# **Multi Threading**

- When you run an application, the software that you have written will be running on an OS( Operating System).

- Below the OS, a layer is present that is known as Hardware. - Software will always run on hardware.

- Hardware consists of:

RAM - acts as a temporary memory for processing

CPU - that executes something (processing done here)

- OS supports multiple software working at the same time and it means it supports Multitasking.

- Multitasking: Multitasking is the ability of the CPU to perform multiple tasks simultaneously. There will be continuous context switching of the CPU between the tasks.

- CPU has a concept of time sharing which means each process runs for some short period of time one by one. The software runs parallelly by sharing the time in the CPU.

- We can also divide our tasks into small units.

- In the same task or a program, we can have multiple threads running at the same time.

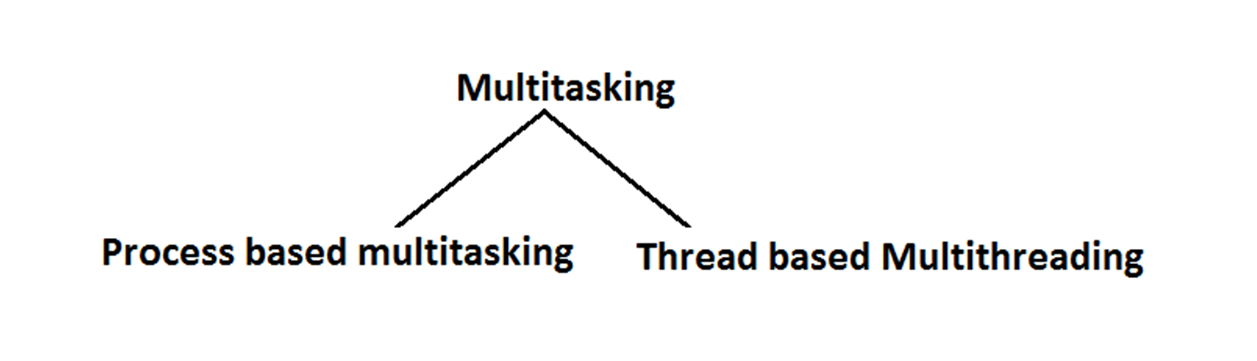
- Thread is light-weight and it is the smallest unit of a task.

- Multithreading:- Multithreading is a system in which many threads are created from a process through which the computer power is increased.

**Multitasking:** Executing several tasks simultaneously is the concept of multitasking. There are two types of multitasking's.

1. **Process based multitasking.**

2. **Thread based multitasking.**

**Diagram:  
  
**

***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:

* While typing a java program in the editor we can able to listen mp3 audio songs at the same time we can download a file from the net all these tasks are independent of each other and executing simultaneously and hence it is Process based multitasking.
* This type of multitasking is best suitable 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.  
 And each independent part is called a "Thread".

1. This type of multitasking is best suitable for "programatic level".

2. When compared with "C++", developing multithreading examples is very easy in java because java provides in built support for multithreading through a rich API (Thread, Runnable, ThreadGroup, ThreadLocal...etc).

3. In multithreading on 10% of the work the programmer is required to do and 90% of the work will be down by java API.

4. ***The main important application areas of multithreading are:***

1. To implement multimedia graphics.

2. To develop animations.

3. To develop video games etc.

4. To develop web and application servers

5. Whether it is process based or Thread based the main objective of multitasking is to improve performance of the system by reducing response time.

## **The ways to define instantiate and start a new Thread:**

We can define a Thread in the following 2 ways.

1. By extending Thread class.

2. By implementing Runnable interface.

1. **By extending Thread class**.

You can create a new class that extends the Thread class and override its run() method to define the task that the thread will execute.

// Define a package for the classes

**package** com.codegnan.multithreading;

// Define a class that extends Thread

**class** MyThread **extends** Thread {

// Override the run() method to define thread's task

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

// Task for child thread

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread");

}

}

}

// Main class that contains the main method

**public** **class** ThreadDemo {

// Main method where the program starts execution

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

// Create an instance of MyThread

MyThread t = **new** MyThread(); // Instantiation of a Thread

// Start the thread using start() method

t.start(); // starting of a Thread

// Task for main thread

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("main thread");

}

}

}

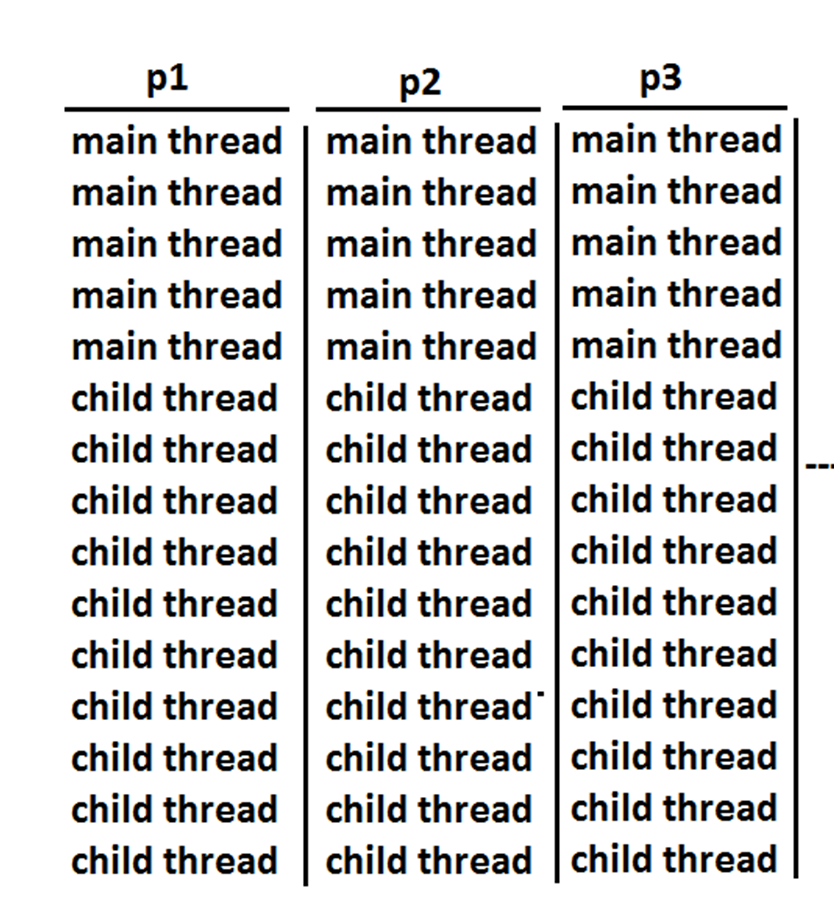
***Case 1: Thread Scheduler:***

* If multiple Threads are waiting to execute then which Thread will execute 1st is decided by "Thread Scheduler" which is part of JVM.
* Which algorithm or behavior followed by Thread Scheduler we can't expect exactly it is the JVM vendor dependent hence in multithreading examples we can't expect exact execution order and exact output.
* The following are various possible outputs for the above program.

***Case 2: Difference between t.start() and t.run() methods.***

* 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 r un() method will be executed just like a normal method by the main Thread.
* In the above program if we are replacing t.start() with t.run() the following is the output.

Entire output produced by only main Thread.



### ***Case 9:***

After starting a Thread we are not allowed to restart the same Thread once again otherwise we will get runtime exception saying "IllegalThreadStateException".

Example:

MyThread t=new MyThread();

t.start();//valid

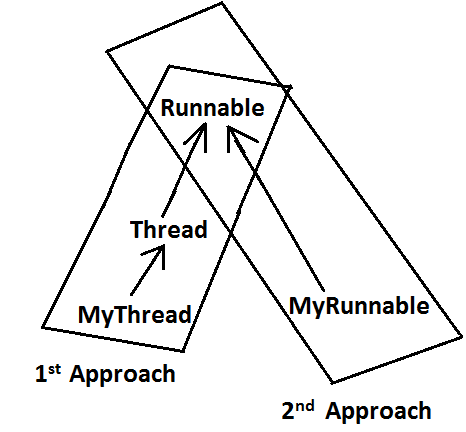
;;;;;;;;

t.start();//we will get R.E saying: IllegalThreadStateException

## **Defining a Thread by implementing Runnable interface:**

· **Implementing Runnable interface**: In Example 2, MyRunnable class implements Runnable interface and provides an implementation for the run() method. This class is then used to create a Thread object (thread) which is started similarly with thread.start().

**Important Note**: It's crucial to start a thread using thread.start() method. Directly calling thread.run() will not create a new thread and will execute the run() method in the current thread sequentially.

We can define a Thread even by implementing Runnable interface also.  
 Runnable interface present in java.lang.pkg and contains only one method run().  
 *Diagram:  
* *Example:*

// Define a package for the classes

**package** com.codegnan.multithreading;

// Define a class that implements Runnable interface

**class** MyRunnable **implements** Runnable {

// Override the run() method to define thread's task

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

// Task for child thread

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread");

}

}

}

// Main class that contains the main method

**class** ThreadDemo {

// Main method where the program starts execution

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

// Create an instance of MyRunnable

MyRunnable r = **new** MyRunnable();

// Create a Thread object with MyRunnable instance

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

// Start the thread using start() method

t.start();

// Task for main thread

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("main thread");

}

}

}

***Best approach to define a Thread:***

* Among the 2 ways of defining a Thread, implements Runnable approach is always recommended.
* In the 1st approach our class should always extends Thread class there is no chance of extending any other class hence we are missing the benefits of inheritance.
* But in the 2nd approach while implementing Runnable interface we can extend some other class also. Hence implements Runnable mechanism is recommended to define a Thread.

## **Getting and setting name of a Thread:**

* Every Thread in java has some name it may be provided explicitly by the programmer or automatically generated by JVM.
* Thread class defines the following methods to get and set name of a Thread.

*Methods:*

1. public final String getName()

2. public final void setName(String name)

example

-------------

// Define a package for the classes

**package** com.codegnan.multithreading;

//Define a subclass of Thread

**class** MyThread **extends** Thread {

// No additional methods or overrides provided

}

//Main class to demonstrate thread behavior

**class** ThreadDemo {

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

// Output the name of the current thread (which is main by default)

System.***out***.println(Thread.*currentThread*().getName()); // Output: main

// Create an instance of MyThread

MyThread t = **new** MyThread();

// Output the name of the newly created thread (default name assigned by JVM)

System.***out***.println(t.getName()); // Output: Thread-0

// Change the name of the current thread to "codegnan Thread"

Thread.*currentThread*().setName("codegnan Thread");

// Output the updated name of the current thread

System.***out***.println(Thread.*currentThread*().getName()); // Output: Bhaskar Thread

}

}

## **Thread Priorities**

* Every Thread in java has some priority it may be default priority generated by JVM (or) explicitly provided by the programmer.
* The valid range of Thread priorities is 1 to 10[but not 0 to 10] where 1 is the least priority and 10 is highest priority.
* Thread class defines the following constants to represent some standard priorities.

1. Thread. MIN\_PRIORITY----------1

2. Thread. MAX\_PRIORITY----------10

3. Thread. NORM\_PRIORITY--------5

* There are no constants like Thread.LOW\_PRIORITY, Thread.HIGH\_PRIORITY
* Thread scheduler uses these priorities while allocating CPU.
* The Thread which is having highest priority will get chance for 1st execution.
* If 2 Threads having the same priority then we can't expect exact execution order it depends on Thread scheduler whose behavior is vendor dependent.
* We can get and set the priority of a Thread by using the following methods.

1. public final int getPriority()

2. public final void setPriority(int newPriority);//the allowed values are 1 to 10

* The allowed values are 1 to 10 otherwise we will get runtime exception saying "IllegalArgumentException".

## ***Default priority:***

The default priority only for the main Thread is 5. But for all the remaining Threads the default priority will be inheriting from parent to child. That is whatever the priority parent has by default the same priority will be for the child also.

// Define a package for the classes

**package** com.codegnan.multithreading;

//Define a class that extends Thread

**class** PriorityThread **extends** Thread {

// Constructor to initialize thread name

**public** PriorityThread(String name) {

**super**(name);

}

// Override run method to define thread behavior

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

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

System.***out***.println(getName() + " is running with priority " + getPriority() + " - Count: " + i);

**try** {

// Simulate some workload

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

//Main class to demonstrate thread priorities

**public** **class** ThreadPriorityDemo {

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

// Create three threads with different priorities

PriorityThread t1 = **new** PriorityThread("Thread 1");

PriorityThread t2 = **new** PriorityThread("Thread 2");

PriorityThread t3 = **new** PriorityThread("Thread 3");

// Set priorities for each thread

t1.setPriority(Thread.***MIN\_PRIORITY***); // MIN\_PRIORITY = 1

t2.setPriority(Thread.***NORM\_PRIORITY***); // NORM\_PRIORITY = 5 (default)

t3.setPriority(Thread.***MAX\_PRIORITY***); // MAX\_PRIORITY = 10

// Start the threads

t1.start();

t2.start();

t3.start();

}

}

## **The Methods to Prevent a Thread from Execution:**

We can prevent(stop) a Thread execution by using the following methods.

1. yield();

2. join();

3. sleep();

### ***yield():***

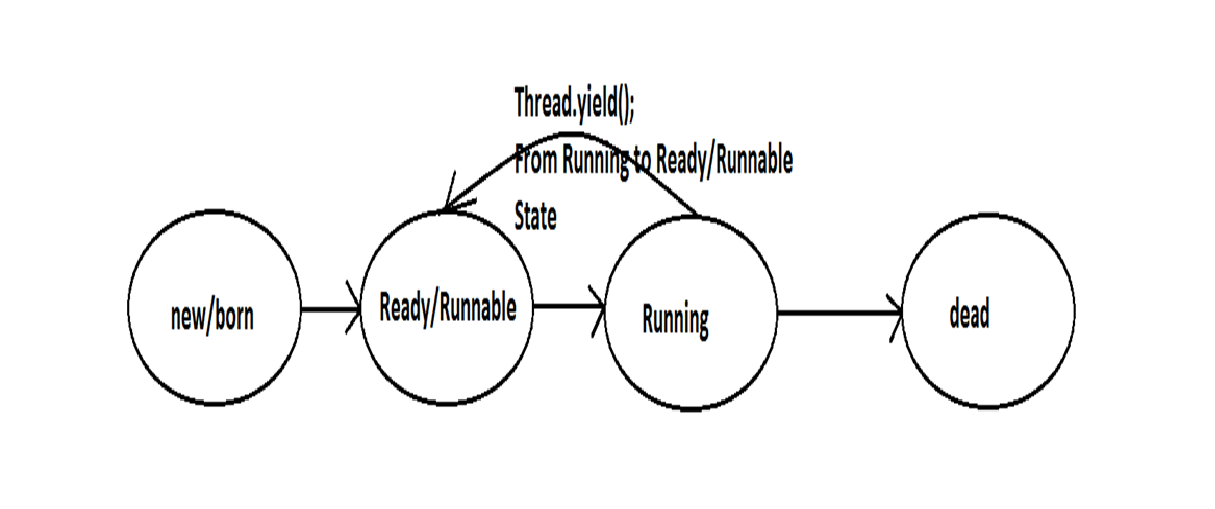
1. yield() method causes "to pause current executing Thread for giving the chance of remaining waiting Threads of same priority".

2. If all waiting Threads have the low priority or if there is no waiting Threads then the same Thread will be continued its execution.

3. If several waiting Threads with same priority available then we can't expect exact which Thread will get chance for execution.

4. The Thread which is yielded when it get chance once again for execution is depends on mercy of the Thread scheduler.

5. public static native void yield();

*Diagram:  
  
*

// Define a package for the classes

**package** com.codegnan.multithreading;

//Define a subclass of Thread

**class** MyThread **extends** Thread {

// Override run method to define thread behavior

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

**for** (**int** i = 0; i < 5; i++) {

// Yield control to other threads

Thread.*yield*();

System.***out***.println("child thread");

}

}

}

//Main class to demonstrate thread yield

**public** **class** ThreadYieldDemo {

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

// Create an instance of MyThread

MyThread t = **new** MyThread();

// Start the child thread

t.start();

// Main thread's execution

**for** (**int** i = 0; i < 5; i++) {

System.***out***.println("main thread");

}

}

}

Output:

main thread

main thread

main thread

main thread

main thread

child thread

child thread

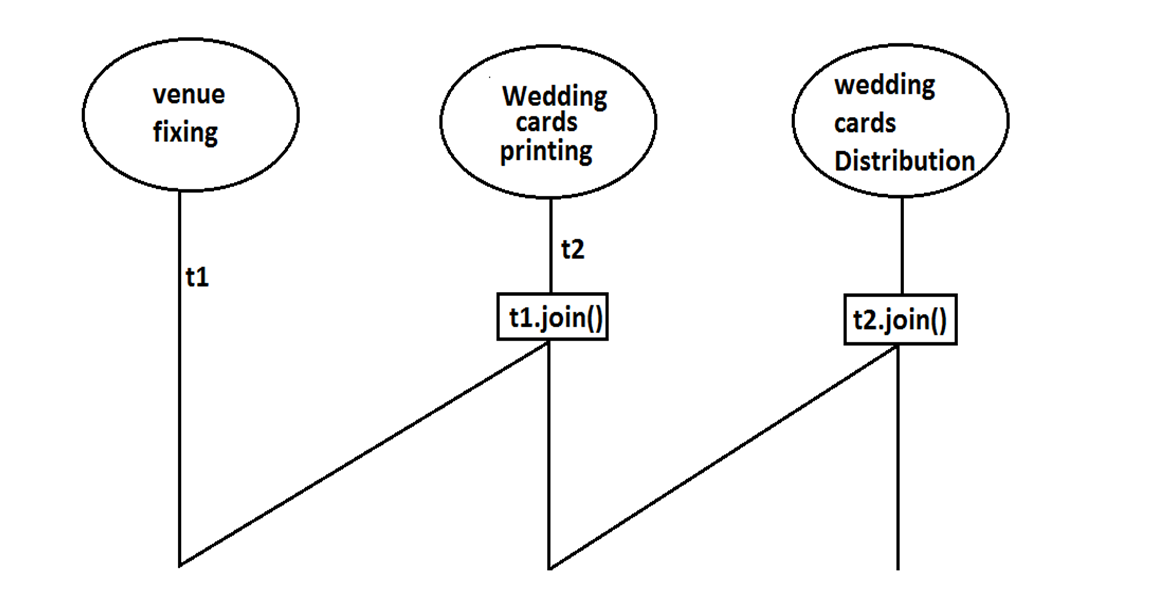
child thread

child thread

child thread

In the above program child Thread always calling yield() method and hence main Thread will get the chance more number of times for execution.  
 Hence the chance of completing the main Thread first is high.  
  
 Note : Some operating systems may not provide proper support for yield() method.

***Join():***

If a Thread wants to wait until completing some other Thread then we should go for join() method.  
 **Example:** If a Thread t1 executes t2.join() then t1 should go for waiting state until completing t2.  
  
 Diagram:  


1. public final void join()throws InterruptedException

2. public final void join(long ms) throws InterruptedException

3. public final void join(long ms,int ns) throws InterruptedException

*Diagram:* Every join() method throws InterruptedException, which is checked exception hence compulsory we should handle either by **try catch** or by **throws** keyword.  
 Otherwise we will get compiletime error.

// Define a package for the classes

**package** com.codegnan.multithreading;

//Define a subclass of Thread

**class** MyThread **extends** Thread {

// Override run method to define thread behavior

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

**for** (**int** i = 0; i < 5; i++) {

System.***out***.println("Sita Thread");

**try** {

// Introduce a delay of 2000 milliseconds (2 seconds)

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

//Main class to demonstrate thread join

**public** **class** ThreadJoinDemo {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

// Create an instance of MyThread

MyThread t = **new** MyThread();

// Start the child thread

t.start();

// Uncomment the line below to see the effect of t.join()

// t.join(); // --->1

// Main thread's execution

**for** (**int** i = 0; i < 5; i++) {

System.***out***.println("Rama Thread");

}

}

}

* If we are commenting line 1 then both Threads will be executed simultaneously and we can't expect exact execution order.
* If we are not commenting line 1 then main Thread will wait until completing child Thread in this the output is sita Thread 5 times followed by Rama Thread 5 times.

## ***Sleep() method:***

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

1. **public static native void sleep(long ms) throws InterruptedException**

2. **public static void sleep(long ms,int ns)throws InterruptedException**

*Diagram:*

Example

------------

// Define a package for the classes

**package** com.codegnan.multithreading;

**public** **class** DailyRoutineSimulation {

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

System.***out***.println("Good morning!");

// Simulate morning routine

*wakeUp*();

*getReady*();

*haveBreakfast*();

// Simulate work or school day

*goWorkOrSchool*();

// Simulate evening routine

*haveDinner*();

*relax*();

*sleep*();

}

**public** **static** **void** wakeUp() {

System.***out***.println("Wake up early in the morning");

**try** {

Thread.*sleep*(2000); // Simulate waking up and getting out of bed

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Get out of bed");

}

**public** **static** **void** getReady() {

System.***out***.println("Get ready for the day");

**try** {

Thread.*sleep*(3000); // Simulate getting dressed, brushing teeth, etc.

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Ready to start the day");

}

**public** **static** **void** haveBreakfast() {

System.***out***.println("Have breakfast");

**try** {

Thread.*sleep*(4000); // Simulate having breakfast

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Breakfast done");

}

**public** **static** **void** goWorkOrSchool() {

System.***out***.println("Go to work or school");

**try** {

Thread.*sleep*(5000); // Simulate travel time and arrival

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Arrive at work or school");

}

**public** **static** **void** haveDinner() {

System.***out***.println("Have dinner");

**try** {

Thread.*sleep*(4000); // Simulate having dinner

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Dinner finished");

}

**public** **static** **void** relax() {

System.***out***.println("Relax and unwind");

**try** {

Thread.*sleep*(3000); // Simulate relaxing time

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Feeling relaxed");

}

**public** **static** **void** sleep() {

System.***out***.println("Go to bed and sleep");

**try** {

Thread.*sleep*(5000); // Simulate going to bed and falling asleep

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Good night!");

}

}

Output:

---------

Good morning!

Wake up early in the morning

Get out of bed

Get ready for the day

Ready to start the day

Have breakfast

Breakfast done

Go to work or school

Arrive at work or school

Have dinner

Dinner finished

Relax and unwind

Feeling relaxed

Go to bed and sleep

Good night!

### **Compression of yield, join and sleep() method?**

| **property** | **Yield()** | **Join()** | **Sleep()** |
| --- | --- | --- | --- |
| 1) Purpose? | To pause current executing Thread for giving the chance of remaining waiting Threads of same priority. | If a Thread wants to wait until completing some other Thread then we should go for join. | If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method. |
| 2) Is it static? | yes | no | yes |
| 3) Is it final? | no | yes | no |
| 4) Is it overloaded? | No | yes | yes |
| 5) Is it throws InterruptedException? | no | yes | yes |
| 6) Is it native method? | yes | no | sleep(long ms) -->native  sleep(long ms,int ns) -->non-native |

## **Synchronization**

1. Synchronized is the keyword applicable for methods and blocks but not for classes and variables.

2. If a method or block declared as the synchronized then at a time only one Thread is allow to execute that method or block on the given object.

3. The main advantage of synchronized keyword 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 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 got the lock of that object then it's allow 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 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].

// Define a package for the classes

**package** com.codegnan.multithreading;

**class** Display {

// synchronized method to ensure thread-safe execution

**public** **synchronized** **void** wish(String name) {

// loop to print "good morning: <name>" 5 times

**for** (**int** i = 0; i < 5; i++) {

System.***out***.print("good morning:");

**try** {

// introducing a delay to simulate real-world scenario

Thread.*sleep*(1000); // thread sleeps for 1000 milliseconds (1 second)

} **catch** (InterruptedException e) {

// handling interrupted exception (not doing anything here)

}

System.***out***.println(name); // prints the name passed to the method

}

}

}

**class** MyThread **extends** Thread {

Display d;

String name;

// constructor to initialize the Display object and name

MyThread(Display d, String name) {

**this**.d = d;

**this**.name = name;

}

// run method where the thread executes

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

d.wish(name); // calls the wish method of Display object d with name as parameter

}

}

**public** **class** SynchronizedDemo {

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

Display d1 = **new** Display(); // creating a single Display object

MyThread t1 = **new** MyThread(d1, "dhoni"); // creating thread t1 with name "dhoni"

MyThread t2 = **new** MyThread(d1, "yuvaraj"); // creating thread t2 with name "yuvaraj"

t1.start(); // starting thread t1

t2.start(); // starting thread t2

}

}

## **Dead lock:**

* 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 dead lock but several prevention(avoidance) techniques are possible.
* Synchronized keyword is the cause for deadlock hence whenever we are using synchronized keyword we have to take special care.

**Daemon Threads:**

The Threads which are executing in the background are called daemon Threads.  
 The main objective of daemon Threads is to provide support for non-daemon Threads like main Thread.

**Example:** *Garbage collector*

When ever the program runs with low memory the JVM will execute Garbage Collector to provide free memory. So that the main Thread can continue it's execution.

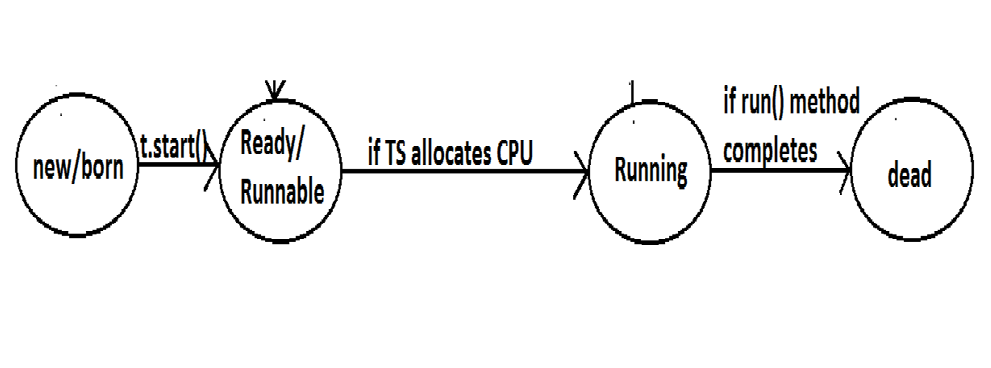
* We can check whether the Thread is daemon or not by using isDaemon() method of Thread class.  
   **public final boolean isDaemon();**
* We can change daemon nature of a Thread by using setDaemon () method.  
   **public final void setDaemon(boolean b);**
* But we can change daemon nature before starting Thread only. That is after starting the Thread if we are trying to change the daemon nature we will get R.E saying ***IllegalThreadStateException***.
* **Default Nature :** Main Thread is always non daemon and we can't change its daemon nature because it's already started at the beginning only.
* Main Thread is always non daemon and for the remaining Threads daemon nature will be inheriting from parent to child that is if the parent is daemon child is also daemon and if the parent is non daemon then child is also non daemon.
* Whenever the last non daemon Thread terminates automatically all daemon Threads will be terminated.

### **RACE condition:**

Executing multiple Threads simultaneously and causing data inconsistency problems is nothing but **Race condition** we can resolve race condition by using synchronized keyword.

Thread States

—--------------------

****

* Once we created a Thread object then the Thread is said to be in 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 run() method completes then the Thread will entered into dead state.