CORE JAVA

With

SCJP / OCJP

Study Material

Chapter 7 : Multi Threading

DURGA M.Tech
(Sun certified & Realtime Expert)
Ex. IBM Employee
Trained Lakhs of Students
for last 14 years across INDIA

Multi Threading

Agenda

1. Introduction.
2. The ways to define, instantiate and start a new Thread.
    1. By extending Thread class
    2. By implementing Runnable interface
3. Thread class constructors
4. Thread priority
5. Getting and setting name of a Thread.
6. The methods to prevent(stop) Thread execution.
    1. yield()
    2. join()
    3. sleep()
7. Synchronization.
8. Inter Thread communication.
9. Deadlock
10. Daemon Threads.
11. Various Conclusion
    1. To stop a Thread
    2. Suspend & resume of a thread
    3. Thread group
    4. Green Thread
    5. Thread Local
12. Life cycle of a Thread

Introduction

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.**

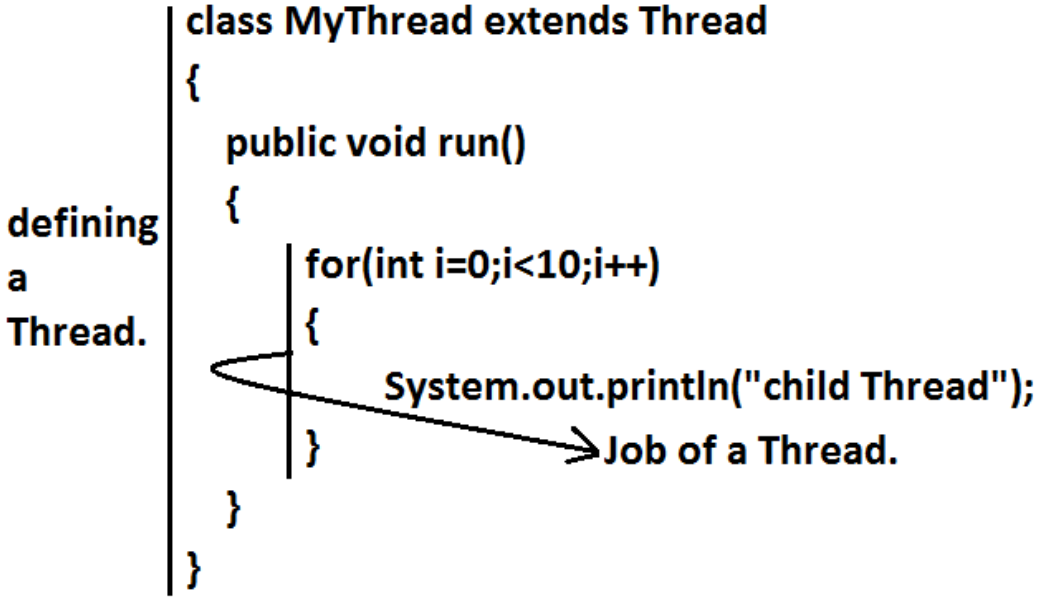
The ways to define instantiate and start a new Thread:

What is singleton? Give example?
We can define a Thread in the following 2 ways.

1. **By extending Thread class.**
2. **By implementing Runnable interface.**

Defining a Thread by extending "Thread class":

Example:




```
class ThreadDemo
{
	public static void main(String[] args)
	{
		MyThread t=new MyThread();//Instantiation of a Thread
		t.start();//starting of a Thread

		for(int i=0;i<5;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.**

p1	p2	p3	
main thread	main thread	main thread	
main thread	main thread	main thread	
main thread	main thread	main thread	
main thread	main thread	main thread	
main thread	main thread	main thread	
child thread	child thread	child thread	
child thread	child thread	child thread	----------
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	

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 run() 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.**

```
Output:
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
```

```
main thread
```
Entire output produced by only main Thread.

Case 3: importance of Thread class start() method.

For every Thread the required mandatory activities like registering the Thread with Thread Scheduler will takes care by Thread class start() method and programmer is responsible just to define the job of the Thread inside run() method.
That is start() method acts as best assistant to the programmer.

```
Example:
start()
{
        1. Register Thread with Thread Scheduler
        2. All other mandatory low level activities.
        3. Invoke or calling run() method.
}
```
We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java. Due to this start() is considered as heart of multithreading.

Case 4: If we are not overriding run() method:

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

```
Example:
class MyThread extends Thread
{}
class ThreadDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
        }
}
```
It is highly recommended to override run() method. Otherwise don't go for multithreading concept.

Case 5: Overloding of run() method.

We can overload run() method but Thread class start() method always invokes no argument run() method the other overload run() methods we have to call explicitly then only it will be executed just like normal method.

```
Example:
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("no arg method");
        }
        public void run(int i)
        {
                System.out.println("int arg method");
        }
}
```

```
class ThreadDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
        }
}
Output:
No arg method
```

Case 6: overriding of start() method:

If we override start() method then our start() method will be executed just like a normal method call and no new Thread will be started.

```
Example:
class MyThread extends Thread
{
        public void start()
        {
                System.out.println("start method");
        }
        public void run()
        {
                System.out.println("run method");
        }
}
class ThreadDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
                System.out.println("main method");
        }
}
Output:
start method
main method
```

Entire output produced by only main Thread.

Note : It is never recommended to override start() method.

Case 7:

Example 1:

```
class MyThread extends Thread
{
    public void start()
    {
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
```

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        System.out.println("main method");
    }
}
```

output:

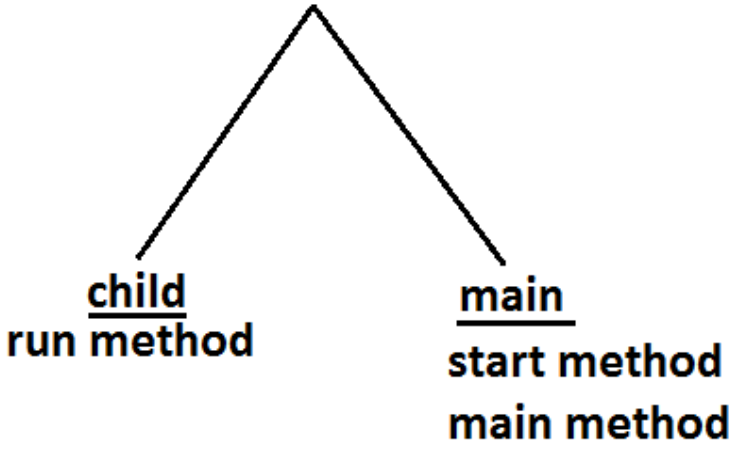
```
main thread
start method
main method
```

Example 2:

```
class MyThread extends Thread
{
    public void start()
    {
        super.start();
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
```

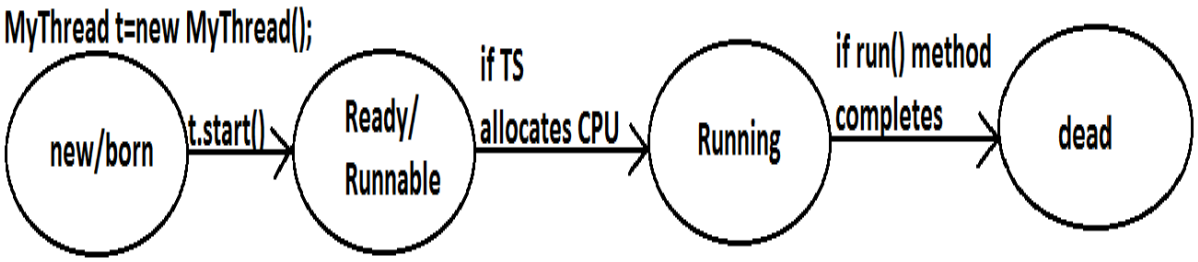
```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        System.out.println("main method");
    }
}
```

Output:



Case 8: life cycle of the Thread:

Diagram:



- **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.**

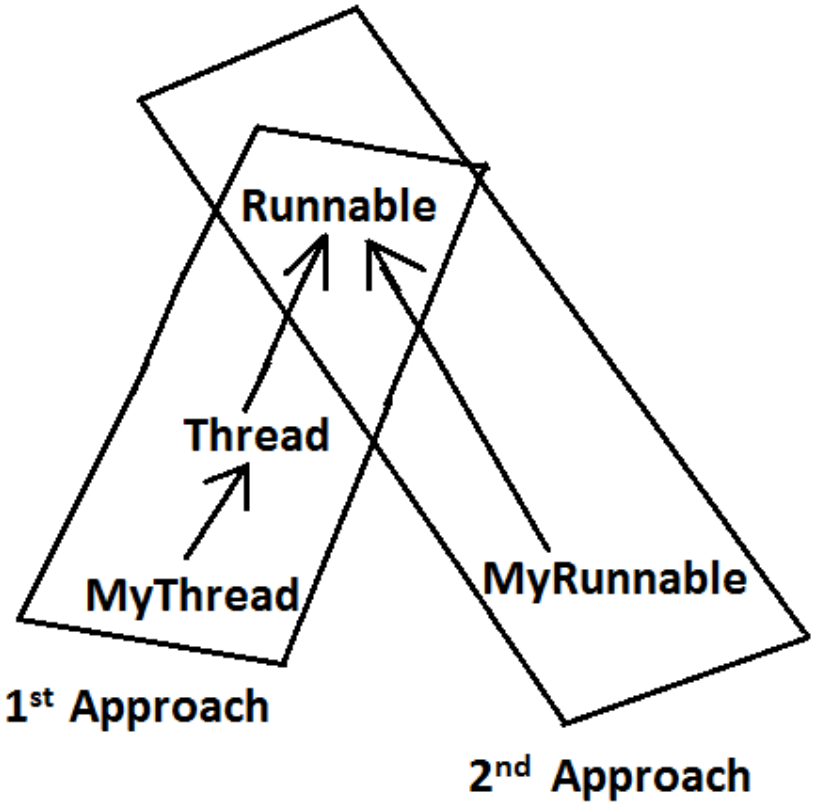
Case 9:

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

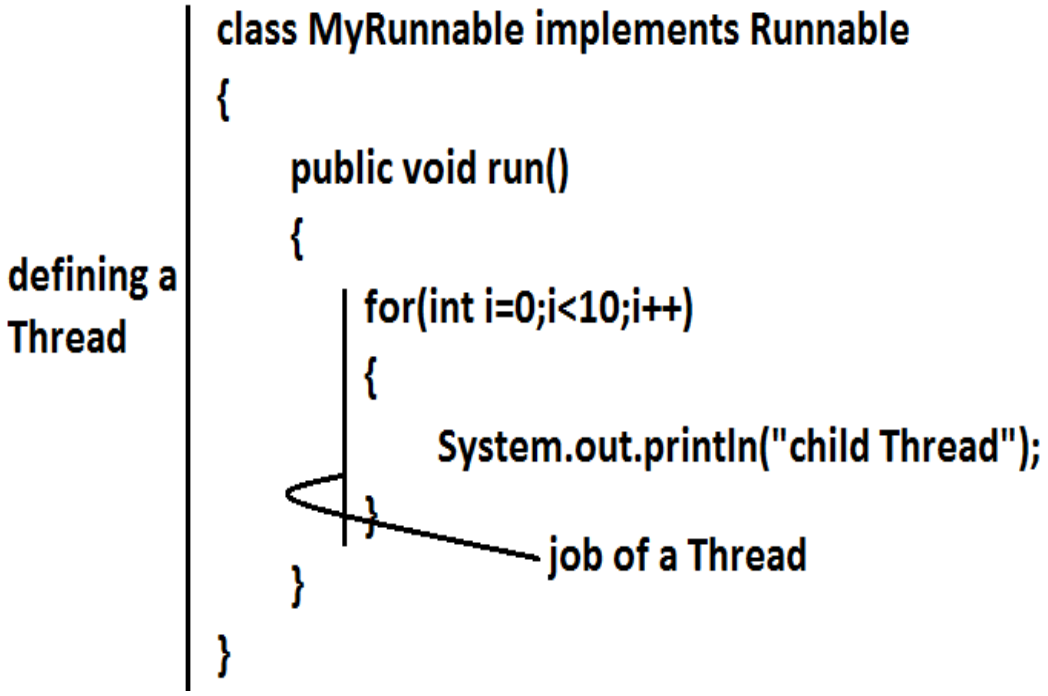
```
Example:
MyThread t=new MyThread();
t.start();//valid
;;;;;;;;
t.start();//we will get R.E saying: IllegalThreadStateException
```

Defining a Thread by implementing Runnable interface:

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:




```
class ThreadDemo
{
	public static void main(String[] args)
	{
		MyRunnable r=new MyRunnable();
		Thread t=new Thread(r);//here r is a Target Runnable
		t.start();

		for(int i=0;i<10;i++)
		{
			System.out.println("main thread");
		}
	}
}
Output:
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
child Thread
```

```
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
```

We can't expect exact output but there are several possible outputs.

Case study:

```
MyRunnable r=new MyRunnable();
Thread t1=new Thread();
Thread t2=new Thread(r);
```

Case 1: t1.start():

A new Thread will be created which is responsible for the execution of Thread class run()method.

```
Output:
main thread
main thread
main thread
main thread
main thread
```

Case 2: t1.run():

No new Thread will be created but Thread class run() method will be executed just like a normal method call.

```
Output:
main thread
main thread
main thread
main thread
main thread
```

Case 3: t2.start():

New Thread will be created which is responsible for the execution of MyRunnable run() method.

```
Output:
main thread
main thread
main thread
main thread
main thread
child Thread
child Thread
child Thread
child Thread
child Thread
```

Case 4: t2.run():

No new Thread will be created and MyRunnable run() method will be executed just like a normal method call.

```
Output:
child Thread
child Thread
child Thread
child Thread
child Thread
main thread
main thread
main thread
main thread
main thread
```

Case 5: r.start():

We will get compile time error saying start()method is not available in MyRunnable class.

```
Output:
Compile time error
E:\SCJP>javac ThreadDemo.java
ThreadDemo.java:18: cannot find symbol
Symbol: method start()
Location: class MyRunnable
```

Case 6: r.run():

No new Thread will be created and MyRunnable class run() method will be executed just like a normal method call.

```
Output:
child Thread
child Thread
child Thread
child Thread
child Thread
main thread
main thread
main thread
main thread
main thread
```

In which of the above cases a new Thread will be created which is responsible for the execution of MyRunnable run() method ?
t2.start();

In which of the above cases a new Thread will be created ?
t1.start();
t2.start();

In which of the above cases MyRunnable class run() will be executed ?
t2.start();

```
t2.run();
r.run();
```

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.**

Thread class constructors:

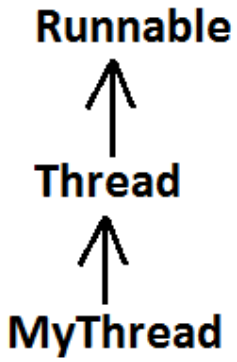
1. **Thread t=new Thread();**
2. **Thread t=new Thread(Runnable r);**
3. **Thread t=new Thread(String name);**
4. **Thread t=new Thread(Runnable r,String name);**
5. **Thread t=new Thread(ThreadGroup g,String name);**
6. **Thread t=new Thread(ThreadGroup g,Runnable r);**
7. **Thread t=new Thread(ThreadGroup g,Runnable r,String name);**
8. **Thread t=new Thread(ThreadGroup g,Runnable r,String name,long stackSize);**

Ashok's approach to define a Thread(not recommended to use):

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("run method");
    }
}
```

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        Thread t1=new Thread(t);
        t1.start();
        System.out.println("main method");
    }
}
```

Diagram:




```
Output:
main method
run method
```

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:
class MyThread extends Thread
{}
class ThreadDemo
{
        public static void main(String[] args)
        {
                System.out.println(Thread.currentThread().getName());//main
                MyThread t=new MyThread();
                System.out.println(t.getName());//Thread-0
                Thread.currentThread().setName("Bhaskar Thread");

        System.out.println(Thread.currentThread().getName());//Bhaskar
Thread
        }
}
```

Note: We can get current executing Thread object reference by using Thread.currentThread() method.

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.

```
Example 1:
class MyThread extends Thread
{}
class ThreadPriorityDemo
{
        public static void main(String[] args)
        {

        System.out.println(Thread.currentThread().getPriority());//5
                Thread.currentThread().setPriority(9);
                MyThread t=new MyThread();
                System.out.println(t.getPriority());//9
        }
}
Example 2:
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<10;i++)
                {
                        System.out.println("child thread");
                }
        }
}
class ThreadPriorityDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
```

```
                //t.setPriority(10);       //----> 1
                t.start();
                for(int i=0;i<10;i++)
                {
                        System.out.println("main thread");
                }
        }
}
```

- **If we are commenting line 1 then both main and child Threads will have the same priority and hence we can't expect exact execution order.**
- **If we are not commenting line 1 then child Thread has the priority 10 and main Thread has the priority 5 hence child Thread will get chance for execution and after completing child Thread main Thread will get the chance in this the output is:**

```
Output:
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
```

Some operating systems(like windowsXP) may not provide proper support for Thread priorities. We have to install separate bats provided by vendor to provide support for priorities.

The Methods to Prevent a Thread from Execution:

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

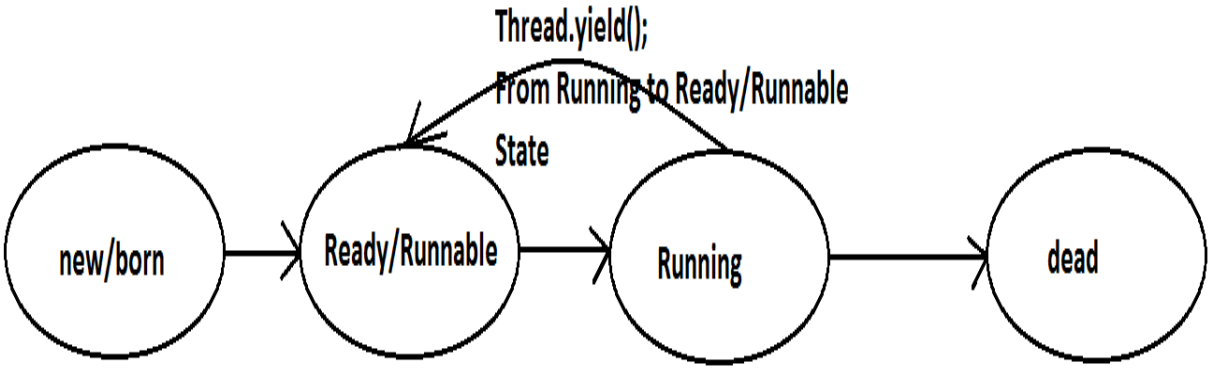
1. **yield();**
2. **join();**
3. **sleep();**

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:




```
Example:
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<5;i++)
                {
                        Thread.yield();
                        System.out.println("child thread");
                }
        }
}
class ThreadYieldDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
                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
```

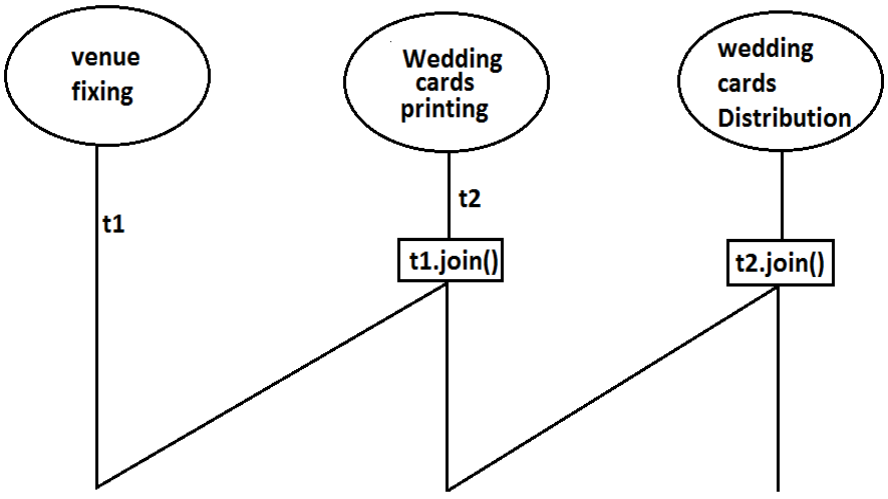
In the above program child Thread always calling yield() method and hence main Thread will get the chance more number of times for execution.
Hence the chance of completing the main Thread first is high.

Note : Some operating systems may not provide proper support for yield() method.

Join():

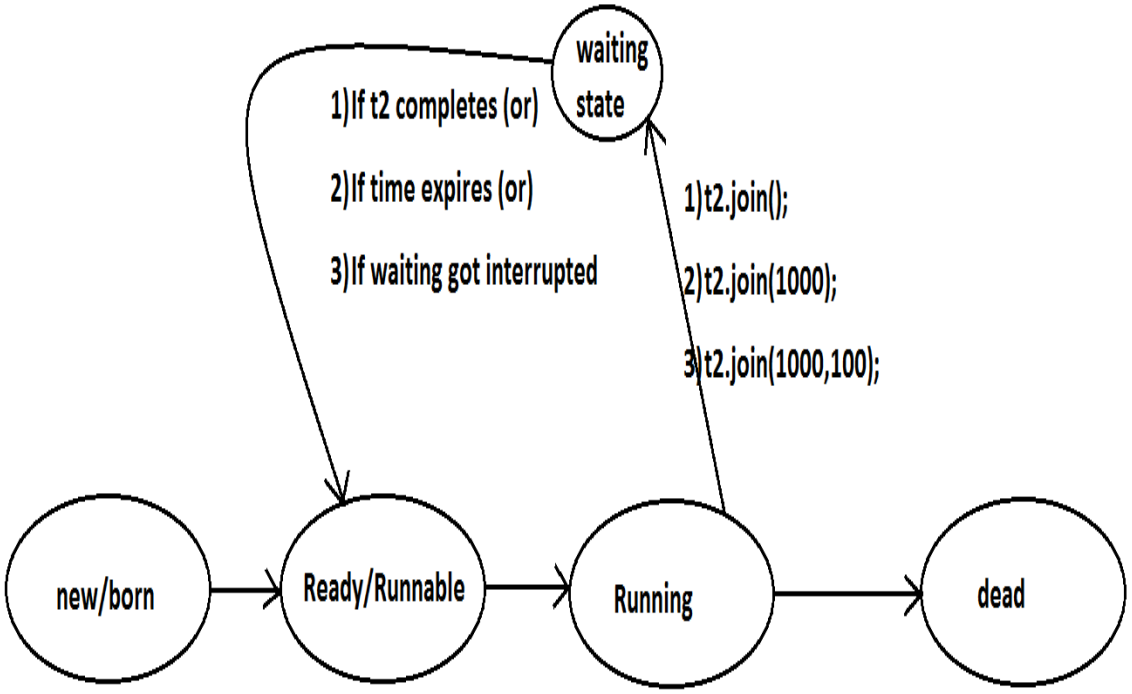
If a Thread wants to wait until completing some other Thread then we should go for join() method.
Example: If a Thread t1 executes t2.join() then t1 should go for waiting state until completing t2.

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.

```
Example:
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<5;i++)
                {
                        System.out.println("Sita Thread");
                        try
                        {
                                Thread.sleep(2000);
                        }
                        catch (InterruptedException e){}
                }
        }
}
class ThreadJoinDemo
{
        public static void main(String[] args)throws InterruptedException
        {
```

```
                MyThread t=new MyThread();
                t.start();
                //t.join();    //--->1
                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.**

Waiting of child Thread untill completing main Thread :

```
Example:
class MyThread extends Thread
{
 static Thread mt;
        public void run()
        {
                try
                {
                        mt.join();
                }
                catch (InterruptedException e){}


                for(int i=0;i<5;i++)
                {
                        System.out.println("Child Thread");
                }
        }
}
class ThreadJoinDemo
{
        public static void main(String[] args)throws InterruptedException
        {
                MyThread mt=Thread.currentThread();
                MyThread t=new MyThread();
                t.start();

                for(int i=0;i<5;i++)
                {
                        Thread.sleep(2000);
                        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
```

Note :
If main thread calls join() on child thread object and child thread called join() on main thread object then both threads will wait for each other forever and the program will be hanged(like deadlock if a Thread class join() method on the same thread itself then the program will be hanged).

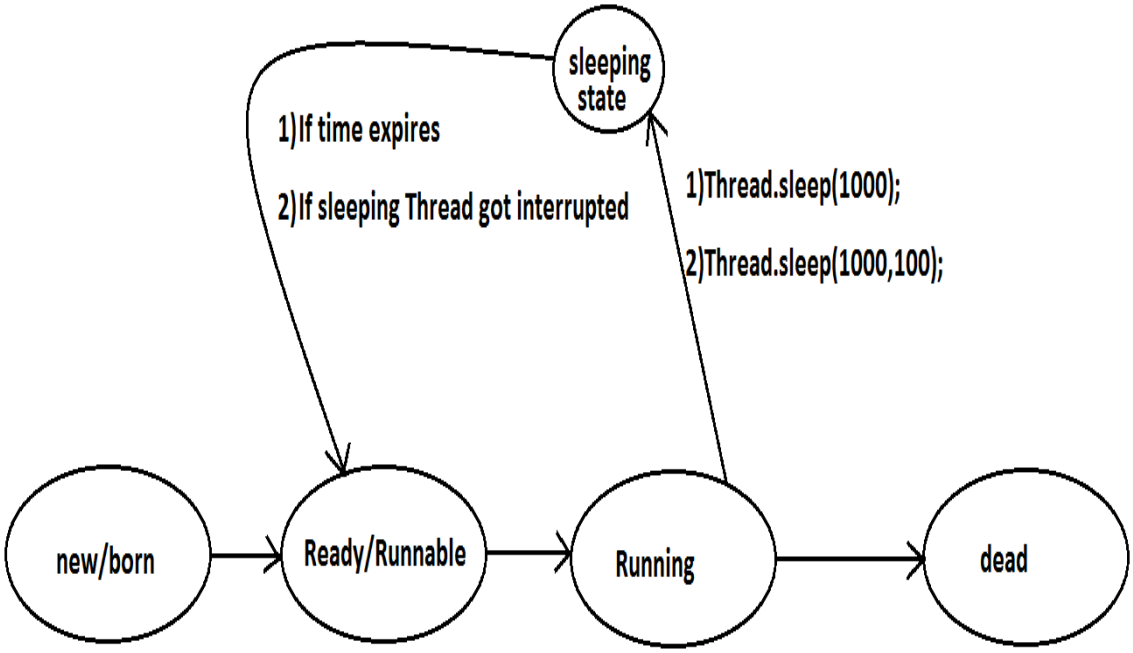
```
Example :

class ThreadDemo {
public static void main() throws InterruptedException {
Thread.currentThread().join();
      --------------   --------
        main             main
}

}
```

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:
class ThreadJoinDemo
{
        public static void main(String[] args)throws InterruptedException
        {
                System.out.println("M");
                Thread.sleep(3000);
                System.out.println("E");
                Thread.sleep(3000);
                System.out.println("G");
                Thread.sleep(3000);
                System.out.println("A");
        }
}
Output:
M
E
G
A
```

Interrupting a Thread:

How a Thread can interrupt another thread ?
If a Thread can interrupt a sleeping or waiting Thread by using interrupt()(break off) method of Thread class.
public void interrupt();

```
Example:
class MyThread extends Thread
```

```
{
        public void run()
        {
                try
                {
                        for(int i=0;i<5;i++)
                        {
                                System.out.println("i am lazy Thread :"+i);
                                Thread.sleep(2000);
                        }
                }
                catch (InterruptedException e)
                {
                        System.out.println("i got interrupted");
                }
        }
}
class ThreadInterruptDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
                //t.interrupt();       //--->1
                System.out.println("end of main thread");
        }
}
```

- **If we are commenting line 1 then main Thread won't interrupt child Thread and hence child Thread will be continued until its completion.**
- **If we are not commenting line 1 then main Thread interrupts child Thread and hence child Thread won't continued until its completion in this case the output is:**

```
End of main thread
I am lazy Thread: 0
I got interrupted
```

Note:

- **Whenever we are calling interrupt() method we may not see the effect immediately, if the target Thread is in sleeping or waiting state it will be interrupted immediately.**
- **If the target Thread is not in sleeping or waiting state then interrupt call will wait until target Thread will enter into sleeping or waiting state. Once target Thread entered into sleeping or waiting state it will effect immediately.**
- **In its lifetime if the target Thread never entered into sleeping or waiting state then there is no impact of interrupt call simply interrupt call will be wasted.**

```
Example:
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<5;i++)
                {
                        System.out.println("iam lazy thread");
                }
```

```
                System.out.println("I'm entered into sleeping stage");
                try
                {
                        Thread.sleep(3000);
                }
                catch (InterruptedException e)
                {
                        System.out.println("i got interrupted");
                }
        }
}
class ThreadInterruptDemo1
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
                t.interrupt();
                System.out.println("end of main thread");
        }
}
```

- **In the above program interrupt() method call invoked by main Thread will wait until child Thread entered into sleeping state.**
- **Once child Thread entered into sleeping state then it will be interrupted immediately.**

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 date 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].**

```
Example:
class Display
{
        public synchronized void wish(String name)
        {
                for(int i=0;i<5;i++)
                {
                        System.out.print("good morning:");
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch (InterruptedException e)
                        {}
                        System.out.println(name);
                }
        }
}
class MyThread extends Thread
{
        Display d;
        String name;
        MyThread(Display d,String name)
        {
                this.d=d;
                this.name=name;
        }
        public void run()
        {
                d.wish(name);
```

```
        }
}
class SynchronizedDemo
{
        public static void main(String[] args)
        {
                Display d1=new Display();
                MyThread t1=new MyThread(d1,"dhoni");
                MyThread t2=new MyThread(d1,"yuvaraj");
                t1.start();
                t2.start();
        }
}
```

If we are not declaring wish() method as synchronized then both Threads will be executed simultaneously and we will get irregular output.

```
Output:
good morning:good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
dhoni
```

If we declare wish()method as synchronized then the Threads will be executed one by one that is until completing the 1st Thread the 2nd Thread will wait in this case we will get regular output which is nothing but

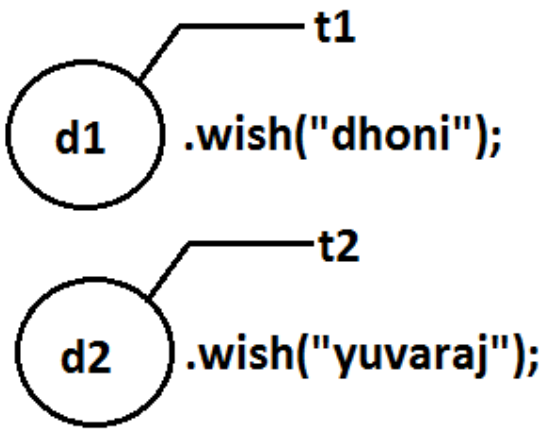
```
Output:
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
```

Case study:

Case 1:

```
Display d1=new Display();
Display d2=new Display();
MyThread t1=new MyThread(d1,"dhoni");
MyThread t2=new MyThread(d2,"yuvaraj");
t1.start();
t2.start();
```

Diagram:



Even though we declared wish() method as synchronized but we will get irregular output in this case, because both Threads are operating on different objects.

Conclusion : If multiple threads are operating on multiple objects then there is no impact of Syncronization.
If multiple threads are operating on same java objects then syncronized concept is required(applicable).

Class level lock:

1. **Every class in java has a unique lock. If a Thread wants to execute a static synchronized method then it required class level lock.**
2. **Once a Thread got class level lock then it is allow to execute any static synchronized method of that class.**
3. **While a Thread executing any static synchronized method the remaining Threads are not allow 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:

1. **If very few lines of the code required synchronization then it's never recommended to declare entire method as synchronized we have to enclose those few lines of the code with in synchronized block.**
2. **The main advantage of synchronized block over synchronized method is it reduces waiting time of Thread and improves performance of the system.**

Example 1: **To get lock of current object we can declare synchronized block as follows.**
If Thread got lock of current object then only it is allowed to execute this block.
Synchronized(this){}

Example 2: **To get the lock of a particular object 'b' we have to declare a synchronized block as follows.**
If thread got lock of 'b' object then only it is allowed to execute this block.
Synchronized(b){}

Example 3: **To get class level lock we have to declare synchronized block as follows.**
Synchronized(Display.class){}
If thread got class level lock of Display then only it allowed to execute this block.

Note:As the argument to the synchronized block we can pass either object reference or ".class file" and we can't pass primitive values as argument [because lock concept is dependent only for objects and classes but not for primitives].

```
Example:
Int x=b;
Synchronized(x){}
Output:
Compile time error.
Unexpected type.
Found: int
Required: reference
```

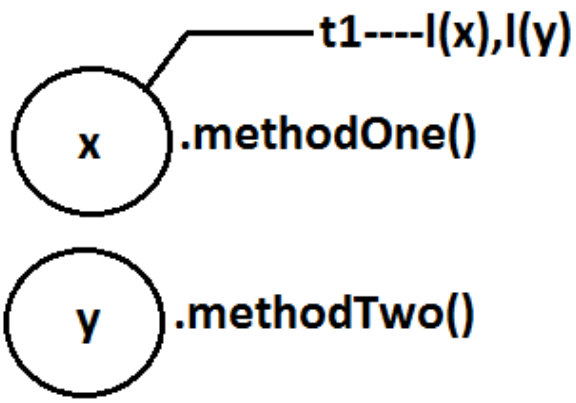
Questions:

1. **Explain about synchronized keyword and its advantages and disadvantages?**
2. **What is object lock and when a Thread required?**
3. **What is class level lock and when a Thread required?**
4. **What is the difference between object lock and class level lock?**
5. **While a Thread executing a synchronized method on the given object is the remaining Threads are allowed to execute other synchronized methods simultaneously on the same object?**
   Ans: No.
6. **What is synchronized block and explain its declaration?**
7. **What is the advantage of synchronized block over synchronized method?**
8. **Is a Thread can hold more than one lock at a time?**
   Ans: Yes, up course from different objects. Example:

```
class X
{
    synchronized void methodOne()
    {
        Y y=new Y();
        y.methodTwo();
    }
}
```

```
class Y
{
    synchronized void methodTwo()
    {}
}
```

Diagram:

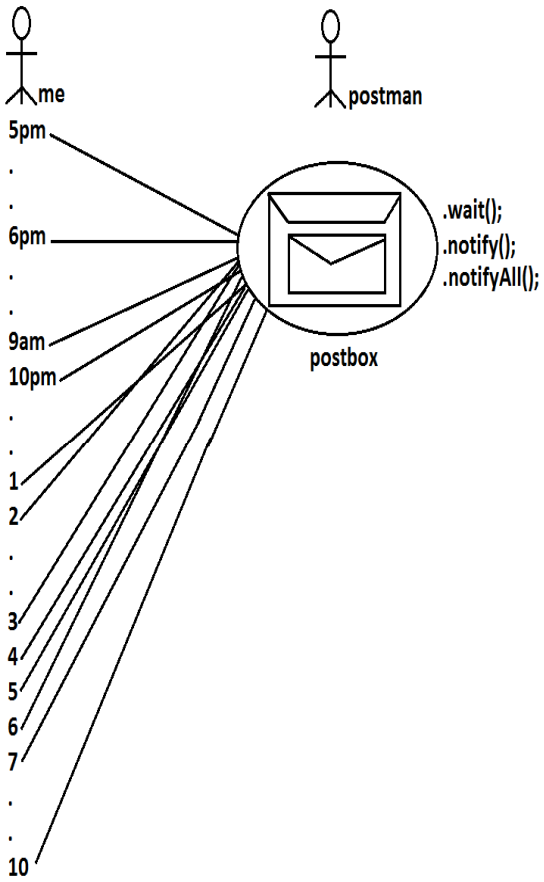


9. **What is synchronized statement?**
   Ans: The statements which present inside synchronized method and synchronized block are called synchronized statements. [Interview people created terminology].

Inter Thread communication (wait(),notify(), notifyAll()):

- **Two Threads can communicate with each other by using wait(), notify() and notifyAll() methods.**
- **The Thread which is required updation it has to call wait() method on the required object then immediately the Thread will entered into waiting state. The Thread which is performing updation of object, it is responsible to give notification by calling notify() method.**
  After getting notification the waiting Thread will get those updations.

  Diagram:



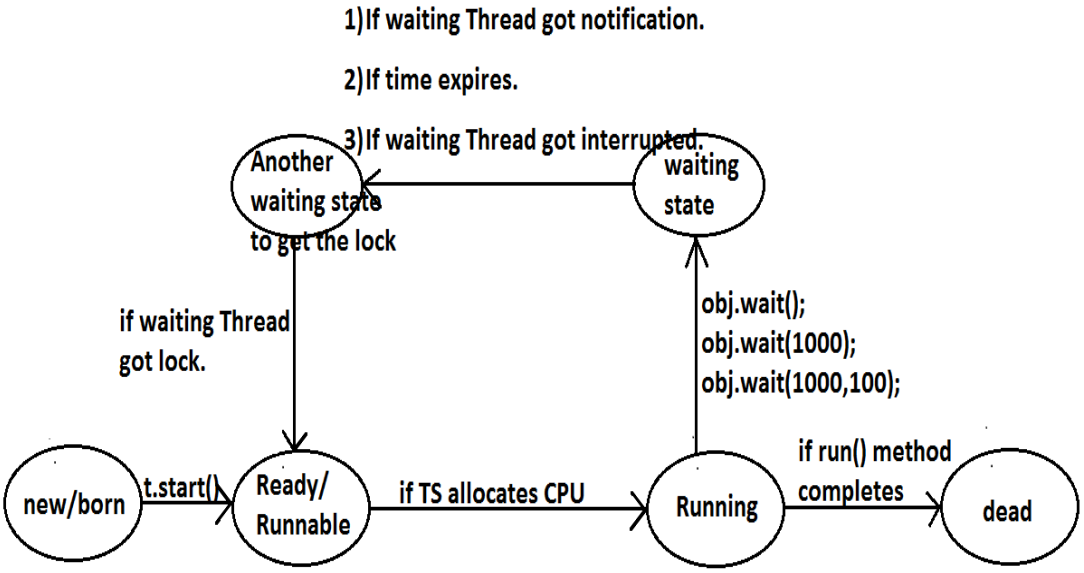
- **wait(), notify() and notifyAll() methods are available in Object class but not in Thread class because Thread can call these methods on any common object.**
- **To call wait(), notify() and notifyAll() methods compulsory the current Thread should be owner of that object**
  i.e., current Thread should has lock of that object
  i.e., current Thread should be in synchronized area. Hence we can call wait(), notify() and notifyAll() methods only from synchronized area otherwise we will get runtime exception saying IllegalMonitorStateException.
- **Once a Thread calls wait() method on the given object 1st it releases the lock of that object immediately and entered into waiting state.**
- **Once a Thread calls notify() (or) notifyAll() methods it releases the lock of that object but may not immediately.**
- **Except these (wait(),notify(),notifyAll()) methods there is no other place(method) where the lock release will be happen.**

Method	Is Thread Releases Lock?
yield()	No
join()	No
sleep()	No
wait()	Yes

notify()	**Yes**
notifyAll()	**Yes**

- **Once a Thread calls wait(), notify(), notifyAll() methods on any object then it releases the lock of that particular object but not all locks it has.**
  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()**

Diagram:




```
Example 1:
class ThreadA
{
        public static void main(String[] args)throws InterruptedException
        {
        ThreadB b=new ThreadB();
        b.start();
        synchronized(b)
        {
        System.out.println("main Thread calling wait() method");//step-1
        b.wait();
        System.out.println("main Thread got notification call");//step-4
        System.out.println(b.total);
        }
        }
}
class ThreadB extends Thread
{
        int total=0;
        public void run()
        {
        synchronized(this)
```

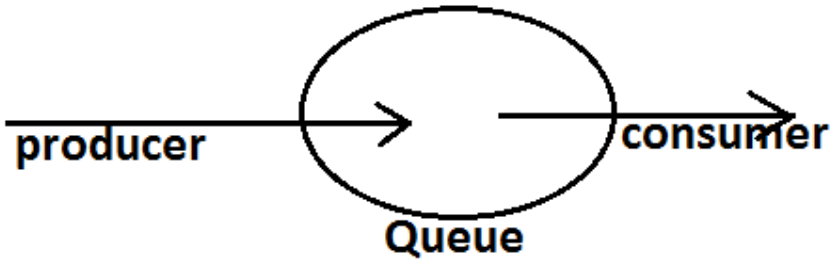
```
        {
        System.out.println("child thread starts calcuation");//step-2
                for(int i=0;i<=100;i++)
                {
                        total=total+i;
                }
        System.out.println("child thread giving notification call");//step-
3
        this.notify();
        }
        }
}
Output:
main Thread calling wait() method
child thread starts calculation
child thread giving notification call
main Thread got notification call
5050
```

Example 2:

Producer consumer problem:

- **Producer(producer Thread) will produce the items to the queue and consumer(consumer thread) will consume the items from the queue. If the queue is empty then consumer has to call wait() method on the queue object then it will entered into waiting state.**
- **After producing the items producer Thread call notify() method on the queue to give notification so that consumer Thread will get that notification and consume items.**

Diagram:



Example:

```
class Producer
{
    Producer()
    {
        synchronized(q)
        {
            produce items to the queue
                q.notify();
        }
    }
}
```

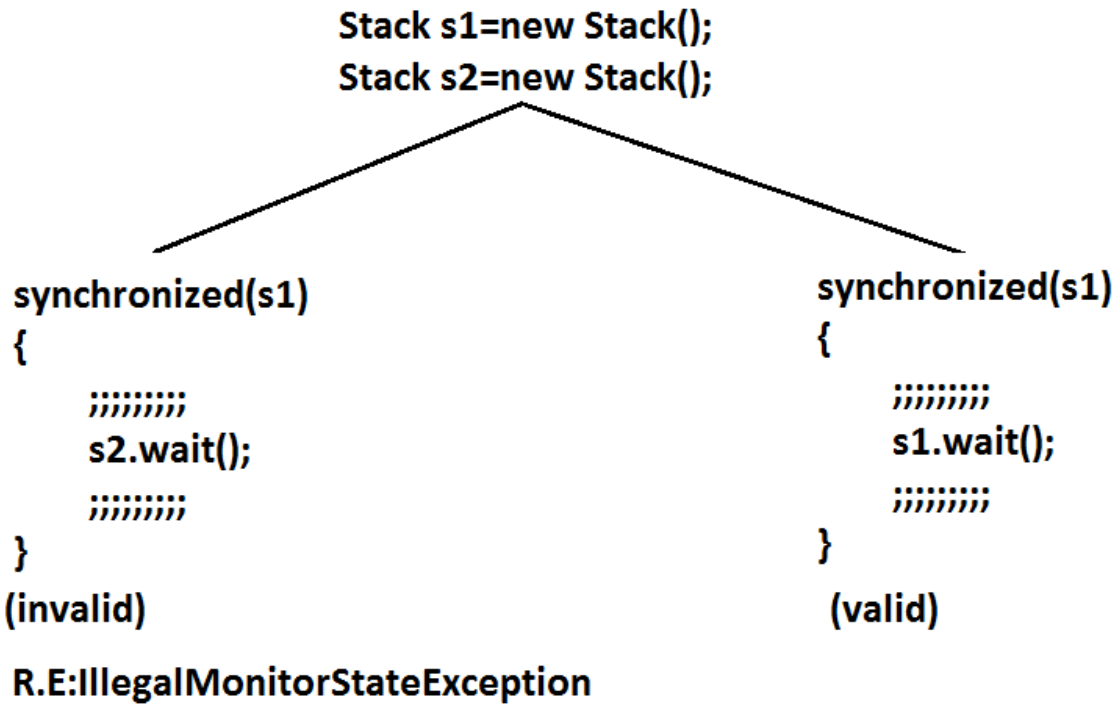
```
{
    Consumer()
    {
        synchronized(q)
        {
            if(q is empty)
            {
                q.wait();
            }
            else
                continue items;
        }
    }
}
```

Notify vs notifyAll():

- **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 has to wait for further notification. But which Thread will be notify(inform) we can't expect exactly it depends on JVM.**
- **We can use notifyAll() method to give the notification for all waiting Threads. All waiting Threads will be notified and will be executed one by one, because they are required 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.

Example:



Which of the folowing statements are True ?

1. **Once a Thread calls wait() on any Object immediately it will entered into waiting state without releasing the lock ?**
   NO
2. **Once a Thread calls wait() on any Object it reduces the lock of that Object but may not immediately ?**
   NO
3. **Once a Thread calls wait() on any Object it immediately releases all locks whatever it has and entered into waiting state ?**
   NO
4. **Once a Thread calls wait() on any Object it immediately releases the lock of that perticular Object and entered into waiting state ?**
   YES
5. **Once a Thread calls notify() on any Object it immediately releases the lock of that Object ?**
   NO
6. **Once a Thread calls notify() on any Object it releases the lock of that Object but may not immediately ?**
   YES

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.

```
Example:
class A
{
        public synchronized void foo(B b)
        {
                System.out.println("Thread1 starts execution of foo()
method");
                try
                {
                        Thread.sleep(2000);
                }
                catch (InterruptedException e)
                {}
                System.out.println("Thread1 trying to call b.last()");
                b.last();
        }
        public synchronized void last()
        {
                System.out.println("inside A, this is last()method");
        }
}
class B
{
        public synchronized void bar(A a)
        {
        System.out.println("Thread2 starts execution of bar() method");
        try
        {
                Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {}
        System.out.println("Thread2 trying to call a.last()");
        a.last();
        }
        public synchronized void last()
        {
        System.out.println("inside B, this is last() method");
        }
}
class DeadLock implements Runnable
{
        A a=new A();
        B b=new B();
        DeadLock()
        {
                Thread t=new Thread(this);
                t.start();
                a.foo(b);//main thread
```

```
        }
        public void run()
        {
                b.bar(a);//child thread
        }
        public static void main(String[] args)
        {
                new DeadLock();//main thread
        }
}
Output:
Thread1 starts execution of foo() method
Thread2 starts execution of bar() method
Thread2 trying to call a.last()
Thread1 trying to call b.last()
//here cursor always waiting.
```

Note : If we remove atleast one syncronized keywoed then we won't get DeadLOck.Hence syncronized keyword in the only reason for DeadLock due to this while using syncronized keyword we have to handling carefully.

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.**

```
Example:
class MyThread extends Thread
{
```

```
}

class DaemonThreadDemo
{
        public static void main(String[] args)
        {
        System.out.println(Thread.currentThread().isDaemon());
        MyThread t=new MyThread();
        System.out.println(t.isDaemon());               1
        t.start();
        t.setDaemon(true);
        System.out.println(t.isDaemon());
        }
}
Output:
false
false
RE:IllegalThreadStateException
Example:
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<10;i++)
                {
                        System.out.println("lazy thread");
                        try
                        {
                                Thread.sleep(2000);
                        }
                        catch (InterruptedException e)
                        {}
                }
        }
}
class DaemonThreadDemo
{
        public static void main(String[] args)
        {
        MyThread t=new MyThread();
        t.setDaemon(true);     //-->1
        t.start();
        System.out.println("end of main Thread");
        }
}
Output:
End of main Thread
```

- **If we comment line 1 then both main & child Threads are non-Daemon , and hence both threads will be executed untill there completion.**
- **If we are not comment line 1 then main thread is non-Daemon and child thread is Daemon. Hence when ever main Thread terminates automatically child thread will be terminated.**

Lazy thread

- **If we are commenting line 1 then both main and child Threads are non daemon and hence both will be executed until they completion.**
- **If we are not commenting line 1 then main Thread is non daemon and child Thread is daemon and hence whenever main Thread terminates automatically child Thread will be terminated.**

Deadlock vs Starvation:

- **A long waiting of a Thread which never ends is called deadlock.**
- **A long waiting of a Thread which ends at certain point is called starvation.**
- **A low priority Thread has to wait until completing all high priority Threads.**
- **This long waiting of Thread which ends at certain point is called starvation.**

How to kill a Thread in the middle of the line?

- **We can call stop() method to stop a Thread in the middle then it will be entered into dead state immediately.**
  public final void stop();
- **stop() method has been deprecated and hence not recommended to use.**

suspend and resume methods:

- **A Thread can suspend another Thread by using suspend() method then that Thread will be paused temporarily.**
- **A Thread can resume a suspended Thread by using resume() method then suspended Thread will continue its execution.**
  1. **public final void suspend();**
  2. **public final void resume();**
- **Both methods are deprecated and not recommended to use.**

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.

ThreadGroup:

Based on functionality we can group threads as a single unit which is nothing but ThreadGroup.

ThreadGroup provides a convenient way to perform common operations for all threads belongs to a perticular group.

We can create a ThreadGroup by using the following constructors
ThreadGroup g=new ThreadGroup(String gName);

We can attach a Thread to the ThreadGroup by using the following constructor of Thread class
Thread t=new Thread(ThreadGroup g, String name);

```
ThreadGroup g=new ThreadGroup("Printing Threads");
MyThread t1=new MyThread(g,"Header Printing");
MyThread t2=new MyThread(g,"Footer Printing");
MyThread t3=new MyThread(g,"Body Printing");
----------
g.stop();
```

ThreadLocal(1.2 v):

We can use ThreadLocal to define local resources which are required for a perticular Thread like DBConnections, counterVariables etc.,

We can use ThreadLocal to define Thread scope like Servlet Scopes(page,request,session,application).

GreenThread:

Java multiThreading concept is implementing by using the following 2 methods :

1. **GreenThread Model**
2. **Native OS Model**

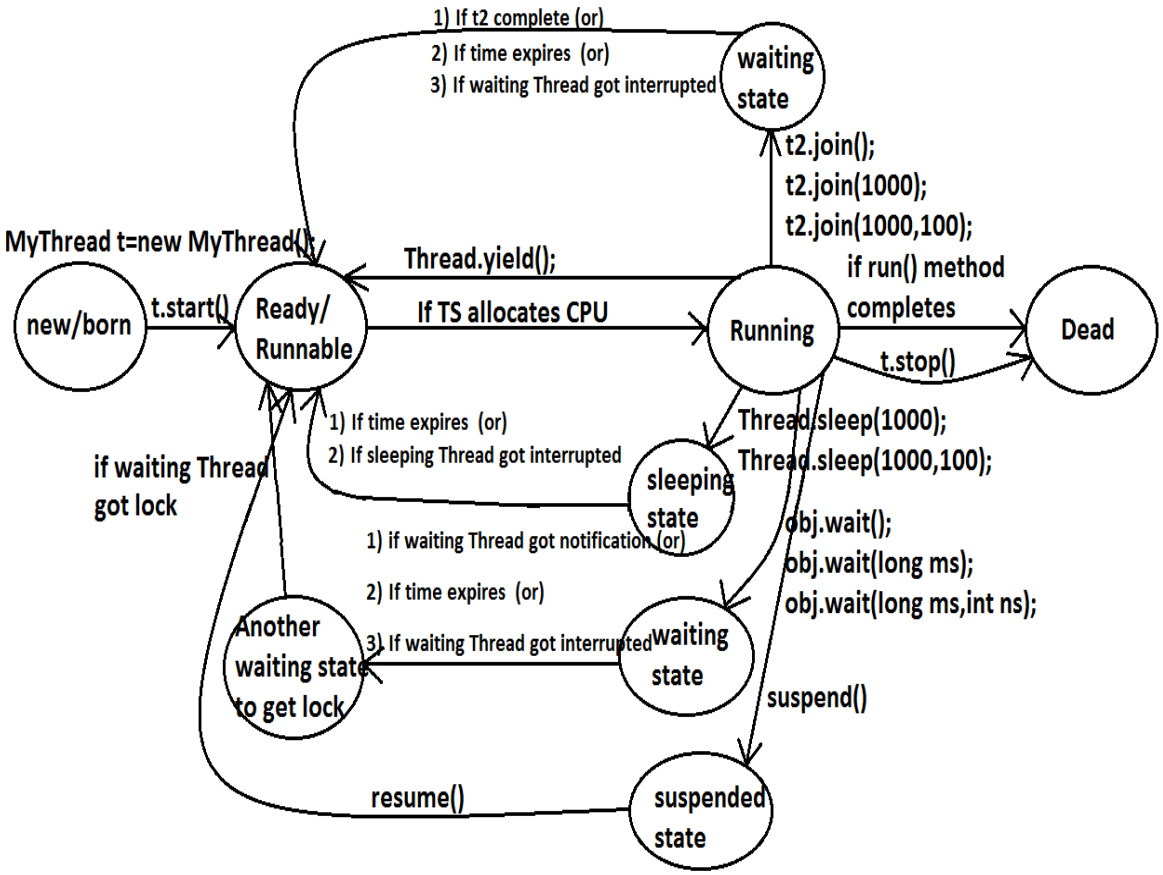
GreenThread Model

The threads which are managed completely by JVM without taking support for underlying OS, such type of threads are called Green Threads.

Native OS Model

- **The Threads which are managed with the help of underlying OS are called Native Threads.**
- **Windows based OS provide support for Native OS Model**
- **Very few OS like SunSolaries provide support for GreenThread Model**
- **Anyway GreenThread model is deprecated and not recommended to use.**

Life cycle of a Thread:



What is the difference between extends Thread and implements Runnable?

1. **Extends Thread is useful to override the public void run() method of Thread class.**
2. **Implements Runnable is useful to implement public void run() method of Runnable interface.**

Extends Thread, implements Runnable which one is advantage?
If we extend Thread class, there is no scope to extend another class.

Example:
Class MyClass extends Frame,Thread//invalid
If we write implements Runnable still there is a scope to extend one more class.

Example:

1. **class MyClass extends Thread implements Runnable**
2. **class MyClass extends Frame implements Runnable**

How can you stop a Thread which is running?

Step 1: **Declare a boolean type variable and store false in that variable.**
boolean stop=false;

Step 2: **If the variable becomes true return from the run() method.**
If(stop) return;

Step 3: **Whenever to stop the Thread store true into the variable.**
System.in.read();//press enter
Obj.stop=true;

Questions:

1. **What is a Thread?**
2. **Which Thread by default runs in every java program?**
   Ans: By default main Thread runs in every java program.
3. **What is the default priority of the Thread?**
4. **How can you change the priority number of the Thread?**
5. **Which method is executed by any Thread?**
   Ans: A Thread executes only public void run() method.
6. **How can you stop a Thread which is running?**
7. **Explain the two types of multitasking?**
8. **What is the difference between a process and a Thread?**
9. **What is Thread scheduler?**
10. **Explain the synchronization of Threads?**
11. **What is the difference between synchronized block and synchronized keyword?**
12. **What is Thread deadlock? How can you resolve deadlock situation?**
13. **Which methods are used in Thread communication?**
14. **What is the difference between notify() and notifyAll() methods?**
15. **What is the difference between sleep() and wait() methods?**
16. **Explain the life cycle of a Thread?**
17. **What is daemon Thread?**

CORE JAVA

With

SCJP / OCJP

Study Material

Chapter 8: Multi Threading Enhancements

DURGA M.Tech

(Sun certified & Realtime Expert)

DURGA SOFTWARE SOLUTIONS

www.durgasoft.com Ph: 9246212143 8096969696

Multi Threading Enhancements

8.1) ThreadGroup

8.2) ThreadLocal

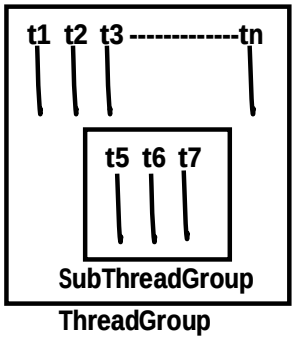
8.3) java.util.concurrent.locks package
->Lock(I)
->ReentrantLock(C)

8.4) Thread Pools

8.5) Callable and Future

ThreadGroup:

- **Based on the Functionality we can Group Threads into a Single Unit which is Nothing but ThreadGroup i.e. ThreadGroup Represents a Set of Threads.**
- **In Addition a ThreadGroup can Also contains Other SubThreadGroups.**



- **ThreadGroup Class Present in java.lang Package and it is the Direct Child Class of Object.**
- **ThreadGroup provides a Convenient Way to Perform Common Operation for all Threads belongs to a Particular Group.**

Eg:Stop All Consumer Threads.
Suspend All Producer Threads.

Constructors:

1) ThreadGroup g = new ThreadGroup(String gname);

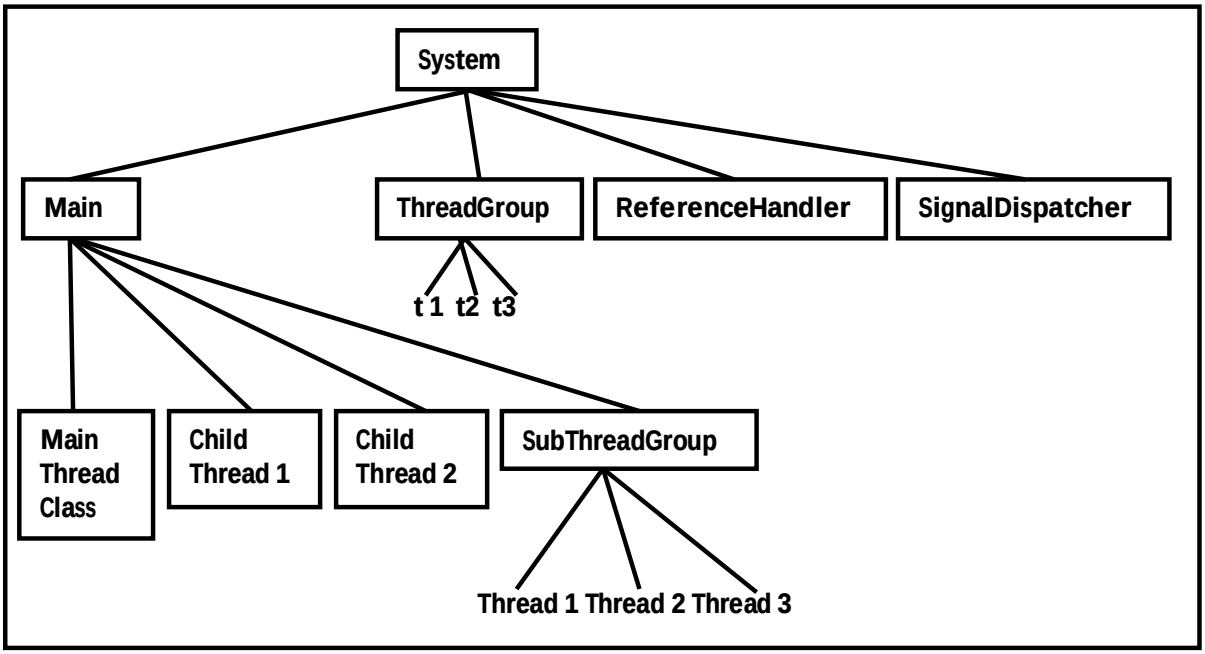
- **Createsa New ThreadGroup.**
- **The Parent of this New Group is the ThreadGroup of Currently Running Thread.**

2) ThreadGroup g = new ThreadGroup(ThreadGrouppg, String gname);

- **Creates a New ThreadGroup.**
- **The Parent of this ThreadGroup is the specified ThreadGroup.**

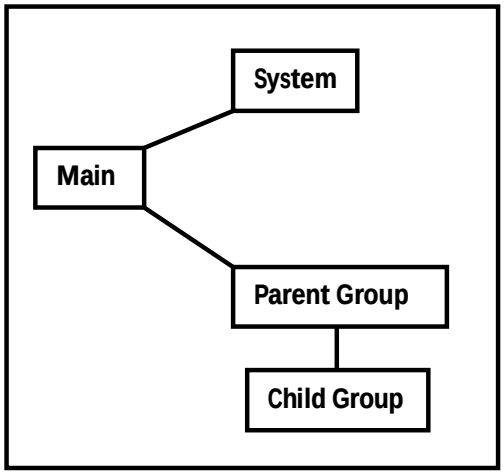
Note:

- **In Java Every Thread belongs to Some Group.**
- **Every ThreadGroup is the Child Group of *System Group* either Directly OR Indirectly. Hence SystemGroup Acts as Root for all ThreadGroup's in Java.**
- **System ThreadGroup Represents System Level Threads Like ReferenceHandler, SignalDispatcher, Finalizer, AttachListener Etc.**



```
classThreadGroupDemo {
public static void main(String[] args) {
        System.out.println(Thread.currentThread().getThreadGroup().getName());
        System.out.println(Thread.currentThread().getThreadGroup().getParent().getName());
        ThreadGrouppg = new ThreadGroup("Parent Group");
        System.out.println(pg.getParent().getName());
        ThreadGroup cg = new ThreadGroup(pg, "Child Group");
        System.out.println(cg.getParent().getName());
}
}
```

```
main
system
main
Parent Group
```



Important Methods of ThreadGroup Class:

1) **String getName();** Returns Name of the ThreadGroup.

2) **intgetMaxPriority();** Returns the Maximum Priority of ThreadGroup.

3) **void setMaxPriority();**
   - To Set Maximum Priority of ThreadGroup.
   - The Default Maximum Priority is 10.
   - Threads in the ThreadGroup that Already have Higher Priority, Not effected but Newly Added Threads this MaxPriority is Applicable.

```
classThreadGroupDemo {
        public static void main(String[] args) {
                ThreadGroup g1 = new ThreadGroup("tg");
                Thread t1 = new Thread(g1, "Thread 1");
                Thread t2 = new Thread(g1, "Thread 2");
                g1.setMaxPriority(3);
                Thread t3 = new Thread(g1, "Thread 3");
                System.out.println(t1.getPriority());→ 5
                System.out.println(t2.getPriority());→ 5
                System.out.println(t3.getPriority());→ 3
        }
}
```

4) **ThreadGroupgetParent(): Returns Parent Group of Current ThreadGroup.**

5) **void list(): It Prints Information about ThreadGroup to the Console.**

6) **intactiveCount(): Returns Number of Active Threads Present in the ThreadGroup.**

7) **intactiveGroupCount(): It Returns Number of Active ThreadGroups Present in the Current ThreadGroup.**

8) **int enumerate(Thread[] t): To Copy All Active Threads of this Group into provided Thread Array. In this Case SubThreadGroup Threads also will be Considered.**

9) **int enumerate(ThreadGroup[] g): To Copy All Active SubThreadGroups into ThreadGroupArray.**

10) **booleanisDaemon():**

11) **void setDaemon(boolean b):**

12) **void interrupt(): To Interrupt All Threads Present in the ThreadGroup.**

13) **void destroy(): To Destroy ThreadGroup and its SubThreadGroups.**

```
classMyThread extends Thread {
        MyThread(ThreadGroup g, String name) {
                super(g, name);
        }
        public void run() {
        System.out.println("Child Thread");
        try {
                Thread.sleep(2000);
          }
        catch (InterruptedException e) {}
        }
}
classThreadGroupDemo {
        public static void main(String[] args) throws InterruptedException {
                ThreadGrouppg = new ThreadGroup("Parent Group");
                ThreadGroup cg = new ThreadGroup(pg, "Child Group");
                MyThread t1 = new MyThread(pg, "Child Thread 1");
                MyThread t2 = new MyThread(pg, "Child Thread 2");
                t1.start();
                t2.start();
                System.out.println(pg.activeCount());
                System.out.println(pg.activeGroupCount());
                pg.list();
                Thread.sleep(5000);
                System.out.println(pg.activeCount());
                pg.list();
        }
}
```

```
2
1
java.lang.ThreadGroup[name=Parent Group,maxpri=10]
Thread[Child Thread 1,5,Parent Group]
Thread[Child Thread 2,5,Parent Group]
java.lang.ThreadGroup[name=Child Group,maxpri=10]
Child Thread
Child Thread
0
java.lang.ThreadGroup[name=Parent Group,maxpri=10]
java.lang.ThreadGroup[name=Child Group,maxpri=10]
```

Write a Program to Display All Thread Names belongs to System Group

```
classThreadGroupDemo {
public static void main(String[] args) {
        ThreadGroup system = Thread.currentThread().getThreadGroup().getParent();
        Thread[] t = new Thread[system.activeCount()];
        system.enumerate(t);
        for (Thread t1: t) {
                System.out.println(t1.getName()+"-------"+t1.isDaemon());
        }
}
}
```

```
Reference Handler-------true
Finalizer-------true
Signal Dispatcher-------true
Attach Listener-------true
main-------false
```

ThreadLocal:

- **ThreadLocalProvides ThreadLocal Variables.**
- **ThreadLocal Class Maintains Values for Thread Basis.**
- **Each ThreadLocal Object Maintains a Separate Value Like userID, transactionIDEtc for Each Thread that Accesses that Object.**
- **Thread can Access its Local Value, can Manipulates its Value and Even can Remove its Value.**
- **In Every Part of the Code which is executed by the Thread we can Access its Local Variables.**

Eg:

- **Consider a Servlet which Calls Some Business Methods. we have a Requirement to generate a Unique transactionID for Each and Every Request and we have to Pass this transactionID to the Business Methods for Logging Purpose.**
- **For this Requirement we can Use ThreadLocal to Maintain a Separate transactionID for Every Request and for Every Thread.**

Note:

- **ThreadLocal Class introduced in 1.2 Version.**
- **ThreadLocal can be associated with Thread Scope.**
- **All the Code which is executed by the Thread has Access to Corresponding ThreadLocal Variables.**
- **A Thread can Access its Own Local Variables and can't Access Other Threads Local Variables.**
- **Once Thread Entered into Dead State All Local Variables are by Default Eligible for Garbage Collection.**

Constructor:ThreadLocaltl = new ThreadLocal();
Creates a ThreadLocal Variable.

Methods:

1) **Object get(); Returns the Value of ThreadLocal Variable associated with Current Thread.**

2) **Object initialValue();**
   - **Returns the initialValue of ThreadLocal Variable of Current Thread.**
   - **The Default Implementation of initialValue() Returns null.**
   - **To Customize Our initialValue we have to Override initialValue().**

3) **void set(Object newValue);To Set a New Value.**

4) **void remove();**
   - **To Remove the Current Threads Local Variable Value.**
   - **After Remove if we are trying to Access it will be reinitialized Once Again by invoking its initialValue().**
   - **This Method Newly Added in 1.5 Version.**

```
classThreadLocalDemo {
public static void main(String[] args) {
        ThreadLocaltl = new ThreadLocal();
        System.out.println(tl.get()); //null
        tl.set("Durga");
        System.out.println(tl.get()); //Durga
        tl.remove();
        System.out.println(tl.get()); //null
}
}
```

```
//Overriding of intialValue()
classThreadLocalDemo {
public static void main(String[] args) {
        ThreadLocaltl = new ThreadLocal() {
        protected Object initialValue() {
                return "abc";
}
        };
        System.out.println(tl.get()); //abc
        tl.set("Durga");
        System.out.println(tl.get()); //Durga
        tl.remove();
        System.out.println(tl.get()); //abc
}
}
```

```
classCustomerThread extends Thread {
        static Integer custID = 0;
        private static ThreadLocaltl = new ThreadLocal() {
        protected Integer initialValue() {
        return ++custID;
             }
        };
        CustomerThread(String name) {
        super(name);
        }
        public void run() {
        for (int i=0; i<5; i++) {
SOP(Thread.currentThread().getName()+" Executing with Customer ID:"+tl.get());
             }
        }
}
classThreadLocalDemo {
        public static void main(String[] args) {
                CustomerThread c1 = new CustomerThread("CustomerThread - 1");
                CustomerThread c2 = new CustomerThread("CustomerThread - 2");
                CustomerThread c3 = new CustomerThread("CustomerThread - 3");
                CustomerThread c4 = new CustomerThread("CustomerThread - 4");
                c1.start();
                c2.start();
                c3.start();
                c4.start();
        }
}
```

```
CustomerThread - 1 Executing with Customer ID:1
CustomerThread - 1 Executing with Customer ID:1
CustomerThread - 1 Executing with Customer ID:1
CustomerThread - 1 Executing with Customer ID:1
CustomerThread - 1 Executing with Customer ID:1

CustomerThread - 2 Executing with Customer ID:2
CustomerThread - 2 Executing with Customer ID:2
CustomerThread - 2 Executing with Customer ID:2
CustomerThread - 2 Executing with Customer ID:2
CustomerThread - 2 Executing with Customer ID:2

CustomerThread - 3 Executing with Customer ID:3
CustomerThread - 3 Executing with Customer ID:3
CustomerThread - 3 Executing with Customer ID:3
CustomerThread - 3 Executing with Customer ID:3
CustomerThread - 3 Executing with Customer ID:3

CustomerThread - 4 Executing with Customer ID:4
CustomerThread - 4 Executing with Customer ID:4
CustomerThread - 4 Executing with Customer ID:4
CustomerThread - 4 Executing with Customer ID:4
CustomerThread - 4 Executing with Customer ID:4
CustomerThread - 4 Executing with Customer ID:4
```

In the Above Program for Every Customer Thread a Separate customerID will be maintained by ThreadLocal Object.

ThreadLocalVs Inheritance:

- **Parent Threads ThreadLocal Variables are by Default Not Available to the Child Threads.**
- **If we want to Make Parent Threads Local Variables Available to Child Threads we should go for InheritableThreadLocal Class.**
- **It is the Child Class of ThreadLocal Class.**
- **By Default Child Thread Values are Same as Parent Thread Values but we can Provide Customized Values for Child Threads by Overriding childValue().**

Constructor:InheritableThreadLocalitl = new InheritableThreadLocal();
InheritableThreadLocal is the Child Class of ThreadLocal and Hence All Methods Present in ThreadLocal by Default Available to the InheritableThreadLocal.

Method: public Object childValue(Object pvalue);

```
classParentThread extends Thread {
public static InheritableThreadLocalitl = new InheritableThreadLocal() {
        public Object childValue(Object p) {
        return "cc";
        }
};
public void run() {
        itl.set("pp");
        System.out.println("Parent Thread --"+itl.get());
        ChildThreadct = new ChildThread();
        ct.start();
}
classChildThread extends Thread {
        public void run() {
                System.out.println("Child Thread --"+ParentThread.itl.get());
        }
}
}
classThreadLocalDemo {
        public static void main(String[] args) {
                ParentThreadpt = new ParentThread();
                pt.start();
        }
}
```

```
Parent Thread --pp
Child Thread --cc
```

Java.util.concurrent.locks package:

Problems with Traditional synchronized Key Word

- If a Thread Releases the Lock then which waiting Thread will get that Lock we are Not having any Control on this.
- We can't Specify Maximum waiting Time for a Thread to get Lock so that it will Wait until getting Lock, which May Effect Performance of the System and Causes Dead Lock.
- We are Not having any Flexibility to Try for Lock without waiting.
- There is No API to List All Waiting Threads for a Lock.
- The synchronized Key Word Compulsory we have to Define within a Method and it is Not Possible to Declare Over Multiple Methods.
- To Overcome Above Problems SUN People introduced *java.util.concurrent.locks* Package in 1.5 Version.
- It Also Provides Several Enhancements to the Programmer to Provide More Control on Concurrency.

Lock(I):

- A Lock Object is Similar to Implicit Lock acquired by a Thread to Execute synchronized Method OR synchronized Block
- Lock Implementations Provide More Extensive Operations than Traditional Implicit Locks.

Important Methods of Lock Interface

1) **void lock();**
   - It Locks the Lock Object.
   - If Lock Object is Already Locked by Other Thread then it will wait until it is Unlocked.

2) **booleantryLock();**
   - To Acquire the Lock if it is Available.
   - If the Lock is Available then Thread Acquires the Lock and Returns true.
   - If the Lock Unavailable then this Method Returns false and Continue its Execution.
   - In this Case Thread is Never Blocked.

```
if (l.tryLock()) {
Perform Safe Operations
}
else {
   Perform Alternative Operations
}
```

3) **boolentryLock(long time, TimeUnit unit);**
   - **To Acquire the Lock if it is Available.**
   - **If the Lock is Unavailable then Thread can Wait until specified Amount of Time.**
   - **Still if the Lock is Unavailable then Thread can Continue its Execution.**

   Eg: if (l.tryLock(1000, TimeUnit.SECONDS)) {}

TimeUnit:TimeUnit is an *enum* Present in *java.util.concurrent* Package.

```
enumTimeUnit {
        DAYS, HOURS, MINUTES, SECONDS, MILLI SECONDS, MICRO SECONDS, NANO SECONDS;
}
```

4) **void lockInterruptedly();**

   Acquired the Lock Unless the Current Thread is Interrupted.
   Acquires the Lock if it is Available and Returns Immediately.
   If it is Unavailable then the Thread will wait while waiting if it is Interrupted then it won't get the Lock.

5) **void unlock(); To Release the Lock.**

ReentrantLock

- **It implements Lock Interface and it is the Direct Child Class of an Object.**
- **Reentrant Means a Thread can acquires Same Lock Multiple Times without any Issue.**
- **Internally ReentrantLock Increments Threads Personal Count whenever we Call lock() and Decrements Counter whenever we Call unlock() and Lock will be Released whenever Count Reaches '0'.**

Constructors:

1) **ReentrantLockrl = new ReentrantLock();**
   Creates an Instance of ReentrantLock.

2) **ReentrantLockrl = new ReentrantLock(boolean fairness);**
   - **Creates an Instance of ReentrantLock with the Given Fairness Policy.**
   - **If Fairness is true then Longest Waiting Thread can acquired Lock Once it is Avaiable i.e. if follows First - In First – Out.**
   - **If Fairness is false then we can't give any Guarantee which Thread will get the Lock Once it is Available.**

   Note: If we are Not specifying any Fairness Property then by Default it is Not Fair.

Which of the following 2 Lines are Equal?

```
ReentrantLockrl = new ReentrantLock(); √

ReentrantLockrl = new ReentrantLock(true);

ReentrantLockrl = new ReentrantLock(false); √
```

Important Methods of ReentrantLock

1) **void lock();**

2) **booleantryLock();**

3) **booleantryLock(long l, TimeUnit t);**

4) **void lockInterruptedly();**

5) **void unlock();**
   - **To Realease the Lock.**
   - **If the Current Thread is Not Owner of the Lock then we will get Runtime Exception Saying IllegalMonitorStateException.**

6) **intgetHoldCount(); Returns Number of Holds on this Lock by Current Thread.**

7) **booleanisHeldByCurrentThread(); Returns true if and Only if Lock is Hold by Current Thread.**

8) **intgetQueueLength(); Returns the Number of Threads waiting for the Lock.**

9) **Collection getQueuedThreads(); Returns a Collection containing Thread Objects which are waiting to get the Lock.**

10) **booleanhasQueuedThreads(); Returns true if any Thread waiting to get the Lock.**

11) **booleanisLocked(); Returns true if the Lock acquired by any Thread.**

12) **booleanisFair();Returns true if the Lock's Fairness Set to true.**

13) **Thread getOwner();Returns the Thread which acquires the Lock.**

```
importjava.util.concurrent.locks.ReentrantLock;
class Test {
        public static void main(String[] args) {
                ReentrantLock l = new ReentrantLock();
                l.lock();

                l.lock();
                System.out.println(l.isLocked()); //true
                System.out.println(l.isHeldByCurrentThread()); //true
                System.out.println(l.getQueueLength()); //0

                l.unlock();
                System.out.println(l.getHoldCount()); //1
                System.out.println(l.isLocked()); //true

                l.unlock();
                System.out.println(l.isLocked()); //false
                System.out.println(l.isFair()); //false
        }
}
```

```
importjava.util.concurrent.locks.ReentrantLock;
class Display {
ReentrantLock l = new ReentrantLock(true);
public void wish(String name) {
        l.lock();→ 1
                for(int i=0; i<5; i++) {
                System.out.println("Good Morning");
                try {
                        Thread.sleep(2000);
                }
                catch(InterruptedException e) {}
                System.out.println(name);
        }
        l.unlock();→ 2
}
}
classMyThread extends Thread {
        Display d;
        String name;
        MyThread(Display d, String name) {
                this.d = d;
                this.name = name;
        }
        public void run() {
                d.wish(name);
        }
}
classReentrantLockDemo {
public static void main(String[] args) {
                Display d = new Display();
                MyThread t1 = new MyThread(d, "Dhoni");
                MyThread t2 = new MyThread(d, "Yuva Raj");
                MyThread t3 = new MyThread(d, "ViratKohli");
                t1.start();
                t2.start();
                t3.start();
        }
}
```

```
Good Morning
Dhoni
Good Morning
Dhoni
Good Morning
Dhoni
Good Morning
Dhoni
Good Morning
Dhoni
Good Morning
Yuva Raj
Good Morning
Yuva Raj
Good Morning
Yuva Raj
Good Morning
Yuva Raj
Good Morning
Yuva Raj
Good Morning
ViratKohli
Good Morning
ViratKohli
Good Morning
ViratKohli
Good Morning
ViratKohli
Good Morning
ViratKohli
```

If we Comment Both Lines 1 and 2 then All Threads will be executed Simultaneously and Hence we will get Irregular Output.

If we are Not Commenting then the Threads will be executed One by One and Hence we will get Regular Output

Demo Program To Demonstrate tryLock();

```
importjava.util.concurrent.locks.ReentrantLock;
classMyThread extends Thread {
staticReentrantLock l = new ReentrantLock();
MyThread(String name) {
        super(name);
}
public void run() {
        if(l.tryLock()) {
SOP(Thread.currentThread().getName()+" Got Lock and Performing Safe Operations");
        try {
                        Thread.sleep(2000);
        }
        catch(InterruptedException e) {}
        l.unlock();
        }
        else {
                        System.out.println(Thread.currentThread().getName()+" Unable To Get Lock
and Hence Performing Alternative Operations");
                }
        }
}
classReentrantLockDemo {
        public static void main(String args[]) {
                MyThread t1 = new MyThread("First Thread");
                MyThread t2 = new MyThread("Second Thread");
                t1.start();
                t2.start();
        }
}
```

```
First Thread Got Lock and Performing Safe Operations
Second Thread Unable To Get Lock and Hence Performing Alternative Operations
```

```
importjava.util.concurrent.TimeUnit;
importjava.util.concurrent.locks.ReentrantLock;
classMyThread extends Thread {
        staticReentrantLock l = new ReentrantLock();
        MyThread(String name) {
                super(name);
        }
        public void run() {
                do {
                        try {
                                if(l.tryLock(1000, TimeUnit.MILLISECONDS)) {
                                        SOP(Thread.currentThread().getName()+"------- Got Lock");
Thread.sleep(5000);
l.unlock();
SOP(Thread.currentThread().getName()+"------- Releases Lock");
break;
                                }
                                else {
SOP(Thread.currentThread().getName()+"------- Unable To Get Lock And Will Try Again");
                                }
                        }
                        catch(InterruptedException e) {}
                }
                while(true);
        }
}
classReentrantLockDemo {
        public static void main(String args[]) {
                MyThread t1 = new MyThread("First Thread");
                MyThread t2 = new MyThread("Second Thread");
                t1.start();
                t2.start();
        }
}
```

```
First Thread------- Got Lock
Second Thread------- Unable To Get Lock And Will Try Again
Second Thread------- Unable To Get Lock And Will Try Again
Second Thread------- Unable To Get Lock And Will Try Again
Second Thread------- Unable To Get Lock And Will Try Again
Second Thread------- Got Lock
First Thread------- Releases Lock
Second Thread------- Releases Lock
```

Thread Pools:

- **Creating a New Thread for Every Job May Create Performance and Memory Problems.**
- **To Overcome this we should go for Thread Pool Concept.**
- **Thread Pool is a Pool of Already Created Threads Ready to do Our Job.**
- **Java 1.5 Version Introduces Thread Pool Framework to Implement Thread Pools.**
- **Thread Pool Framework is Also Known as Executor Framework.**
- **We can Create a Thread Pool as follows**
  ExecutorService service = Executors.newFixedThreadPool(3);//Our Choice
- **We can Submit a Runnable Job by using *submit()*.**
  service.submit(job);
- **We can Shutdown*ExecutiorService* by using *shutdown()*.**
  service.shutdown();

```
importjava.util.concurrent.ExecutorService;
importjava.util.concurrent.Executors;
classPrintJob implements Runnable {
        String name;
        PrintJob(String name) {
           this.name = name;
        }
        public void run() {
        SOP(name+"....Job Started By Thread:" +Thread.currentThread().getName());
                try {
                        Thread.sleep(10000);
                }
                catch (InterruptedException e) {}
                SOP(name+"....Job Completed By Thread:" +Thread.currentThread().getName());
        }
}
class  ExecutorDemo {
        public static void main(String[] args) {
                PrintJob[] jobs = {
newPrintJob("Durga"),
newPrintJob("Ravi"),
newPrintJob("Nagendra"),
newPrintJob("Pavan"),
newPrintJob("Bhaskar"),
newPrintJob("Varma")
};
                ExecutorService service = Executors.newFixedThreadPool(3);
                for (PrintJob job : jobs) {
                        service.submit(job);
                }
                service.shutdown();
        }
}
```

Output

```
Durga....Job Started By Thread:pool-1-thread-1
Ravi....Job Started By Thread:pool-1-thread-2
Nagendra....Job Started By Thread:pool-1-thread-3
Ravi....Job Completed By Thread:pool-1-thread-2
Pavan....Job Started By Thread:pool-1-thread-2
Durga....Job Completed By Thread:pool-1-thread-1
Bhaskar....Job Started By Thread:pool-1-thread-1
Nagendra....Job Completed By Thread:pool-1-thread-3
Varma....Job Started By Thread:pool-1-thread-3
Pavan....Job Completed By Thread:pool-1-thread-2
Bhaskar....Job Completed By Thread:pool-1-thread-1
Varma....Job Completed By Thread:pool-1-thread-3
```

On the Above Program 3 Threads are Responsible to Execute 6 Jobs. So that a Single Thread can be reused for Multiple Jobs.

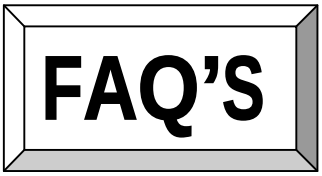
Note: Usually we can Use ThreadPool Concept to Implement Servers (Web Servers And Application Servers).

Callable and Future:

- **In the Case of Runnable Job Thread won't Return anything.**
- **If a Thread is required to Return Some Result after Execution then we should go for Callable.**
- **Callable Interface contains Only One Method *public Object call() throws Exception.***
- **If we Submit a Callable Object to Executor then the Framework Returns an Object of Type *java.util.concurrent.Future***
- **The Future Object can be Used to Retrieve the Result from Callable Job.**

```
importjava.util.concurrent.Callable;
importjava.util.concurrent.ExecutorService;
importjava.util.concurrent.Executors;
importjava.util.concurrent.Future;

classMyCallable implements Callable {
        intnum;
        MyCallable(intnum) {
                this.num = num;
        }
        public Object call() throws Exception {
                int sum = 0;
                for(int i=0; i<num; i++) {
                        sum = sum+i;
                }
                return sum;
        }
}
classCallableFutureDemo {
        public static void main(String args[]) throws Exception {
                MyCallable[] jobs = {
newMyCallable(10),
newMyCallable(20),
newMyCallable(30),
newMyCallable(40),
newMyCallable(50),
newMyCallable(60)
                                        };
                ExecutorService service = Executors.newFixedThreadPool(3);
                for(MyCallable job : jobs) {
                        Future f = service.submit(job);
                        System.out.println(f.get());
                }
                service.shutdown();
        }
}
```

```
45
190
435
780
1225
1770
```



1) **What Is Multi Tasking?**

2) **What Is Multi ThreadingAnd Explain Its Application Areas?**

3) **What Is Advantage Of Multi Threading?**

4) **When Compared With C++ What Is The Advantage In Java With Respect To Multi Threading?**

5) **In How Many Ways We Can Define A Thread?**

6) **Among Extending Thread And Implementing Runnable Which Approach Is Recommended?**

7) **Difference Between t.start() And t.run()?**

8) **Explain About Thread Scheduler?**

9) **If We Are Not Overriding run() What Will Happen?**

10) **Is It Possible Overloading Of run()?**

11) **Is It Possible To Override a start() And What Will Happen?**

12) **Explain Life Cycle Of A Thread?**

13) **What Is The Importance Of Thread Class start()?**

14) **After Starting A Thread If We Try To Restart The Same Thread Once Again What Will Happen?**

15) **Explain Thread Class Constructors?**

16) **How To Get And Set Name Of A Thread?**

17) **Who Uses Thread Priorities?**

18) **Default Priority For Main Thread?**

19) **Once We Create A New Thread What Is Its Priority?**

20) How To Get Priority From Thread And Set Priority To A Thread?

21) If We Are Trying To Set Priority Of Thread As 100, What Will Happen?

22) If 2 Threads Having Different Priority Then Which Thread Will Get Chance First For Execution?

23) If 2 Threads Having Same Priority Then Which Thread Will Get Chance First For Execution?

24) How We Can Prevent Thread From Execution?

25) What Is yield() And Explain Its Purpose?

26) Is Join Is Overloaded?

27) Purpose Of sleep()?

28) What Is synchronized Key Word? Explain Its Advantages And Disadvantages?

29) What Is Object Lock And When It Is Required?

30) What Is A Class Level Lock When It Is Required?

31) While A Thread Executing Any Synchronized Method On The Given Object Is It Possible To Execute Remaining Synchronized Methods On The Same Object Simultaneously By Other Thread?

32) Difference Between Synchronized Method And Static Synchronized Method?

33) Advantages Of Synchronized Block Over Synchronized Method?

34) What Is Synchronized Statement?

35) How 2 Threads Will Communicate With Each Other?

36) wait(), notify(), notifyAll() Are Available In Which Class?

37) Why wait(), notify(), notifyAll() Methods Are Defined In Object Instead Of Thread Class?

38) Without Having The Lock Is It Possible To Call wait()?

39) If A Waiting Thread Gets Notification Then It Will Enter Into Which State?

40) In Which Methods Thread Can Release Lock?

41) Explain wait(), notify() and notifyAll()?

42) Difference Between notify() and notifyAll()?

43) Once A Thread Gives Notification Then Which Waiting Thread Will Get A Chance?

44) How A Thread Can Interrupt Another Thread?

45) What Is Deadlock? Is It Possible To Resolve Deadlock Situation?

46) Which Keyword Causes Deadlock Situation?

47) How We Can Stop A Thread Explicitly?

48) Explain About suspend() And resume()?

49) What Is Starvation And Explain Difference Between Deadlock and Starvation?

50) What Is Race Condition?

51) What Is Daemon Thread? Give An Example Purpose Of Daemon Thread?

52) How We Can Check Daemon Nature Of A Thread? Is It Possible To Change Daemon Nature Of A Thread? Is Main Thread Daemon OR Non-Daemon?

53) Explain About ThreadGroup?

54) What Is ThreadLocal?