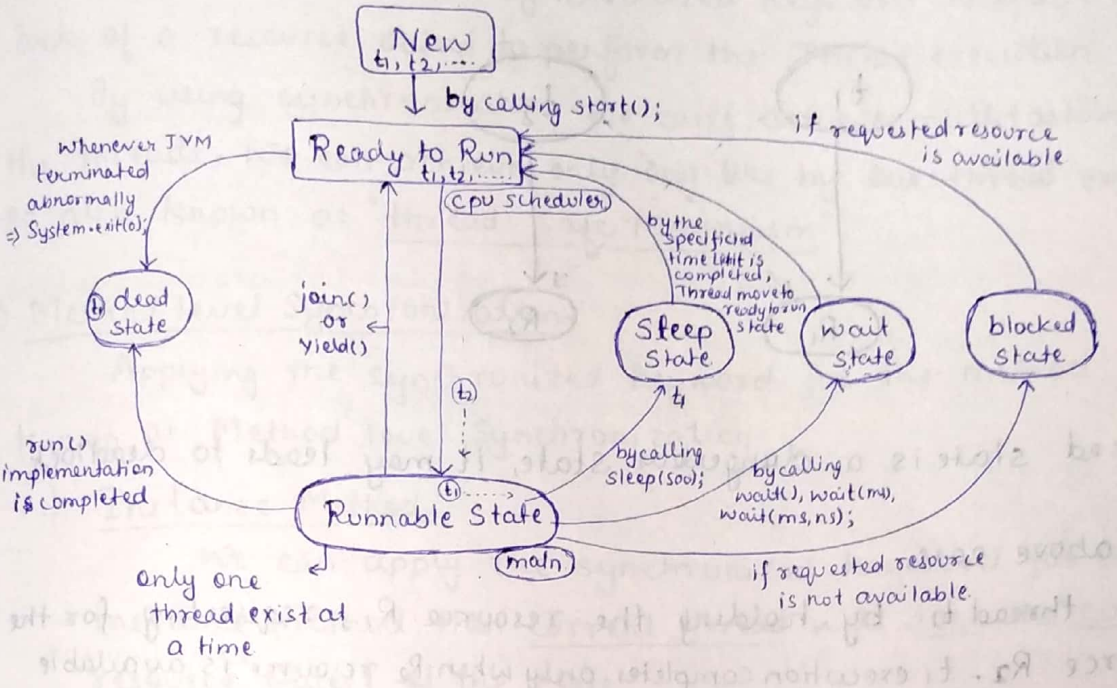
NOTE -

We can modify only ThreadGroup Name for the custom thread by implementing runnable interface by following constructor

ex -

```
class Lab {
public static void main (String[] args){
  ThreadGroup tg = new ThreadGroup ("STUDENT-DB");
  MyThread t = new MyThread (tg);
 }
}

class MyThread implements Runnable{
    public MyThread (ThreadGroup, tgname){
        Thread th = new Thread (tgname, this);
        th.start();
    }
    public void run(){
        Thread th = Thread.currentThread();
        System.out.println(th.getThreadGroup.getName());
    }
}
```

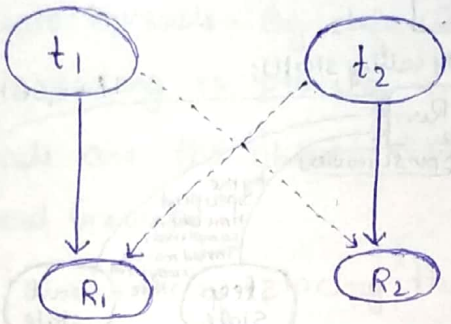
* Life Cycle of Thread -



Statement -

1) When we create a thread, its in new State.
2) When we start the thread, thread will be placed to ready to run state
3) Depending on CPU scheduler algorithm, thread will move from ready to run state to runnable state or vice-versa.
4) When we call sleep method then current running thread will move to sleep state and will stay for specified amount of time (i.e -500 ms).
5) Once the specified time is over then thread will move to ready to run state automatically.
6) Sleep state is also called as "timed_waiting" state.
7) When we call wait method, then current thread will move to wait state.
8) Thread in wait state will move to ready to run state when:
   (i) we calle notify() or notifyAll().
   (ii) When the specified time is over.
9) When a running thread is requesting for unavailable resources then it will moved to blocked state.
10) Thread in blocked state will move to ready to run state when the resource is available.
11) When we call stop() or destroy() method on running thread, then thread will move to dead state.
12) When thread task is completed i.e run() method execution is over then also thread move to dead state.
13) Dead state is also called as "Terminated state"
14) Once the thread is dead, it will not come back to utilize CPU time

* Deadlocks-



⇒ Blocked state is a dangerous state, it may leads to deadlock

In above case,

thread t_1, by holding the resource R_1, requesting for the resource R_2. t_1 execution completes only when R_2 resource is available Similary, thread t_2, by holding the resource R_2, requesting for the resource R_1, t_2 execution completes only when R_1 resource is available

So, here both processes are requesting for unavailable resources, both will placed in blocked state and will never come out of blocked state. This situation is called as Deadlock.

* Synchronization- 14/12/2017

We need to use synchronized keyword to enable the lock of a resource object to perform the Thread execution.

By using synchronization we can't achieve multitasking of the threads. We can achieve only only One by one thread execution so also known as "thread safe Mechanism"

=> Method level Synchronization-

Applying the synchronized keyword for the method is known as Method level Synchronization.

1) Instance Method-

We can apply the synchronized keyword for the instance method then current thread will lock the "current resource" Object of the class.

2) Static Method-

We can apply the synchronized keyword for the static method then current will lock the "default Object" of class (i.e default Object created by JVM durring class loading of java.lang.Class)

=> Block level Synchronization- (Recommendable)

Applying the synchronized keyword for the local blocks.

ex.

```
class Hello{
    void show(){
        synchronized (whichObject) { }
    }
}
```

Note.

- synchronized (this){} → current Object
- synchronized (Hello.class){} → default Object of class
- String Str = "JLC"; →
- Synchronized (Str){} → third party object
- Synchronized (this.getClass()){} → default object of class.

NOTE -

1) In the case of thread execution, synchronized keyword is not there then thread doesn't verify about the looking of resources and it will directly use the resources Object & perform its task or execution.

2) In the case of thread execution, if synchronized keyword is there then thread will check for lock of resource if it is available then it will perform execution or else if resource lock is not available then thread will move to blocked state until the resource availability

3) In the case of inheritance synchronization, the current Object will be locked by the thread then, if multiple thread are using same resource then it will achieve one by one execution or else if multiple thread are using different resources then it will achieve concurrent execution.

4) In the case of static synchronization the default Object will be locked by the thread and there is only one default object created by JVM for one class description at the time of class loading. In this scenario, if multiple threads are using same resources or different resources then always, it will achieve one by one execution.

ex-

```
class Lab{
    public static void main (String[] args){
    Account acc = new Account();
    new AccountThread(acc);
    }
}

class Account{
int bal = 980;
public void withdraw(int amt){
  if (bal >= amt){
    System.out.println(Thread.currentThread().getName() + " is going to
                        withdraw from: " + bal);
   try{
     Thread.sleep(1200);
   }catch(Exception e){}
     bal -= amt;
     System.out.println(Thread.currentThread().getName() + " is completed
                        withdrawing. Remaining balance: " + bal);
   }else{
     System.out.println("No Fund to withdraw." + Thread.currentThread().
                        getName());
   }
}
public int getBal(){
    return bal;
}
}

class AccountThread implements Runnable{
    Account acc = null;
    AccountThread(Account acc){
      this.acc = acc;
      Thread t1 = new Thread(this, "A ");
      Thread t2 = new Thread(this, "B ");
      t1.start();
      t2.start();
    }
    public void run(){
      for(int i=0; i<=5; i++){
        acc.withdraw(225);
          if(acc.getBal() < 0){
              System.out.println("Amount is Overdrawn");
          }
      }
}
```

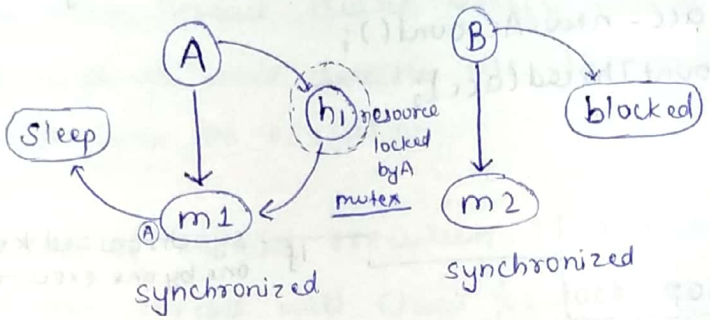
if → synchronized keyword
one by one execution

//→ concurrent execution
Not recommend for bank appln's

15/12/2017

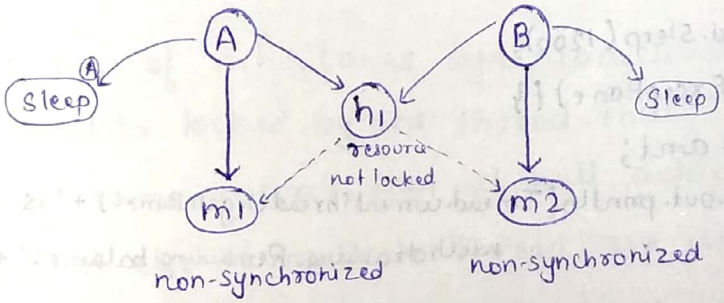
* Various cases to understand Synchronization-

=> Case 1 -



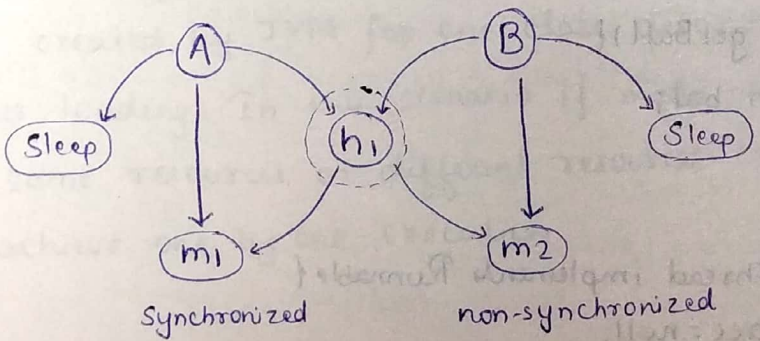
one by one execution

=> Case 2 -



concurent execution.

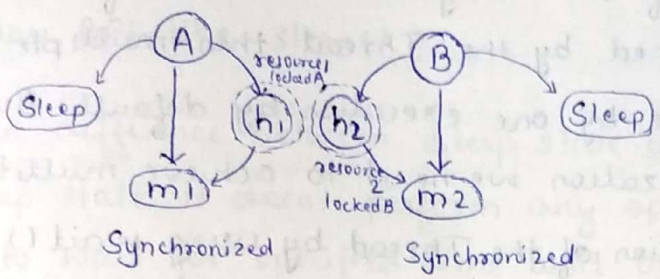
=> Case 3 -



concurent execution.

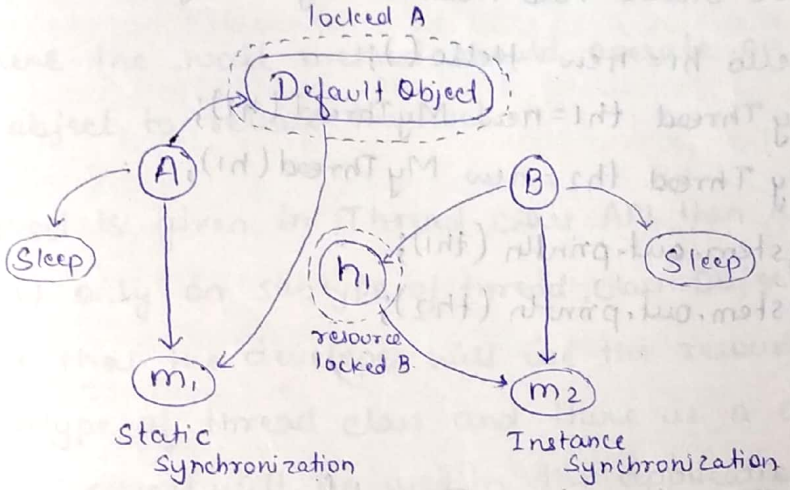
Since B thread is non-synchronized so it not not verify the lock key/locking of resource and use the resource and will perform execution.

Case 4-



Concurent execution

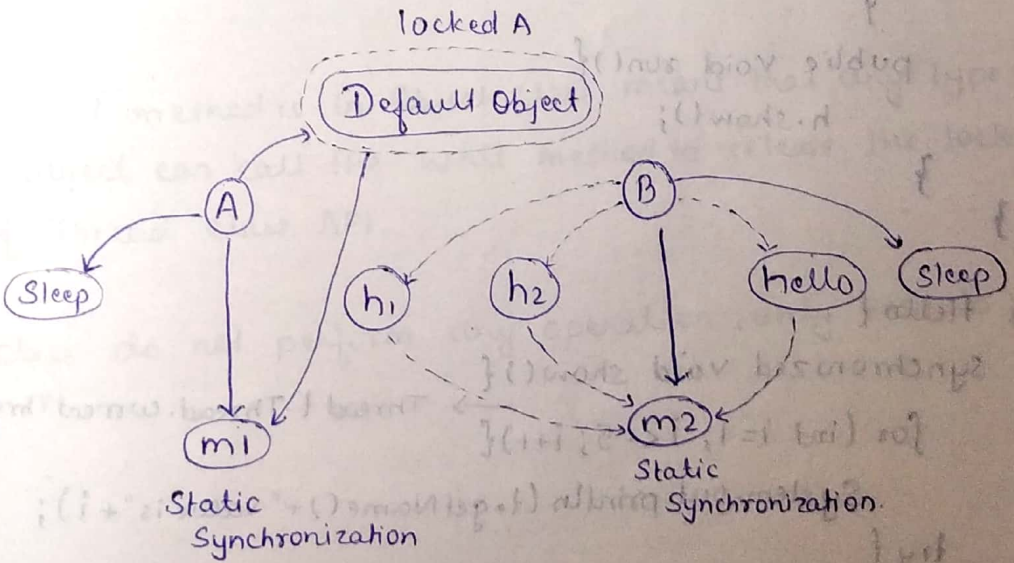
Case 5-



concurent execution.

Static synchronization locks default Object, not the current Object. So, the resource Object is available for B thread. hence both will perform concurent execution.

Case 6-



one by one execution

Static synchronization locks the default Object. So thread B has to be in que until thread A release the lock from default Object

=> In the case of static synchronization default object of the class is locked by the Thread then multiple Thread will perform one by one execution by default. But even in static synchronization we need to achieve multitasking / concurrent execution of the Thread by using wait() as follows.

ex-

```
public class Lab{
    public static void main (String[] args){
        Hello h1= new Hello();
        MyThread th1=new MyThread(h1);
        MyThread th2=new MyThread(h1);
        System.out.println(th1);
        System.out.println(th2);
    }
}

Class MyThread extends Thread{
    Hello h;
    public MyThread(Hello h){
        this.h=h;
        start();
    }
    public void run(){
        h.show();
    }
}

Class Hello{
static synchronized void show(){
if 1                                    --> Thread t=Thread.currentThread();
    for(int i=1; i<=5; i++){
        System.out.println(t.getName()+" value is "+i);
    try{
        wait(500);          //->one by one execution (i.e concurrent execution)
    // Thread.sleep(1000); //->noConcurrent execution (i.e one by one execution)
    // this.getClass().wait(500); //-> "this" cannot be used in static context
    // Hello.class.wait(500); //-> when using static then valid.
    }catch(Exception e){
        e.printStackTrace();
    }
}
}
```

Q- Why wait() is provided in Object class API but not in Thread class API like sleep() ?

Ans: The main difference between sleep state and wait state is that in sleep state, it doesn't perform any operation and make the thread to wait for specified time limit but in wait state it is performing some operation that is calling wait method on locked resource objects has to be released before moving to the wait state.

So, here the wait method should operate on any type of resources object to release the lock.

If wait method is given in Thread class API then it should call the wait() only on subtype of thread class Objects but there is no gurantee that the developer will use the resource object which are subtype of thread class and there is a chance any type of resource object will be used in the application then in that scenario wait method should be available in Object class API only since Object is a default super class for any type of class either custom or predefined class and object class members will be inherited to each and every class and can be used directly.

If the wait method is in Object class means that any type of resource Object can call the wait method to release the lock instead of Thread class API.

Sleep class do not perform any operation, only used to sleep.

Sleep State

1) It is a timed waiting state among the Thread Execution to perform concurent excertion always.

2) sleep() is provided in Thread class

3) sleep() must and should be invoked by providing the time limit as the parameter

4) In sleep state the thread cannot release the lock on the resource object

5) sleep() can be applied on any type of object either locked or unlocked object

Wait State

1) It is a timed waiting state to perform the concurrent execution of thread only in synchronized process

2) wait() is provided in Object class

3) wait() can be used in 2 ways with time limit or without time limit

4) In wait state before moving the thread to wait state the thread will release the lock on the resource object.

5) wait() must and should be invoked only on locked objects. If we call on any unlock object, it will throw exception.

18/12/2017

* wait()-

It is used in the scenario of making Thread control from wait state to the ready to run state in custom way as per the client requirement but not automatically. It has no time limit.

We can make the thread to be in wait state for a long time until we notify the thread explicitely to move from ~~rea~~ wait state to ready to run state by using following method

```
notify();
notifyAll();
```

* notify() and notifyall();

These methods are instance methods available in java.lang.Object class

=> We can call notify() on any locked object then following task will happen

1) Only one thread which has been waiting for long long time will be picked
2) Checks whether the resources released by thread are available or not.
3) If resources are available then the thread acquires the resources and goes to ready to run state
4) If resources are not available then thread goes to blocked state.

=> When we call notifyAll() on any locked object then following task will happen

1) All or some required no of threads as per the requirement which are in waiting state will be picked.
2) Checks whether the resources released are available or not
3) If resources are available then the thread acquires the resources and goes to "Ready to Run" State.
4) If resources are not available then the thread will go to blocked state.

ex-

```
public class Lab {
    public static void main (String [] args){
        Stack st = new Stack();
        B obj2= new B(st, "B");
        A obj1 = new A(st,"A");
    }
}

Class Stack{
    int x;
    boolean flag=false;
    public synchronized void push(int x) {
        if (flag){
            try{
                wait();
            } catch (Exception e) {
                System.out.println(e);
            }
        }
        this.x = x;
        System.out.println(x+" is pushed");
        flag = true;
        notify();
    }
    synchronized public int pop(){
        if(flag){
            try {
                wait();
            } catch(Exception e){
                System.out.println(e);
            }
        }
        system.out.println(x+" is poped");
        try{
            Thread.sleep(1000);
        } catch(Exception e){
            System.out.println(e);
        }
        flag= false;
        notify();
        return x;
    }
```

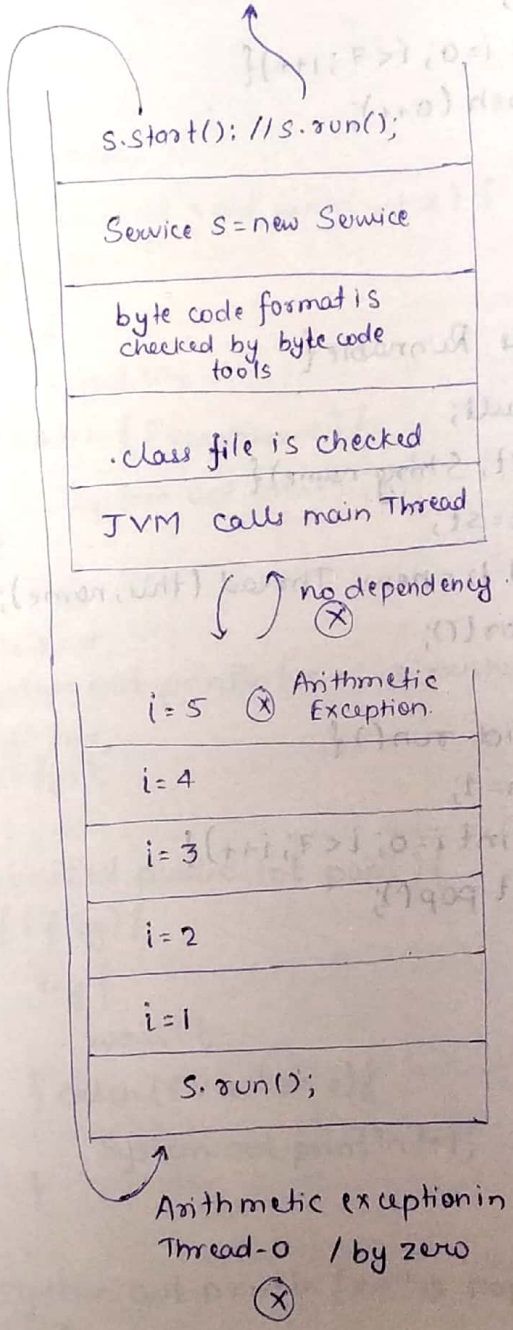
```
class A implements Runnable {
    Stack st = null;
    A(Stack st, String name) {
        this.st = st;
        Thread t1 = new Thread(this, name);
        t1.start();
    }
    public void run() {
        int a = 1;
        for(int i=0; i<7; i++) {
            st.push(a++);
        }
    }
}

class B implements Runnable {
    Stack st = null;
    B(Stack st, String name) {
        this.st = st;
        Thread t2 = new Thread(this, name);
        t2.start();
    }
    public void run() {
        int a = 1;
        for(int i=0; i<7; i++) {
            st.pop();
        }
    }
}
```

* Thread and stack Memory:-

> When we are executing multiple threds then seperate STACK will be created for each thread

> When any exception is occured in on thread execution then other thread execution will not be terminated

> ex.



ex-

```
public class Lab{
  public static void main (String[] str){
    Service s = new Service();
     s.start();
   // s.run();
    for(char ch= 'A'; ch <='J'; ch++){
       System.out.println ("main Running"+ ch);
      try{
        Thread.sleep(500);
      } catch (Exception e){
        e.printStackTrace();
       }
    }
  }
}

class Service extends Thread{
   public void run(){
     for(int i=1; i<=10; i++){
       System.out.println ("Service Running"+i);
       if(i==5){
         int x = 10/0;
       }
      try{
        Thread.sleep(500);
       }catch(Exception e){
          e.printStackTrace();
       }
     }
   }
}
```

* Difference between start() and run()-

> After creating the custom Thread object. then if we call start() on the custom Thread reference then a seperate stack memory will be created and the execution of the task will be performed by the custom Thread.

> But if we call run() on the custom Thread Object reference then there is no stack memory allocation and execution of the task will be performed by the current Thread in the current stack memory (- i.e main Thread)

#NOTE-

Every thread either custom or predefined Threads are independent Threads and each and every Thread will not depend on each other. Since every thread has its own stack memory and if any Thread has the problem, that Thread will stop the Execution and it will not create any problem for other thread execution.

* join() or yield()-

It is used to make the normal flow of execution of Threads has to be interrupted whatever we need to execute some high priority thread first then which ever the thread is interrupted that Thread will only join after the completion of high priority Thread.

=> If join() we are providing the priority preference to the Thread but in case of yield() no priority preference and any Thread will move from Ready to Run state from Runnable state

=> It is throwing checked InterruptedException.

NOTE-

While developing any custom Thread from the main method it is giving priority for executing main thread first then only custom threads will be executed that also either main thread execution completed or else it is giving the time to execute custom thread so 1st priority will go for executing main thread and later only custom threads will have priority to execute.

This is because JVM will developing main Thread and starting main Thread to perform execution of standard main method and every application starts the execution from main method so already main Thread is available in Runnable State

From the main method stack whatever we are developing custom thread and starting those will create a seperate stack memory and wait in ready to run state.

In the runnable state whenever the main Thread get a chance either by completing main() execution or moving to timed waiting state (wait(), sleep()) then only custom Threads will get time to go to Runnable state and perform execution.

This is reason 1st of all main() statement will be executed by main Thread and then only custom Thread will perform execution.

> join() will be used to join one thread at the end of another thread

> In main thread → t1.join()

main thread will join at the end of t1.

> In t1 thread → t2.join()

t1 thread will join at the end of t2.

eg:-

```
public class Lab{
    public static void main (String[] args) throws Exception{
        Greeting g=new Greeting();
        g.start();
    //  g.join();        } main thread will be executed
        System.out.println("Greetings! from JLC");
    }
}

class Greeting extends Thread{
    public void run(){
        for(int i=10; i>=1; i--){
            System.out.println(i);
            try{
                Thread.sleep(500);
            }catch(Exception e){
                e.printStackTrace();
            }
        }
    }
}
```

O/P

without g.join()

```
Greetings! from JLC
10
9
8
7
6
5
4
3
2
1
```

with g.join()

```
10
9
8
7
6
5
4
3
2
1
Greeting! from JLC.
```

19/12/2017

* Classification of Threads-

1) User Thread i.e. independent thread
2) Daemon Thread i.e. dependent thread.

1) User Thread-

By default, which ever the thread we are creating for the application either by extends Thread class or by implementing Runnable interface are known as User Thread. These threads execution will not depend on another threads execution. So every Thread will perform its task completely and then only it will be destroyed.

2) Daemon Thread-

These thread execution depends on the another user Threads Execution. If no user threads are running then JVM will destroy the daemon Thread at the point of time automatically.

By default any custom Thread will not be Daemon Thread. As per the Application Requirement if we want to make Daemon thread then explicitely we can make the thread as daemon.

Syntax-

```
setDaemon(true);   => used for making the Thread as the daemon.
isDaemon();        => used to verify whether thread is daemon or not.
```

NOTE:

If we want to make the thread as daemon then before starting the thread we need to make it as a Daemon Thread.

=> From Java 5, this new API is given for performing the same task for multiple Times by the Thread Object.

java.util.Concurrent.Executors
java.util.Concurrent.Executor
java.util.Concurrent.ExecutorService
java.util.Concurrent.ThreadPoolExecutor

#NOTE-

With normal life cycle of thread, the thread cannot perform the task for multiple times because once we start the thread then until it will perform the task and after completing the task the Thread will move to the dead state and it will not come back and there is no chance of restarting the Thread to perform the Task.

So for the repetive task of the Thread for multiple times then place the Thread Object in the Thread pool then Thread reference cannot move to the dead state and it will perform the task for multiple times with multiple Thread. Inside the Thread pool. Multiple thread will share the task and perform concurrently.