

Multithreading

Topics Covered Under this Chapter:

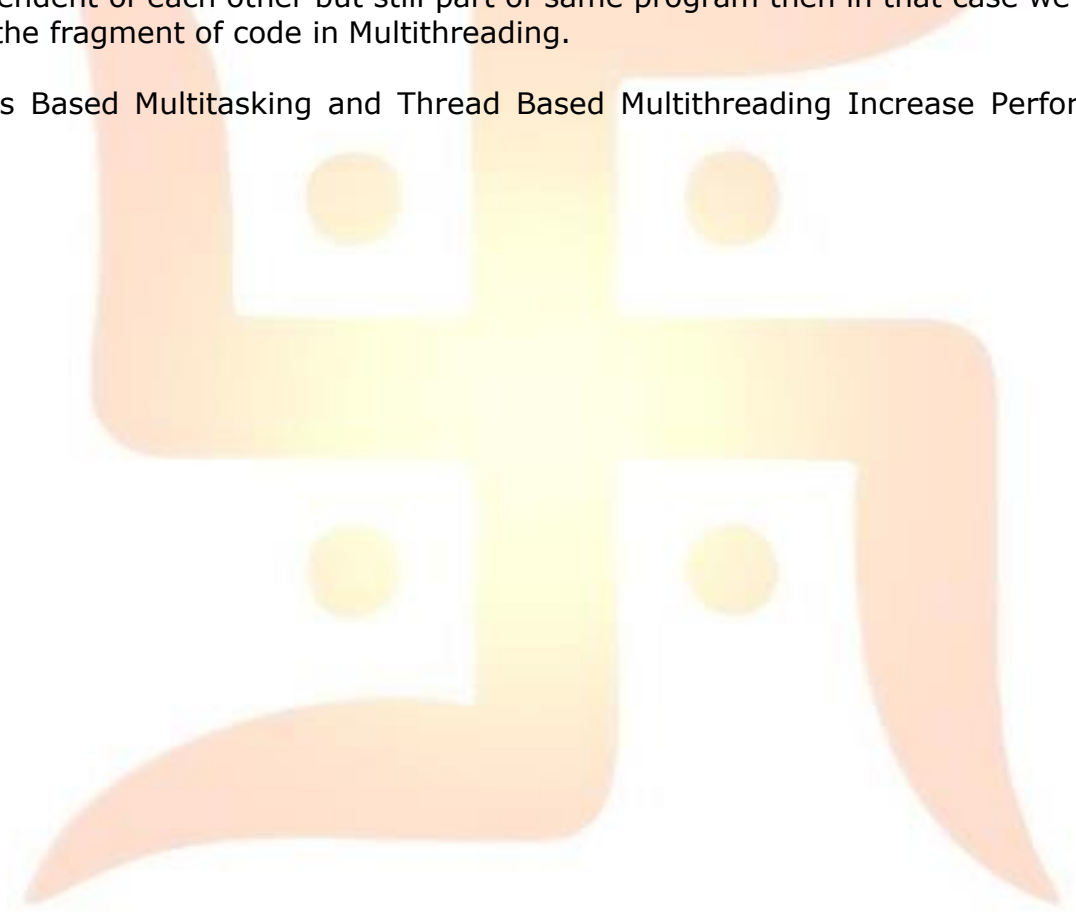
- Introduction
- Thread Life Cycle
- Creating Threads
- Naming and Prioritizing Threads
- Thread Operations
- Thread Interruption
- Synchronization
- Thread Communication
- Deadlock
- Thread Types
- ThreadGroup
- Lock Concept
- Thread Pools
- Thread Local

1.. Introduction

Multitasking means executing more than one task i.e., multiple tasks at the same time. There are two types of Multitasking we have:

- **Process Based Multitasking:** When we execute several tasks parallelly and each task is a separate independent process, this type of multitasking is called Process Based Multitasking. Example – While writing a Java code on IDE, we can also listen Music on Mp3 player and also, we can download a file from internet. All these three tasks are independent of each other and this is an example of Process Based Multitasking. Process based multitasking is suitable or occurs at Operating System (OS) Level.
- **Thread Based Multitasking:** When we execute several tasks parallelly and each task is the separate and independent task of same program, this type of multitasking is called Thread Based Multitasking or Multithreading. Here each task represents one Thread. Multithreading or Thread based multitasking is best suitable at Programming level. Example– We have a code of 1000 Lines, but first 500 lines and last 500 lines completely independent of each other but still part of same program then in that case we can execute both the fragment of code in Multithreading.

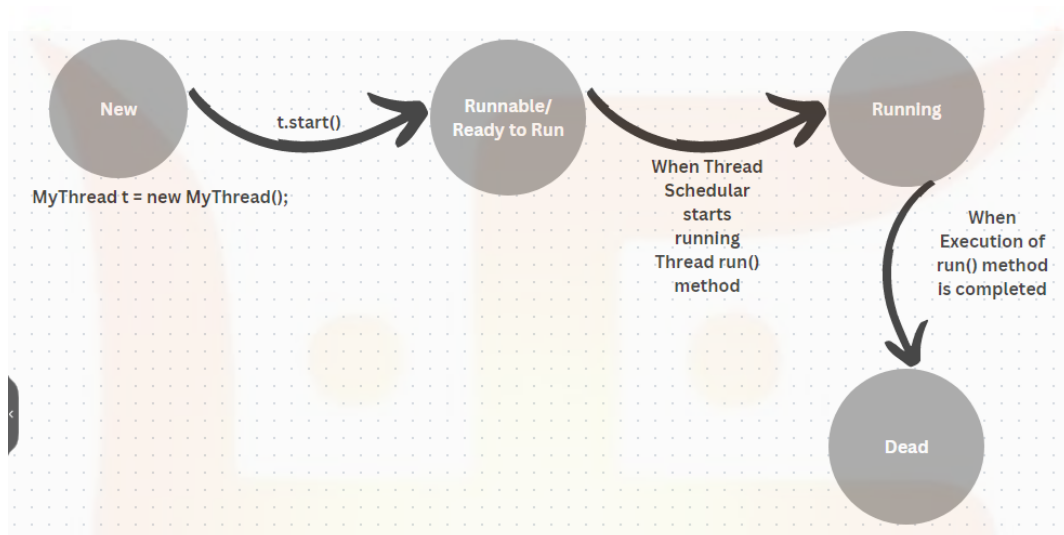
Both Process Based Multitasking and Thread Based Multithreading Increase Performance of a Program.



2.. Thread Life Cycle

Generally, In Thread life cycle, we have below primary stages:

1. **New**: When we create the Object of a Thread Class or any of its subclass, we create a new Thread.
2. **Runnable**: When start() method is invoked by the reference of Thread class or its subclass, then thread enters into Runnable state.
3. **Running**: When Thread scheduler picks up the thread to start its execution i.e., start executing run() method, then that state of thread is a Running State.
4. **Dead**: When the execution of run() gets completed, then that state of thread is called Dead State.



But apart from these four primary stages of Threads, there can be additional Thread Stages too which will occur in case of Thread Interruption mainly. We will discuss them in Thread Interruption.

Note:

When a thread completed its execution of **run()** method if we try to call the start() method again on that thread then we will get **IllegalThreadStateException**.

Main.java	Output
<pre>1 class MyThread implements Runnable 2 { 3 public void run() 4 { 5 System.out.println("Child Thread"); 6 } 7 } 8 class ThreadDemo { 9 public static void main(String[] args) 10 { 11 MyThread t = new MyThread(); 12 Thread t1 = new Thread(t); 13 t1.start(); 14 System.out.println("After Child Thread Ends Execution"); 15 t1.start(); 16 } 17 }</pre>	<pre>java -cp /tmp/NKthIi7Ndc ThreadDemo After Child Thread Ends Execution Child Thread Exception in thread "main" java.lang.IllegalThreadStateException at java.base/java.lang.Thread.start(Thread.java:789) at ThreadDemo.main(ThreadDemo.java:15)</pre>

3.. Creating Threads

There are three ways in which we can define or create a Thread in Java:

1. By Extending Thread Class
2. By Implementing Runnable Interface
3. By Implementing Callable Interface

Out of these three methods, creating a Thread by implementing Runnable/Callable interface depending on the situation is recommended as if we create a Thread by extending Thread class then we cannot inherit from any other class, as java doesn't support Multiple Inheritance. But by implementing Runnable/Callable interface we can also inherit any other class as well. Also, when our class inherits Thread class, internally Thread class implements Runnable Interface.

3.1 Thread Class

- Create a Class MyThread which will extend Thread class present in java.lang package.
- In the created class, we have to define run() method which is of void type which will contain a fragment of code which will run as a parallel task.
- Create one more class ThreadDemo class which contains the main method, we have to create an object of MyThread class created in step 1.
- Call the start() method with the reference of MyThread type.

```
1  class MyThread extends Thread
2  {
3      public void run()
4      {
5          for(int i=1; i <= 20; i++)
6          {
7              System.out.println("Thread 1 : " + i);
8          }
9      }
10 }
11 class ThreadDemo
12 {
13     public static void main(String[] args)
14     {
15         MyThread t = new MyThread();
16         t.start();
17         for(int i=1; i <= 20; i++)
18         {
19             System.out.println("Main Thread : " + i);
20         }
21     }
22 }
```

Constructors of Thread Class

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);`

3.2 Runnable Interface

- Create a Class MyThread which will implement Runnable Interface present in java.lang package. Runnable interface contains only one method which is a run() method which we need to override in our class which will contain a fragment of code which will run as a parallel task.

- Create one more class ThreadDemo class which contains the main method, in this method we need to create an object of MyThread class first and then create the object of Thread class and pass the reference of MyThread class in argument of constructor.
- Call the start() method with reference of Thread class.

```

1  class MyThread implements Runnable
2  {
3      public void run()
4      {
5          for(int i=1; i <= 20; i++)
6          {
7              System.out.println("Thread 1 : " + i);
8          }
9      }
10 }
11 class ThreadDemo
12 {
13     public static void main(String[] args)
14     {
15         MyThread t1 = new MyThread();
16         Thread t = new Thread(t1);
17         t.start();
18         for(int i=1; i <= 20; i++)
19         {
20             System.out.println("Main Thread : " + i);
21         }
22     }
23 }

```

Note:

If we just write

```
Thread t = new Thread();
```

Then in that case t.start() will call the start() which will internally call the run() of Thread class which has no implementation.

To make Thread class to call the run() defined in our class which implements Runnable interface, we have to define

```
Thread t = new Thread(t2);
```

3.3 Callable Interface

When we create a Thread by implementing **Runnable** Interface which contains only one method which is **run()** of return type **void**. So **run()** method or **Runnable** Interface won't return anything to the caller.

While there is some case where we want our child thread to return something to the caller, in that case we can create a thread by implementing **Callable** Interface. Callable interface also contains only one method **call()**. But **call()** method has a return type of **Object** Class i.e., **call()** method can return anything. **Callable** interface was introduced in Java 1.5 version.

```
public Object call() throws Exception;
```

A run() method of Runnable Interface doesn't throw any exception, if we have to handle any exception then we have to write that portion of code within try and catch block but call() method of Callable Interface throws Exception.

To Hold the return value of call() method of Callable Interface we need a **Future** object. We can call get() method of Future Object to get the actual return value.

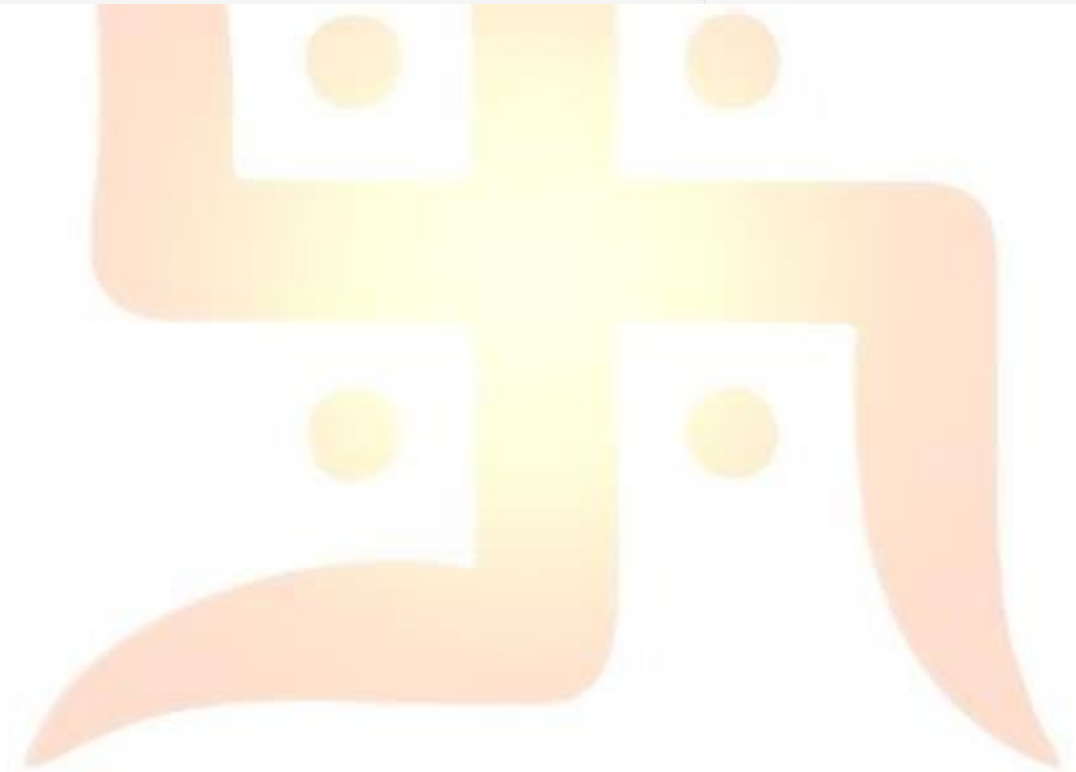
Main.java

Run

```
1- import java.util.concurrent.*;
2- class MyCallable implements Callable{
3     int num;
4     MyCallable(int num){
5         this.num = num;
6     }
7
8     public Object call() throws Exception{
9         System.out.println(Thread.currentThread().getName()+"..Is responsible t find sum of First "+
10             num + " numbers");
11         int sum = 0;
12         for(int i=1; i<=num; i++){
13             sum+=i;
14         }
15         return sum;
16     }
17 }
18 class CallableDemo {
19     public static void main(String[] args) throws Exception {
20         MyCallable[] callableJobs = {
21             new MyCallable(10),
22             new MyCallable(20),
23             new MyCallable(30),
24             new MyCallable(40),
25             new MyCallable(50),
26             new MyCallable(60)
27         };
28         ExecutorService service = Executors.newFixedThreadPool(3);
29         for(MyCallable job:callableJobs){
30             Future f = service.submit(job);
31             System.out.println(f.get());
32         }
33         service.shutdown();
34     }
35 }
```

Output

```
java -cp /tmp/U7yc5L5tG1 CallableDemo
pool-1-thread-1..Is responsible t find sum of First 10 numbers
55
pool-1-thread-2..Is responsible t find sum of First 20 numbers
210
pool-1-thread-3..Is responsible t find sum of First 30 numbers
465
pool-1-thread-1..Is responsible t find sum of First 40 numbers
820
pool-1-thread-2..Is responsible t find sum of First 50 numbers
1275
pool-1-thread-1..Is responsible t find sum of First 60 numbers
1830
```



4.. Naming and Prioritizing Threads

We can set the name and Priority of any threads in Java if we don't want to have the threads default name and priority set by the JVM.

4.1 Naming Thread

- If we as a program doesn't specify the name of a Thread then by default JVM names the Thread as Thread-0, Thread-1 and so on.
- We have below method signature for **getName()** and **setName()** methods.

public final String getName();

public final void setName(String name);

- We can also get and set the name of a Current Thread as well as below:

Thread.currentThread().getName();

Thread.currentThread().setName(String name);

```
1 class MyThread implements Runnable
2 {
3     public void run()
4     {
5         System.out.println("Child Thread");
6     }
7 }
8 class ThreadDemo {
9     public static void main(String[] args)
10    {
11        System.out.println(Thread.currentThread().getName());
12        Thread.currentThread().setName("ChangedCurrentThread");
13        System.out.println("\n"+Thread.currentThread().getName()+"\n");
14        MyThread t = new MyThread();
15        Thread t1 = new Thread(t);
16        System.out.println(t1.getName());
17        t1.setName("Vikash");
18        System.out.println(t1.getName());
19    }
20 }
```

Output

```
java -cp /tmp/NKthIi7Ndc ThreadDemo
main
ChangedCurrentThread
|
Thread-0
Vikash
```


4.2 Prioritizing Threads

- Every thread in Java has some priority whether it is provided by JVM as a default priority or explicitly set by programmer. The priority of a thread lies in the range of 1-10, where 1 is the Minimum Priority or Low Priority and 10 is the maximum priority and Highest Priority.
Thread.MIN_PRIORITY : 1
Thread.NORM_PRIORITY : 5
Thread.MAX_PRIORITY : 10

```
1 class MyThread implements Runnable
2 {
3     public void run()
4     {
5         System.out.println("Child Thread");
6     }
7 }
8 class ThreadDemo {
9     public static void main(String[] args)
10    {
11        System.out.println(Thread.MIN_PRIORITY+"\n");
12        System.out.println(Thread.NORM_PRIORITY+"\n");
13        System.out.println(Thread.MAX_PRIORITY+"\n");
14    }
15 }
```

```
java -cp /tmp/NKthIi7Ndc ThreadDemo
1
5
10
```

- Thread Scheduler will use this priority and based on the priority of the Thread, Thread Scheduler will decide, the Thread with Highest Priority will be the first which will get executed and then the Thread with lower Priority will get executed.
- If two Threads are having the same priority then its Thread scheduler will decide whom to execute first, based on internal Algorithm of Round Robin or FIFO, or any other algorithm, Thread scheduler can execute any of the Two Threads having the same priority.

We have below methods for getting and setting thread priorities.

public final int getPriority();

public final void setPriority(int p);

```
Main.java  Run  Output  Clear
1 class MyThread implements Runnable
2 {
3     public void run()
4     {
5         System.out.println("Child Thread");
6     }
7 }
8 class ThreadDemo {
9     public static void main(String[] args)
10    {
11        MyThread t1 = new MyThread();
12        Thread t = new Thread(t1);
13        System.out.println("Default Priority : " + t.getPriority() + "\n");
14        t.setPriority(7);
15        System.out.println("In Range Priority : " + t.getPriority() + "\n");
16        t.setPriority(17);
17        System.out.println("Out Of Range Priority : " + t.getPriority() + "\n");
18    }
19 }
```


```
java -cp /tmp/NKthIi7Ndc ThreadDemo
Default Priority : 5

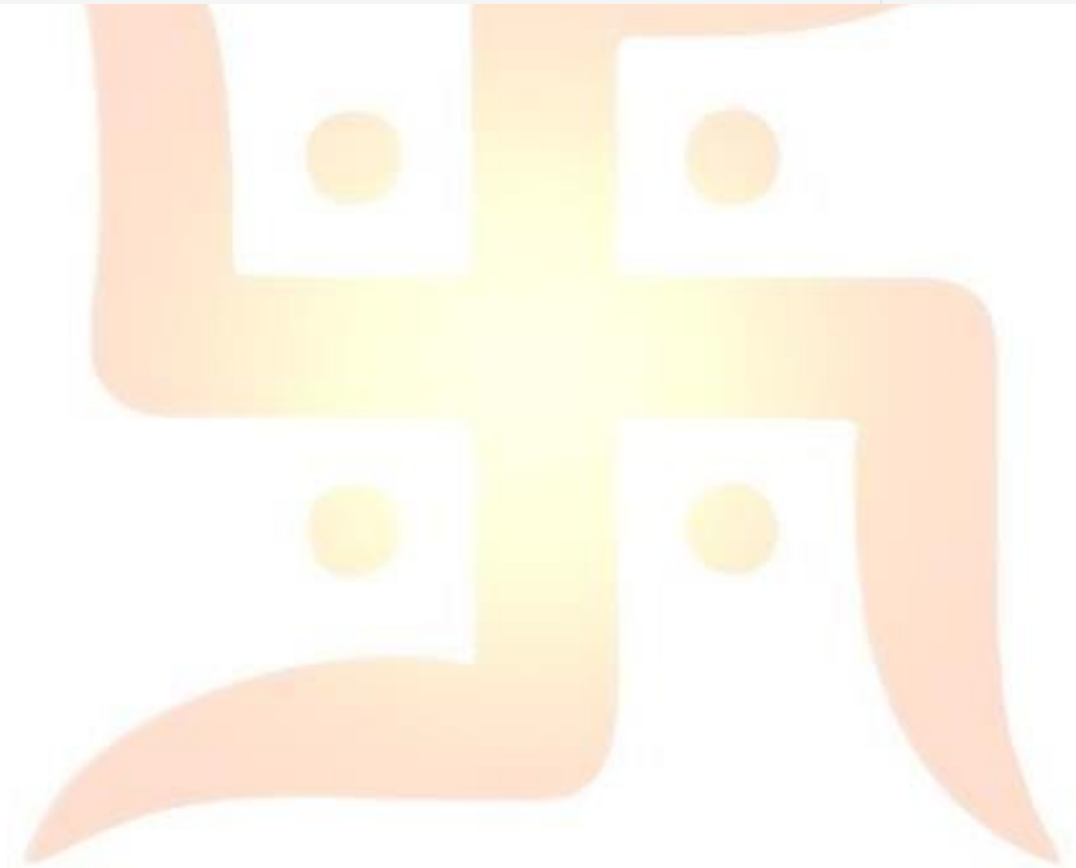
In Range Priority : 7

Exception in thread "main" java.lang.IllegalArgumentException: at java.base/java.lang.Thread.setPriority(Thread
.java:1136)
    at ThreadDemo.main(ThreadDemo.java:16)
```

In this example we can see, when we are trying to get the default priority of Thread defined by JVM, it returns 5. Then we are setting the thread priority within the valid Range 1-10, and then when we print the priority of Thread, it returns the priority which we have set, here 7 in this example. When we are trying to set the thread priority beyond the Range 1-10 i.e., let's say 17 then in this case we are going to get Run Time Exception which is **IllegalArgumentException**.

Note: Default Priority of a Main Thread is 5 but for all other Threads the default priority will be inherited from parent to child. If the Priority of Parent Thread is 10, then child thread priority will be 10.

Main.java	Run	Output
<pre>1 class MyThread implements Runnable 2 { 3 public void run() 4 { 5 System.out.println("Child Thread"); 6 } 7 } 8 class ThreadDemo { 9 public static void main(String[] args) 10 { 11 System.out.println("Default Priority of Main Thread : " + Thread.currentThread().getPriority() 12 + "\n"); 13 Thread.currentThread().setPriority(2); 14 System.out.println("Changed Priority of Main Thread : " + Thread.currentThread().getPriority() 15 + "\n"); 16 MyThread t1 = new MyThread(); 17 Thread t = new Thread(t1); 18 System.out.println("Child Thread Priority : " + t.getPriority() + "\n"); 19 t.setPriority(7); 20 System.out.println("Child Thread Priority Changed : " + t.getPriority() + "\n"); 21 } 22 }</pre>		<pre>java -cp /tmp/NKthIi7Ndc ThreadDemo Default Priority of Main Thread : 5 Changed Priority of Main Thread : 2 Child Thread Priority : 2 Child Thread Priority Changed : 7 Default Priority of Main Thread : 2</pre>



5.. Thread Operations

We can perform below Operations on Thread.

- Starting a Thread
- Stopping a Thread
- Suspending/Resuming a Thread

5.1 Starting a Thread

start() method is a method present in **Thread** class of java.lang package. When we invoke start() method like the below code snippet:

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

Here start() method of Thread class performs a set of instructions before calling run() method. Thread class also has run() method which has empty definition. Jobs done by start() method before calling run() are as follows:

- Register Thread to Thread Scheduler
- Perform any other Mandatory activities
- Invoke run() method.

Hence, start() method is the one which will create a thread by performing all mandatory tasks for thread creation and once all mandatory tasks are completed, it will go on calling run() method.

Now we have couple of cases with respect to overriding of start() and run():

- **t1.start():** This method will create an actual thread by performing actual activity of Thread creation before calling run() method. We cannot override the start() method in a class which is extending Thread class because in that case our start() will be called as child class method gets the priority and in this case also no new thread is created and start() method will be executed as a normal method and no multithreading is achieved. But in this manner we can use super.start() method in start() method which we define which will call start() of Thread class and this can be the workaround of overriding start().
- **t1.run():** We cannot call directly t1.run() if we do so, it will be trigger as a normal method and no new thread is created. Thread class also has run() method which has empty definition that's the reason we override the run() method in the class which extends Thread class or implements Runnable interface.

So, it is highly recommended to override the run() method in a class extending Thread class or implementing Runnable interface and it is also highly recommended not to override the start() method of Thread class at any cost.

```

class MyThread extends Thread
{
    public void run()
    {
        //We should override run() method of Thread Class which is having empty definition
        //To write a piece of code which will run in parallel
    }

    public void start()
    {
        //We should not override start() method at any cost.
        super.start(); // Workaround if we override start() method
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t1 = new MyThread();
        t1.start(); // Creates a actual Thread
        t1.run();   // executes run() as a normal method
    }
}

```

Now we have couple of cases with respect to overloading run():

We can overload run() method in our class but JVM will consider only that run() method which has no arguments. The method with arguments will be treated and executed as a normal method.

```

class MyThread extends Thread
{
    public void run()
    {
        //Thread execution Code
    }

    public void run(int i)
    {
        //It will be executed as a normal method
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t1 = new MyThread();
        t1.start(); //JVM will call run() not run(int i) internally to create Thread
    }
}

```

Consider a Below Program:

```
1 class MyThread implements Runnable
2 {
3     public void run()
4     {
5         System.out.println("Child Thread");
6     }
7 }
8 class ThreadDemo {
9     public static void main(String[] args)
10    {
11        MyThread t = new MyThread();
12        Thread t1 = new Thread();
13        Thread t2 = new Thread(t);
14
15        t1.start();
16        t1.run();
17        t2.start();
18        t2.run();
19        t.start();
20        t.run();
21    }
22 }
```

Here:

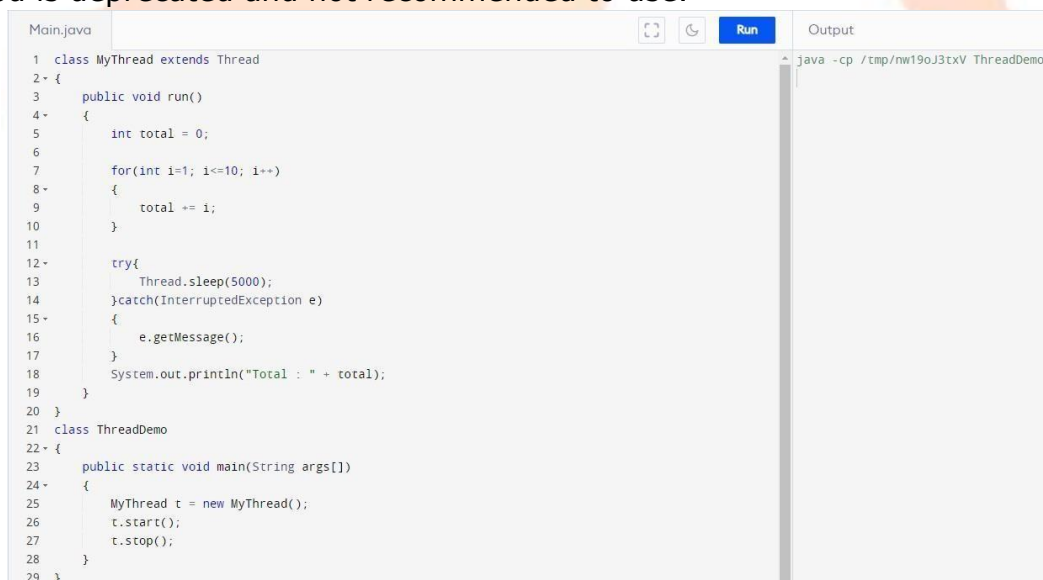
1. **t1.start()** : Creates a Thread but here start() method of Thread class will be called and which will call internally run() method Thread Class which is of empty implementation.
2. **t1.run()** : Calls run() method of Thread class as a Normal method.
3. **t2.start()** : Creates a Thread and call our run() method which we have define in MyThread Class.
4. **t2.run()** : Calls run() method of MyThread class as a Normal method.
5. **t.start()** : As there is no start() method in MyThread Class, it will give Compile time Error No Symbol found.
6. **t.run()** : Calls run() method of MyThread class as a Normal method.

5.2 Stopping a Thread

We can stop a thread in middle of execution by calling stop() method.

```
Thread t = new Thread();
t.stop();
```

stop() method will move the thread from running state to dead state in Thread life cycle. But stop() method is deprecated and not recommended to use.



```
Main.java
1 class MyThread extends Thread
2 {
3     public void run()
4     {
5         int total = 0;
6
7         for(int i=1; i<=10; i++)
8         {
9             total += i;
10        }
11
12        try{
13            Thread.sleep(5000);
14        }catch(InterruptedException e)
15        {
16            e.getMessage();
17        }
18        System.out.println("Total : " + total);
19    }
20 }
21 class ThreadDemo
22 {
23     public static void main(String args[])
24     {
25         MyThread t = new MyThread();
26         t.start();
27         t.stop();
28     }
29 }
```

Output

```
java -cp /tmp/nw19oJ3txV ThreadDemo
```

5.3 Suspending and Resuming a Thread

A thread can suspend any thread by calling `suspend()` method then immediately thread will be entered into suspended state. Thread can resume suspended thread by calling `resume()` method. These are also deprecated methods.

```
public void suspend();
```

```
public void resume();
```



6.. Thread Interruptions

We can prevent the execution of any Thread by using any of the below methods.

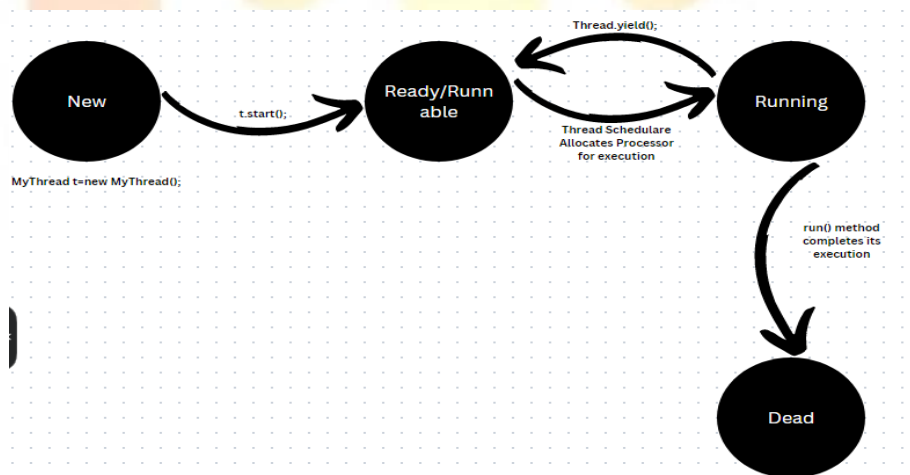
yield()
join()
sleep()
interrupt()

6.1 yield()

yield() method causes to pause the current executing thread in order to give chance to other waiting thread in the queue of same priority. If there is not any waiting thread or all waiting threads are of low priority then same thread will be continue to execute. If multiple waiting Threads say T1,T2,T3 and T4 are of same priority then which thread will get a chance to execute first is completely dependent on Thread Scheduler and we cannot predict.

The Thread which is yielded say TX which is also of same priority as T1, T2, T3 and T4 gets placed in the waiting queue along with other waiting Threads. Now, since priority of all Threads are same, we cannot predict when will this TX get the chance of execution again. It's totally dependent on Thread Scheduler.

Prototype of yield() method is : *public static native void yield();*



Main.java	Output
<pre>1 class MyThread implements Runnable 2 { 3 public void run() 4 { 5 for(int i=1; i <= 10; i++) 6 { 7 System.out.println("Child Thread : " + i); 8 Thread.yield(); 9 } 10 } 11 } 12 class ThreadDemo { 13 public static void main(String[] args) 14 { 15 MyThread t1 = new MyThread(); 16 Thread t = new Thread(t1); 17 t.start(); 18 for(int i=1; i <= 10; i++) 19 { 20 System.out.println("Main Thread : " + i); 21 } 22 } 23 }</pre>	<pre>java -cp /tmp/NKthIi7Ndc ThreadDemo Main Thread : 1 Main Thread : 2 Main Thread : 3 Main Thread : 4 Main Thread : 5 Main Thread : 6 Main Thread : 7 Main Thread : 8 Main Thread : 9 Main Thread : 10 Child Thread : 1Child Thread : 2 Child Thread : 3Child Thread : 4 Child Thread : 5 Child Thread : 6Child Thread : 7Child Thread : 8 Child Thread : 9 Child Thread : 10</pre>

Here we have only 2 Threads, Main Thread and Child Thread and Child Thread is calling Thread.yield() method so it will always leaves the processor after executing One Statement. So here chance of completing Main Thread execution first is higher then the child Thread.

6.2 join()

If a Thread say t1 wants to wait for the completion of any other thread say t2 in order to use the output of t2 then in that case the Thread t1 who wants to wait will call the method join() as t2.join(); And when the Thread t1 calls the method t2.join(); method then thread t1 enters into a Waiting state until t2 completes. Once t2 completes then only t1 can continue its execution.

Simple Example to understand the concept:

Consider a 3 Wedding Management activities i.e., Venue Selection, Wedding Card Printing, and Wedding card Distribution and these three activities are executed by three threads say t1, t2 and t3. Now all three threads are running in parallel, but Thread t2 has to wait until t1 work gets completed i.e., venue gets fixed so thread t2 will call t1.join() method and thread t3 has to wait until thread t2 completed its execution i.e., wedding cards gets printed then only it can be distributed so thread t3 will call t2.join() method.

Prototype of join() method is :

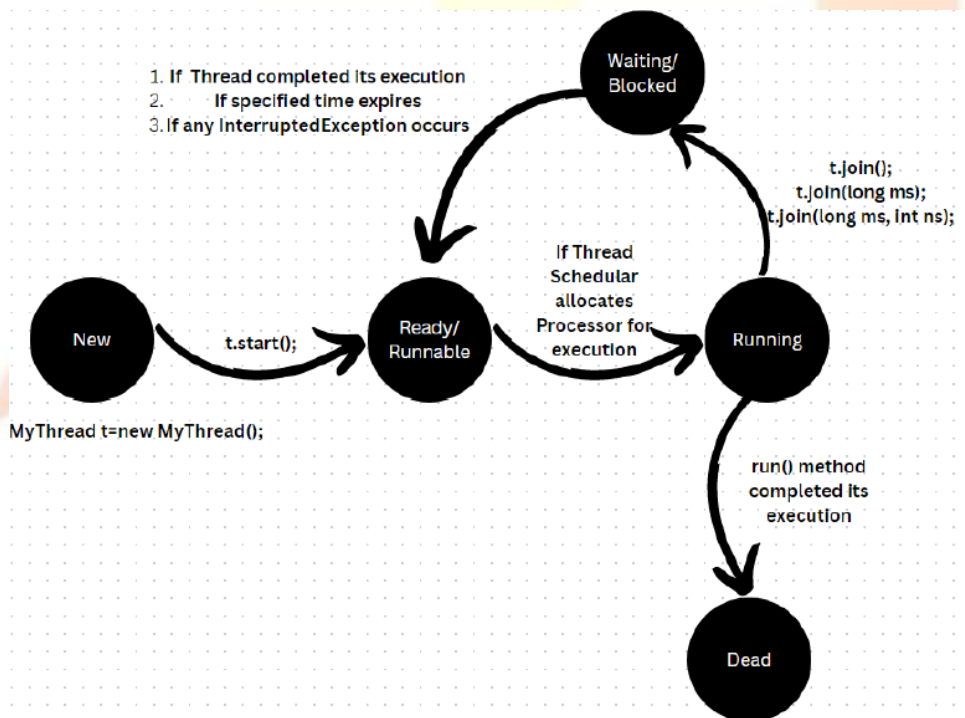
```
public final void join();
```

```
public final void join(long timeToWait); timeToWait is time in Milliseconds
```

```
public final void join(long milliseconds, int nanoseconds); // Strict time wait
```

Every join() methods throws **InterruptedException** because while waiting for other Thread to completed its execution there might be a chance that any other thread interrupt in between. So its mandatory to handle InterruptedException if we use join() method if we don't do it then we will face compile time error.

Below is the Thread Life cycle when join() method gets called in between by any of the thread.



Use Case 1: Main Thread waits for completion of Child Thread

```
Main.java
1 class MyThread implements Runnable
2 {
3     public void run()
4     {
5         for(int i=1; i <= 10; i++)
6         {
7             System.out.println("Child Thread : " + i);
8             try{
9                 Thread.sleep(2000);
10            }catch(InterruptedException e){
11                e.getMessage();
12            }
13        }
14    }
15 }
16 class ThreadDemo {
17     public static void main(String[] args) throws InterruptedException
18     {
19         MyThread t1 = new MyThread();
20         Thread t = new Thread(t1);
21         t.start();
22         t.join();
23         for(int i=1; i <= 10; i++)
24         {
25             System.out.println("Main Thread : " + i);
26         }
27     }
28 }
```

```
Output
java -cp /tmp/NKthI17Ndc ThreadDemo
Child Thread : 1
Child Thread : 2
Child Thread : 3
Child Thread : 4
Child Thread : 5
Child Thread : 6
Child Thread : 7
Child Thread : 8
Child Thread : 9
Child Thread : 10
Main Thread : 1
Main Thread : 2
Main Thread : 3
Main Thread : 4
Main Thread : 5
Main Thread : 6
Main Thread : 7
Main Thread : 8
Main Thread : 9
Main Thread : 10
```

Use Case 2: Child Thread waits for completion of Main Thread

```
Main.java
1 class MyThread implements Runnable
2 {
3     static Thread mt;
4     public void run()
5     {
6         try{
7             mt.join();
8         }catch(InterruptedException e){
9             e.getMessage();
10        }
11        for(int i=1; i <= 10; i++)
12        {
13            System.out.println("Child Thread : " + i);
14        }
15    }
16 }
17 class ThreadDemo {
18     public static void main(String[] args) throws InterruptedException
19     {
20         MyThread.mt=Thread.currentThread();
21         MyThread t1 = new MyThread();
22         Thread t = new Thread(t1);
23         t.start();
24         for(int i=1; i <= 10; i++)
25         {
26             System.out.println("Main Thread : " + i);
27             Thread.sleep(2000);
28         }
29     }
30 }
```

```
Output
java -cp /tmp/NKthI17Ndc ThreadDemo
Main Thread : 1
Main Thread : 2
Main Thread : 3
Main Thread : 4
Main Thread : 5
Main Thread : 6
Main Thread : 7
Main Thread : 8
Main Thread : 9
Main Thread : 10
Child Thread : 1
Child Thread : 2
Child Thread : 3
Child Thread : 4
Child Thread : 5
Child Thread : 6
Child Thread : 7
Child Thread : 8
Child Thread : 9
Child Thread : 10
```

Use Case 3: Both Main Thread and Child Thread Waits for Completion of Each Other

If Main Thread waits for completion of Child Thread and Child Thread waits for completion of the Main Thread both at the same time, then this kind of situation is called a **deadlock** situation where both the threads are waiting for each other to complete. Here in this case, there won't be any output of the program as both the threads are waiting for each other to complete and program will run forever.

```
Main.java
1 class MyThread implements Runnable
2 {
3     static Thread mt;
4     public void run()
5     {
6         try{
7             mt.join();
8         }catch(InterruptedException e){
9             e.getMessage();
10        }
11        for(int i=1 ; i <= 10; i++)
12        {
13            System.out.println("Child Thread : " + i);
14        }
15    }
16 }
17 class ThreadDemo {
18     public static void main(String[] args) throws InterruptedException
19     {
20         MyThread mt=Thread.currentThread();
21         MyThread t1 = new MyThread();
22         Thread t = new Thread(t1);
23         t.start();
24         t.join();
25         for(int i=1; i <= 10; i++)
26         {
27             System.out.println("Main Thread : " + i);
28             Thread.sleep(2000);
29         }
30     }
31 }
```

Output
java -cp /tmp/NKth117Ndc ThreadDemo

Use Case 4: Any Thread Main/Child calls a join() method on itself

Consider the below code snippet where any thread say Main Thread is calling a join() method on same thread i.e., Main Thread Itself. This will also lead to a **deadlock** situation. For this also no output will be generated and program will run forever.

```
17 class ThreadDemo {
18     public static void main(String[] args) throws InterruptedException
19     {
20         Thread.currentThread().join();
21     }
22 }
```

6.3 sleep()

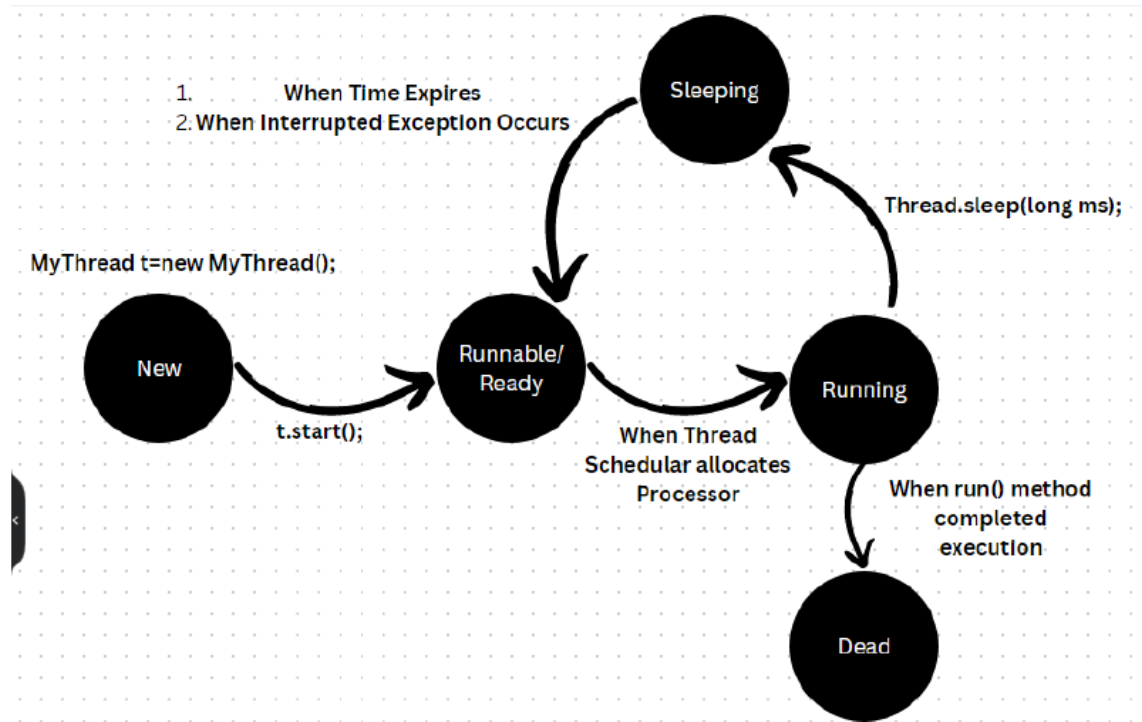
When thread wants to pause its operation or execution for a specific period of time then we should go for sleep() method.

Prototype of sleep() method is :

public static native void sleep(long milliseconds);

public static void sleep(long milliseconds, int nanoseconds);

Every sleep() method throws **InterruptedException** as while thread is sleeping some other Thread might come and interrupt the sleep of sleeping thread. Every running thread go into sleeping state when sleep() method is called and will be in the sleeping state when the time expires or any Interrupted exception occurs in between.



Lets understand sleep() method with code snippet:

Main.java	Output
<pre> 1 class SlideTimer 2 { 3 public static void main(String[] args) throws InterruptedException 4 { 5 for(int i=1; i <= 10; i++) 6 { 7 System.out.println("Slide : " +i); 8 Thread.sleep(5000); 9 } 10 } 11 } </pre>	<pre> java -cp /tmp/x8QKJ1HEqR SlideTimer Slide : 1Slide : 2 Slide : 3 Slide : 4 Slide : 5 Slide : 6 Slide : 7 Slide : 8 Slide : 9Slide : 10 </pre>


6.4 interrupt()

A thread can interrupt a sleeping or waiting thread by calling interrupt() method. A sleeping thread is a thread which is sleeping by sleep() method and waiting thread is a thread which is in waiting state by call of join() method.


Prototype of Interrupt method:
public void interrupt();

Interrupt() method will keep on waiting for the target thread to go on sleep state or waiting state and once thread enters into this state, it will throw the InterruptedException. If target thread is already in sleeping or waiting state then immediately it will throw InterruptedException. If target thread never enters into sleeping or target thread for the entire life time then in that case interrupt() call will be wasted.

Code without interruption:

Main.java	Run	Output
<pre>1 class MyThread implements Runnable 2 { 3 public void run() 4 { 5 try 6 { 7 for(int i=1; i <= 10; i++) 8 { 9 System.out.println("Child Thread : " + i); 10 Thread.sleep(2000); 11 } 12 } 13 catch(InterruptedException e) 14 { 15 System.out.println("OOPS !! I Got Interrupted"); 16 } 17 } 18 } 19 } 20 class SlideTimer 21 { 22 public static void main(String[] args) throws InterruptedException 23 { 24 MyThread myThread = new MyThread(); 25 Thread t=new Thread(myThread); 26 t.start(); 27 System.out.println("End of Main Thread"); 28 } 29 } 30 }</pre>		<pre>java -cp /tmp/x8QKJ1HEqR SlideTimer End of Main Thread Child Thread : 1 Child Thread : 2Child Thread : 3 Child Thread : 4 Child Thread : 5 Child Thread : 6 Child Thread : 7 Child Thread : 8 Child Thread : 9 Child Thread : 10</pre>

Code With Interruption:

Main.java	Run	Output
<pre>1 class MyThread implements Runnable 2 { 3 public void run() 4 { 5 try 6 { 7 for(int i=1; i <= 10; i++) 8 { 9 System.out.println("Child Thread : " + i); 10 Thread.sleep(2000); 11 } 12 } 13 catch(InterruptedException e) 14 { 15 System.out.println("OOPS !! I Got Interrupted"); 16 } 17 } 18 } 19 } 20 class SlideTimer 21 { 22 public static void main(String[] args) throws InterruptedException 23 { 24 MyThread myThread = new MyThread(); 25 Thread t=new Thread(myThread); 26 t.start(); 27 t.interrupt(); 28 System.out.println("End of Main Thread"); 29 } 30 }</pre>		<pre>java -cp /tmp/x8QKJ1HEqR SlideTimer End of Main Thread Child Thread : 1 OOPS !! I Got Interrupted</pre>

7.. Synchronization

If multiple threads say T1, T2, T3 etc. is executing an operation simultaneously on same Object then this can lead to data inconsistency issue. This is nothing but race condition. For example, consider a situation where I am booking an online bus ticket, On Online Bus Ticket Booking Application it is showing 6 vacant seats available in a bus. Suppose I want to book 4 seats and at the same time some other user also wants to book the bus ticket and he wants to book 5 seats. Since both the users are operating on the same object almost at the same time, this can lead to data inconsistency. This problem can be solved with the concept of Synchronization.

In Java, **synchronized** is a modifier which is applicable only on **methods** or **blocks**, not on variables or class. This modifier is responsible to make methods or block gets executed only by the one Thread at a time and once that thread completes its operation, it gives a chance to another Thread.

Advantages of using Synchronization:

1. We can overcome data inconsistency problem

Disadvantages of using Synchronization:

1. It increases waiting time of Threads and that reduces performance.

Hence, **synchronized** keyword should be only used if there is any specific requirement which can be solved by using synchronized keyword.

Internal Implementation of Synchronization

Internally, synchronization concept is implemented based on lock concept. Every object in Java has a unique lock. Whenever we are using synchronized method or block then only this lock concept will come into picture. Whenever any thread say T wants to execute a synchronized method on any object then first thread T has to get lock of that object. Once thread T gets the lock of an object then it will start executing that synchronized method and once the execution of synchronized method is completed, then thread T will release the lock of that object which can be used by another thread waiting in a queue. This getting lock or releasing lock will be taken care by JVM, as a programmer we don't have to worry about it.

Consider there is a class say X, which contains three methods m1(), m2() and m3(). m1() and m2() both are synchronized method while m3() is a normal method. Let's consider we have an object Y of class X. Now Thread T1 comes and wants to execute the m1() method on Y object, then it will get the lock of object Y and started execution. While execution is in progress, thread T2, T3 and T4 comes and T2 wants to execute m1(), T3 wants to execute m2() and T4 wants to execute m3() on same object Y. Since m3() is a normal method so T4 will complete its execution. But Thread T2 and T3 though they want to call two separate methods say m1() and m2() both are synchronized but they want to operate these methods on same object Y whose lock is already acquired by T1, that's the reason both T2 and T3 will go to waiting state. This proves that lock concept is based on object but not based on methods.

Now question arises – When should we go for synchronized method and when should we go for non-synchronized method?

Whenever the operation didn't change the state of object then we should go for non- synchronized method where any number of threads can execute at the same time. For example – While performing any read operation like getting product details before making any shopping. But when the operation changes the state of object then we should go for synchronized method where only

Suppose if wish() method in non-synchronized then both the threads t1 and t2 will try to operate on it at the same time, then we will see mixed output. Let's see what will be the output.

```
Main.java
1- class Display{
2-     public void wish(String name){
3-         for(int i=1; i < 10; i++){
4-             System.out.print("Good Morning : ");
5-             try{
6-                 Thread.sleep(2000);
7-             }catch(InterruptedException e){
8-                 System.out.print("Got Interrupted!!");
9-             }
10-            System.out.println(name);
11-        }
12-    }
13- }
14- class MyThread extends Thread{
15-     Display display;
16-     String name;
17-     MyThread(Display display, String name){
18-         this.display=display;
19-         this.name=name;
20-     }
21-     public void run(){
22-         display.wish(name);
23-     }
24- }
25- class SynchronizedDemo {
26-     public static void main(String[] args) {
27-         Display display = new Display();
28-         MyThread t1=new MyThread(display,"King");
29-         MyThread t2=new MyThread(display,"Queen");
30-         t1.start();
31-         t2.start();
32-     }
33- }
```

```
Output
java -cp /tmp/10on5qvrux SynchronizedDemo
Good Morning : Good Morning : KingGood Morning : Queen
Good Morning : King
Good Morning : Queen
Good Morning : King
Good Morning : QueenGood Morning : King
Good Morning : Queen
Good Morning : King
Good Morning : Queen
Good Morning : King
Good Morning : QueenGood Morning : King
Good Morning : QueenGood Morning : KingGood Morning : QueenGood Morning : King
Queen
```

Note: If multiple Threads T1, T2 are executing wish() method on two different display object say D1 and D2, then in that case we are not required to make wish() as synchronized and we are going to get irregular output.

In this case if we declare whether the method as synchronized or non-synchronized output will be the same i.e. irregular output.

```
25- class SynchronizedDemo {
26-     public static void main(String[] args) {
27-         Display display1 = new Display();
28-         Display display2 = new Display();
29-         MyThread t1=new MyThread(display1,"King");
30-         MyThread t2=new MyThread(display2,"Queen");
31-         t1.start();
32-         t2.start();
33-     }
34- }
```

In this case, When we declared a method wish() as **static synchronized** then output will be regular. This is because when we declare a method as only synchronized then thread will get the object level lock but when we declare method as static synchronized then thread will acquire a class level lock. Once thread gets class level lock then only thread can execute static synchronized method of that class. Once the execution of method is over, it will release the lock.


```
Main.java
1- class Display{
2-     public static synchronized void wish(String name){
3-         for(int i=1; i < 10; i++){
4-             System.out.print("Good Morning : ");
5-             try{
6-                 Thread.sleep(2000);
7-             }catch(InterruptedException e){
8-                 System.out.print("Got Interrupted!!");
9-             }
10-            System.out.println(name);
11-        }
12-    }
13- }
14- class MyThread extends Thread{
15-     Display display;
16-     String name;
17-     MyThread(Display display, String name){
18-         this.display=display;
19-         this.name=name;
20-     }
21-     public void run(){
22-         display.wish(name);
23-     }
24- }
25- class SynchronizedDemo {
26-     public static void main(String[] args) {
27-         Display display1 = new Display();
28-         Display display2 = new Display();
29-         MyThread t1=new MyThread(display1,"King");
30-         MyThread t2=new MyThread(display2,"Queen");
31-         t1.start();
32-         t2.start();
33-     }
34- }
```

```
Output
java -cp /tmp/10on5qvruX SynchronizedDemo
Good Morning : King
Good Morning : King
Good Morning : King
Good Morning : KingGood Morning : King
Good Morning : King
Good Morning : King
Good Morning : King
Good Morning : King
Good Morning : Queen
Good Morning : QueenGood Morning : Queen
Good Morning : QueenGood Morning : Queen
Good Morning : Queen
Good Morning : Queen
Good Morning : Queen
Good Morning : Queen
```

While any thread say T is executing a static synchronized method then any other thread cannot execute the same or other static synchronized method because to do it thread will require a class level lock which is acquired by thread T, but other threads can execute normal, static or synchronized methods without waiting for thread T to complete.

Synchronized Block

Suppose we are writing a method of say 1000 lines, in which only 10 lines play a role in synchronization then instead of making the whole method as synchronized, we can write those 10 lines in a synchronized block. This is because if we will make the whole method as synchronized then it will degrade our application performance because in synchronized method only one thread can enter to execute the operation at a time and execution of 1000 lines of code will take time and this will increase the waiting time of other threads.

We can take a simple example to understand this - Suppose we want to travel from Jaipur to Delhi and there is a major accident occurred in between Delhi and Jaipur, now instead of blocking the entire road from Jaipur to Delhi, it is always recommended to block the part of road where the accident occurs and deal accordingly.

We can declare synchronized block in three different ways.

To get the lock of current object

synchronized(this)

```
{
}
```

To get the lock of specified object say "display":

`synchronized(display)`

```
{  
}
```

To get the class level lock:

`synchronized(Display.class)`

```
{  
}
```

We cannot pass any primitive data type in synchronized block.

```
1  class Display  
2  {  
3      public void wish(String name)  
4      {  
5          ;;;;;;;;; // 1 lakh lines of code  
6          synchronized(this)  
7          {  
8              for(int i=1; i <= 10; i++)  
9              {  
10                 System.out.print("Hello : ");  
11                 try{  
12                     Thread.sleep(2000);  
13                 }catch(InterruptedException e){  
14                     System.out.println("Oops!! Got Exception");  
15                 }  
16                 System.out.println(name);  
17             }  
18         }  
19         ;;;;;;;;; // 1 lakh lines of code  
20     }  
21 }
```

8.. Thread Communication

Two threads say T1 and T2 can communicate with each other by using **wait()**, **notify()** and **notifyAll()** methods. The thread say T1 which is expecting a updation will call wait() method and enters into waiting state, while the thread which is responsible for making the updation say T2 will call notify() method after performing the updation. So once notify() method is call by T2, the thread T1 which is in waiting queue will get notification and continue its execution with updated data.

wait(), notify() and notifyAll() methods are present in Object class. They are the part of Object class but not of a Thread class because a thread can call all these three methods for the Object of any type like Student, Employee, Postbox etc.

Now, there is one very important point for these methods. Whenever the thread calls these three methods on any object, then that thread should be the owner of that object i.e., the thread should acquire the lock of that object. So, these methods should be called from either synchronized method or from the synchronized block, if we call these methods from the non- synchronized area of the program then it will throw an exception called **IllegalMonitorStateException**.

Main.java	Output
<pre>1 class MyThread extends Thread 2 { 3 int total = 0; 4 public void run() 5 { 6 for(int i=1; i <=100; i++) 7 { 8 total = total + i; 9 } 10 this.notify(); 11 } 12 } 13 class ThreadCommunication 14 { 15 public static void main(String[] args) throws InterruptedException 16 { 17 MyThread t=new MyThread(); 18 t.start(); 19 t.wait(); 20 System.out.println(t.total); 21 } 22 }</pre>	<pre>java -cp /tmp/nwi9oJ3txV ThreadCommunication Exception in thread "main" java.lang.IllegalMonitorStateException at java.base/java.lang.Object.wait(Native Method) at java.base/java.lang.Object.wait(Object.java:328) at ThreadCommunication.main(ThreadCommunication.java:19) Exception in thread "Thread-0" java.lang.IllegalMonitorStateException at java.base/java.lang.Object.notify(Native Method) at MyThread.run(ThreadCommunication.java:10)</pre>

Also, when any thread say T wants to call wait() method on any Object say X then immediately it will release the lock of that specific object X and it enters into waiting state after lock is released. A thread can hold the lock of multiple objects but it will release only that lock which is of specific object. When another thread say T1 calls notify() method on any object say X, it will release the lock but not immediately, possibility is after calling notify() method it will perform some other operation before releasing the lock. Except these three methods namely wait(), notify() and notifyAll() there is no other methods where Thread will release the lock.

notify() method will only notify the single thread present in a waiting queue of a specific object say X but notifyAll() method will notify all the threads present in a waiting queue of same object say X but execution will be performed one by one since execution require a lock which one thread will get at a time.

Every wait() methods in Java throws **InterruptedException**. We have the following prototype for wait() methods.

Wait Until Notify:

public final void wait() throws InterruptedException;

Wait for certain time in Milliseconds:

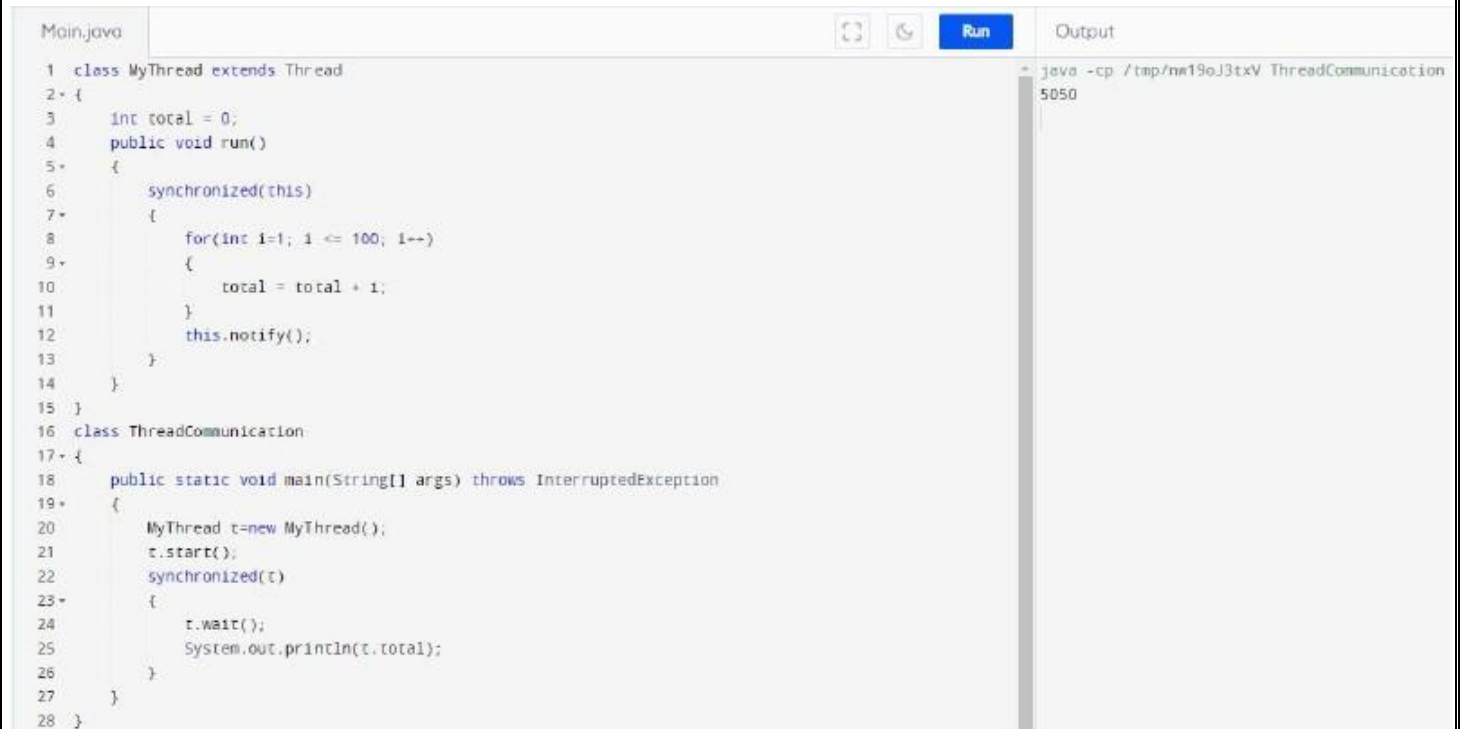
public final native void wait(long milliseconds) throws InterruptedException;

Wait for specific time specified in milliseconds and nanoseconds:

public final void wait(long milliseconds, int nanoseconds) throws InterruptedException;

public final native void notify();

public final native void notifyAll();



```
Main.java
1 class MyThread extends Thread
2 {
3     int total = 0;
4     public void run()
5     {
6         synchronized(this)
7         {
8             for(int i=1; i <= 100; i++)
9             {
10                 total = total + 1;
11             }
12             this.notify();
13         }
14     }
15 }
16 class ThreadCommunication
17 {
18     public static void main(String[] args) throws InterruptedException
19     {
20         MyThread t=new MyThread();
21         t.start();
22         synchronized(t)
23         {
24             t.wait();
25             System.out.println(t.total);
26         }
27     }
28 }
```

Output

```
* java -cp /tmp/nw19oJ3txV ThreadCommunication
5050
```

9.. Deadlock

If two threads say T1 and T2 are waiting for the completion of each other i.e., T1 is waiting for the operation of T2 to complete and T2 is waiting for the thread T1 to complete. This kind of situation is called deadlock situation. In this situation both the threads are going to wait forever.

Synchronized keyword is the only reason for deadlock to occur. So while using synchronized keyword we have to take a special care. There is no resolution once our program enters into deadlock situation but there are some preventive measures which we can take to avoid deadlock to occur.

```
31- class Deadlock extends Thread{
32     A a = new A();
33     B b = new B();
34     public void m1(){
35         this.start();
36         a.d1(b);
37     }
38     public void run(){
39         b.d2(a);
40     }
41     public static void main(String[] args){
42         Deadlock d = new Deadlock();
43         d.m1();
44     }
45 }
```

Main.java

```
1- class A {
2-     public synchronized void d1(B b){
3         System.out.println("Thread 1 Starts Execution of d1 method");
4         try{
5             Thread.sleep(5000);
6         }catch(InterruptedException e){
7             e.getMessage();
8         }
9         System.out.println("Thread 1 calls B last()");
10        b.last();
11    }
12    public synchronized void last(){
13        System.out.println("Inside A, last()");
14    }
15 }
16 class B {
17     public synchronized void d2(A a){
18         System.out.println("Thread 2 Starts Execution of d1 method");
19         try{
20             Thread.sleep(5000);
21         }catch(InterruptedException e){
22             e.getMessage();
23         }
24         System.out.println("Thread 2 calls A last()");
25         a.last();
26     }
27     public synchronized void last(){
28         System.out.println("Inside B, last()");
29     }
30 }
```

Output

Clear

```
java -cp /tmp/nw19oJ3txV Deadlock
Thread 1 Starts Execution of d1 method
Thread 2 Starts Execution of d1 method
Thread 1 calls B last()
Thread 2 calls A last()
```

Starvation is a process where two threads wait for a long time but it is sure that thread will continue its execution after some time.

10.. Thread Types

The following are the types of Threads:

- Daemon Threads
- Green Thread
- Native OS Thread

10.1 Daemon Threads

The thread which gets executed in the background are called daemon threads. For example – Garbage collector, Signal Dispatcher, Attach Listener etc. Daemon threads are used to provide support to non-daemon threads (main thread or any other child thread) to complete their executions without any issues. For example, if main thread is having sort of memory, then JVM internally will run garbage collector to destroy all the unused objects to provide free memory to main thread to complete its execution.

Usually, daemon threads with low priority but as work demands its priority can be increased and decreased whenever required.

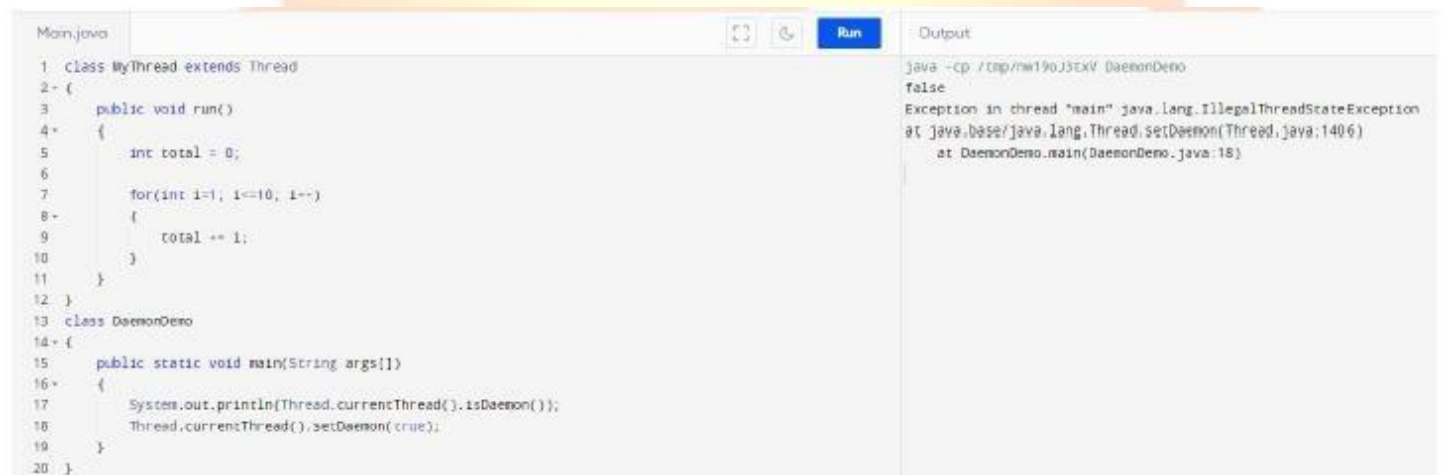
We can check for any thread if it is daemon or not by using method `isDaemon()`

public boolean isDaemon();

We can make any thread as a daemon thread by calling `setDaemon()` method

public void setDaemon(Boolean b);

But it is to be noted that we can make a thread as daemon thread before starting of that thread. If we are trying to change the thread to daemon thread while thread is running then it will throw a run time exception called **IllegalThreadStateException**.



```
1 class MyThread extends Thread
2 {
3     public void run()
4     {
5         int total = 0;
6
7         for(int i=1; i<=10; i++)
8         {
9             total += i;
10        }
11    }
12 }
13 class DaemonDemo
14 {
15     public static void main(String args[])
16     {
17         System.out.println(Thread.currentThread().isDaemon());
18         Thread.currentThread().setDaemon(true);
19     }
20 }
```

Output

```
java -cp ./bin/nw190j3EXV DaemonDemo
false
Exception in thread "main" java.lang.IllegalThreadStateException
at java.base/java.lang.Thread.setDaemon(Thread.java:1406)
at DaemonDemo.main(DaemonDemo.java:18)
```

By default, main thread is non daemon in nature but all other threads nature is inheriting from a parent thread. If parent thread is daemon all its child threads are daemon and if parent thread is non-daemon all its child threads are non-daemon.

It is impossible to change the daemon nature of main thread because it is already started by JVM at the beginning.

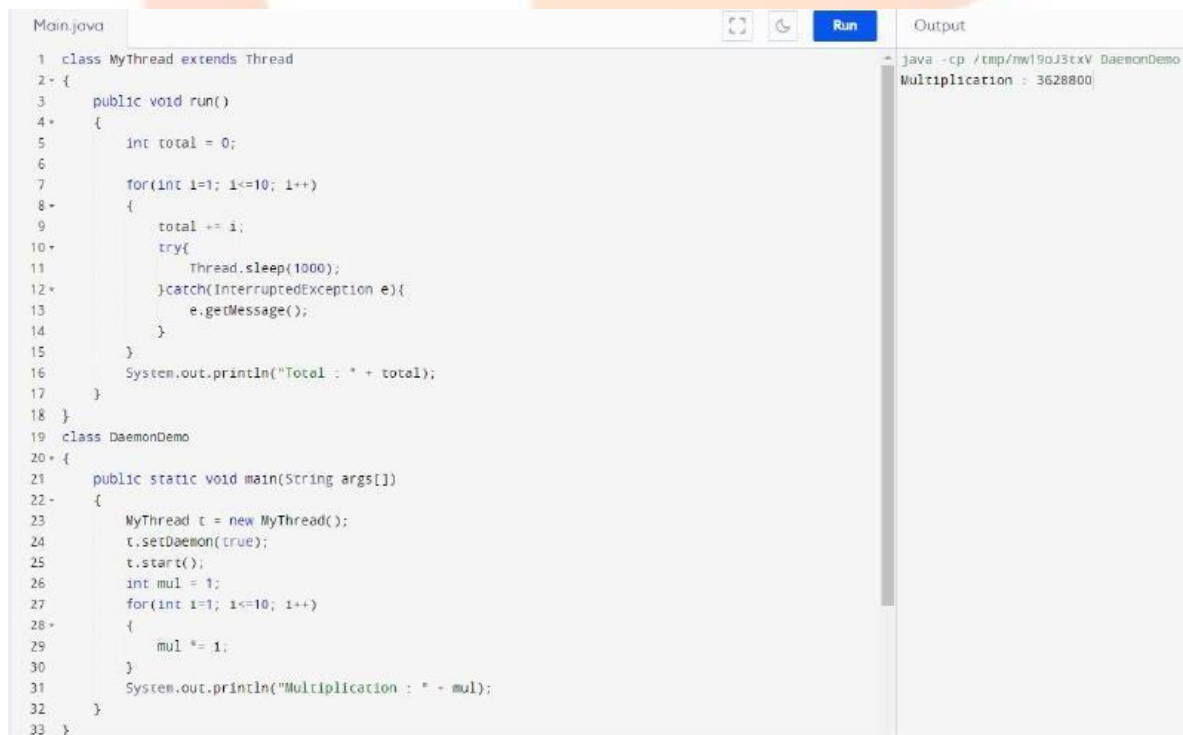


```
1 class MyThread extends Thread
2 {
3     public void run()
4     {
5         int total = 0;
6
7         for(int i=1; i<=10; i++)
8         {
9             total += 1;
10        }
11    }
12 }
13 class DaemonDemo
14 {
15     public static void main(String args[])
16     {
17         System.out.println(Thread.currentThread().isDaemon());
18         MyThread t = new MyThread();
19         System.out.println(t.isDaemon());
20         t.setDaemon(true);
21         System.out.println(t.isDaemon());
22     }
23 }
```

Output

```
java -cp /tmp/mw19oJ3txV DaemonDemo
falsefalse
true
```

Once last non-daemon thread terminates after completing the executions, then all daemon threads will be terminated automatically by the JVM irrespective of their position.



```
1 class MyThread extends Thread
2 {
3     public void run()
4     {
5         int total = 0;
6
7         for(int i=1; i<=10; i++)
8         {
9             total += 1;
10            try{
11                Thread.sleep(1000);
12            }catch(InterruptedException e){
13                e.getMessage();
14            }
15        }
16        System.out.println("Total : " + total);
17    }
18 }
19 class DaemonDemo
20 {
21     public static void main(String args[])
22     {
23         MyThread t = new MyThread();
24         t.setDaemon(true);
25         t.start();
26         int mul = 1;
27         for(int i=1; i<=10; i++)
28         {
29             mul *= i;
30         }
31         System.out.println("Multiplication : " + mul);
32     }
33 }
```

Output

```
java -cp /tmp/mw19oJ3txV DaemonDemo
Multiplication : 3628800
```

Here once main thread which is a non-daemon thread completed its execution and printed multiplication then child daemon thread is terminated automatically and no total is printed.

10.2 Green Thread

The thread which is completely managed by JVM without any underlying support of Operating system internally then that model is called Green Thread Model. Very few Operating system like Sun Solaris supports Green Thread model. Green Thread Model is completely deprecated now a days.

10.3 Native OS Thread

The thread which also takes the help of underlying Operating system as well for their management, they are called Native OS Thread Model. All windows based operating systems provide support for Native OS Thread Model.

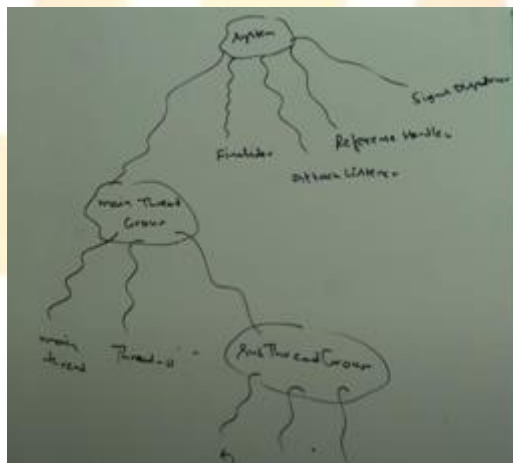
11.. ThreadGroup

Based on functionality, we can group multiple threads into a single unit called ThreadGroup. ThreadGroup contains threads and some sub thread groups as well. We can perform a common operation on multiple threads at the same time if we have maintained those threads in one thread group.

We can get the thread group to which thread belongs to by using **getThreadGroup()** methods.

Main.java	Run	Output
<pre>1 class ThreadDemo 2 { 3 public static void main(String args[]) 4 { 5 System.out.println("Main Thread Group : " + Thread.currentThread().getThreadGroup()); 6 System.out.println("Main Thread Group Name : " + Thread.currentThread().getThreadGroup().getName()); 7 System.out.println("Main Parent Thread Group Name : " + Thread.currentThread().getThreadGroup().getParent().getName()); 8 } 9 }</pre>		<pre>java -cp /tmp/nw19oJ3txV ThreadDemo Main Thread Group : java.lang.ThreadGroup[name=main,maxpri=10] Main Thread Group Name : main Main Parent Thread Group Name : system</pre>

Every thread in java belongs to some thread group. Main thread belongs to thread group called main. Every thread group in java is a child of system group either directly or indirectly. System group act as a root for all thread groups. System group contains several System level threads like Garbage collector, Attach listener, Signal dispatcher, Reference handler etc.



How to create a ThreadGroup now?

ThreadGroup is a Java class present in java.lang package. It is a direct child of Object class.

Various Constructors of ThreadGroup we have are:

ThreadGroup threadGroup = new ThreadGroup(String name);

Main.java	Run	Output
<pre>1 class ThreadDemo 2 { 3 public static void main(String args[]) 4 { 5 ThreadGroup threadGroup = new ThreadGroup("FirstThreadGroup"); 6 System.out.println(threadGroup.getName()); 7 System.out.println(threadGroup.getParent().getName()); 8 } 9 }</pre>		<pre>java -cp /tmp/nw19oJ3txV ThreadDemo FirstThreadGroup main</pre>

We can create a ThreadGroup as a child of specific parent ThreadGroup as well.

ThreadGroup threadGroup = new ThreadGroup(ThreadGroup tg, String name);

In below example, we have SecondThreadGroup which is a child of FirstThreadGroup which in fact is a child of main group and main group is a child of system group.

Main.java	Output
<pre> 1 class ThreadDemo 2 { 3 public static void main(String args[]) 4 { 5 System.out.println("First"); 6 ThreadGroup threadGroup1 = new ThreadGroup("FirstThreadGroup"); 7 System.out.println(threadGroup1.getName()); 8 System.out.println(threadGroup1.getParent().getName()); 9 10 System.out.println("Second"); 11 ThreadGroup threadGroup2 = new ThreadGroup(threadGroup1, "SecondThreadGroup"); 12 System.out.println(threadGroup2.getName()); 13 System.out.println(threadGroup2.getParent().getName()); 14 } 15 } </pre>	<pre> java -cp /tmp/nw19oJ3cxV ThreadDemo First FirstThreadGroup main Second SecondThreadGroup FirstThreadGroup </pre>

Methods of ThreadGroup Class

String getName()	Returns the Name of ThreadGroup.
int getMaxPriority()	Returns the Maximum Priority of ThreadGroup. Default MaxPriority of ThreadGroup is 10
void setMaxPriority(int P)	Set the Maximum priority of ThreadGroup to P. If there are already threads with priority higher than P in a ThreadGroup then that priority will not be impacted only new threads added will be set to priority as P.
ThreadGroup getParent()	Returns the parent ThreadGroup of current thread.
void list()	It prints Information about ThreadGroup on the console.
int activeCount()	Returns number of active threads present in a ThreadGroup.
int activeGroupCount()	Returns the number of active groups present in a current ThreadGroup.
int enumerate(Thread[] t)	Copy all active threads of this ThreadGroup to the specified array t. In this sub thread groups are also considered.
int enumerate(ThreadGroup[] tg)	Copy all sub thread groups of this ThreadGroup to the specified thread group tg.
boolean isDaemon()	To Check if thread group is daemon or not.
void setDaemon(Boolean b)	To set the nature of thread group as daemon.
void interrupt()	Interrupt all waiting or sleeping threads present in a ThreadGroup.
void destroy()	To destroy Thread groups and its sub Thread Groups.

Main.java	Output
<pre> 1 class ThreadDemo 2 { 3 public static void main(String args[]) 4 { 5 ThreadGroup threadGroup1 = new ThreadGroup("FirstThreadGroup"); 6 Thread t1 = new Thread(threadGroup1, "T1"); 7 Thread t2 = new Thread(threadGroup1, "T2"); 8 threadGroup1.setMaxPriority(3); 9 Thread t3 = new Thread(threadGroup1, "T3"); 10 System.out.println(t1.getPriority()); 11 System.out.println(t2.getPriority()); 12 System.out.println(t3.getPriority()); 13 } 14 } </pre>	<pre> java -cp /tmp/nw19oJ3cxV ThreadDemo 5 5 3 </pre>

Main.java	Run	Output
<pre> 1 class MyThread extends Thread 2 { 3 MyThread(ThreadGroup g, String name) 4 { 5 super(g,name); 6 } 7 public void run() 8 { 9 System.out.println("Child Thread"); 10 try{ 11 Thread.sleep(5000); 12 }catch(InterruptedException e){ 13 e.getMessage(); 14 } 15 } 16 } 17 class ThreadDemo 18 { 19 public static void main(String args[]) throws InterruptedException 20 { 21 ThreadGroup pg = new ThreadGroup("ParentGroup"); 22 ThreadGroup cg = new ThreadGroup(pg, "ChildGroup"); 23 MyThread t1 = new MyThread(pg,"ChildThread1"); 24 MyThread t2 = new MyThread(pg,"ChildThread2"); 25 t1.start(); 26 t2.start(); 27 System.out.println(pg.activeCount()); 28 System.out.println(pg.activeGroupCount()); 29 pg.list(); 30 Thread.sleep(10000); 31 System.out.println(pg.activeCount()); 32 System.out.println(pg.activeGroupCount()); 33 pg.list(); 34 } 35 } </pre>	Run	<pre> java -cp /tmp/nw19oJ3cxV ThreadDemo Child Thread Child Thread 2 1 java.lang.ThreadGroup[name=ParentGroup,maxpri=10] Thread[ChildThread1,5,ParentGroup] Thread[ChildThread2,5,ParentGroup] java.lang.ThreadGroup[name=ChildGroup,maxpri=10] 0 1 java.lang.ThreadGroup[name=ParentGroup,maxpri=10] java.lang.ThreadGroup[name=ChildGroup,maxpri=10] </pre>

Main.java	Run	Output
<pre> 1 class ThreadDemo 2 { 3 public static void main(String args[]) throws InterruptedException 4 { 5 ThreadGroup s = Thread.currentThread().getThreadGroup().getParent(); 6 Thread[] t = new Thread[s.activeCount()]; 7 s.enumerate(t); 8 for(Thread t1:t) 9 { 10 System.out.println(t1.getName()+"-----"+t1.isDaemon()); 11 } 12 } 13 } </pre>	Run	<pre> java -cp /tmp/nw19oJ3cxV ThreadDemo Reference Handler-----true Finalizer-----true Signal Dispatcher-----true main-----false Common-Cleaner-----true </pre>

12.. Lock Concept

The problems with traditional synchronized keyword include:

We don't have flexibility to try for the lock without waiting.

There is no way to specify a maximum waiting time for a thread to get the lock so that thread will wait until getting the locks which may create performance problems and causedeadlock.

Once the thread releases the lock then which thread from the waiting queue will get the thread, developer has no control over it.

There is no API to list out all waiting threads for a lock.

We can use synchronized keyword either at the method level or within the method as a block. But we cannot use it across multiple methods.

To overcome these problems of traditional synchronized keyword, in java 1.5 java.util.concurrent.Lock gets introduced. It provides several enhancements to the programmer to provide more control and concurrency.

Lock Interface

Lock object is similar to implicit lock acquired by the Thread to execute synchronized method or the synchronized block. Lock implementations provides more extensive operations than traditional implicit locks.

Important methods of Lock Interface:

- 10.3.1 **void lock()** : We can use this method to acquire a lock. If lock is already available then current thread will immediately get that lock. If lock is not already available then it will wait until getting the lock. It is exactly same behavior of traditional synchronized keyword.
- 10.3.2 **void unlock()** : This method is used to release the lock. To call this method it is compulsory that current thread is the owner of lock or holding the lock otherwise we will get run time exception saying IllegalMonitorStateException.
- 10.3.3 **boolean tryLock()** : To acquire a lock without waiting we use tryLock() method. If lock is available then thread will acquire that lock and returns true and else return false and can continue its execution without waiting. In this case thread never enters into waiting state.
- 10.3.4 **boolean tryLock(long time, TimeUnit unit)** : It will also to acquire a lock but here it will wait for specified time and TimeUnit like 1 Hour or so on and within that time if lock gets available it will return true and else return false and can continue its execution. TimeUnit is an Enum present in java.util.concurrent package.

```
1 enum TimeUnit{
2     NANOSECONDS,
3     MICROSECONDS,
4     MILLISECONDS,
5     SECONDS,
6     MINUTES,
7     HOURS,
8     DAYS;
9 }
```

- 10.3.5 **void lockInterruptibly()** : Acquires the lock and returns immediately if lock is available. If lock is not available then it will wait. While waiting if thread gets interrupted then it won't get the lock while if the thread is not interrupted in between then only it will get the lock.

ReentrantLock

This is an implementation class of Lock interface. It is the direct child of Object class. Reentrant means a thread can acquire same lock multiple times without any issue. Internally ReentrantLock increments Threads personal count whenever we call lock() method and decrements the count whenever thread calls unlock() method and lock will be released whenever the count reaches 0.

Constructors of ReentrantLock

Creates an instance of ReentrantLock:

```
ReentrantLock l = new ReentrantLock(); // False fairness default value
```

```
ReentrantLock l = new ReentrantLock(boolean fairness);
```

Creates a Reentrant lock with fairness policy. If fairness is set to true then longest waiting thread in the waiting queue will get the first chance to get the lock if available and execute. If the value is false then which waiting thread will get the first chance to execute, we cannot predict it. By default, value of fairness is false. Internally, it follows first come first serve policy.

Since ReentrantLock is an implementation class of Lock Interface so it provides the implementation of all 5 methods of Lock interface. Additionally, it has some methods as follows:

- 10.3.6 **int getHoldCount():** Returns the number of Holds on this lock by current Thread.
- 10.3.7 **boolean isHeldByCurrentThread():** Returns true if lock is hold by current thread else returns false.
- 10.3.8 **int getQueueLength():** Returns the length of waiting queue i.e., return the number of threads waiting in the waiting queue for the lock.
- 10.3.9 **Collection getQueueThreads():** Return the collection of Threads waiting in the waiting queue for the lock.
- 10.3.10 **boolean hasQueuedThreads():** Returns true if any thread waiting to get the lock.
- 10.3.11 **boolean isLocked():** Returns true if the lock is acquired by some thread.
- 10.3.12 **boolean isFair():** Returns true if fairness policy is set with true value.
- 10.3.13 **Thread getOwner():** Returns the thread which acquires the

lock. Example of Code Snippet:

Main.java	Run	Output
<pre>1 import java.util.concurrent.locks.*; 2 3 class ReentrantLockDemo { 4 public static void main(String[] args) 5 { 6 ReentrantLock l = new ReentrantLock(); 7 l.lock(); 8 l.lock(); 9 System.out.println(l.getHoldCount()); 10 System.out.println(l.isLocked()); 11 System.out.println(l.isHeldByCurrentThread()); 12 System.out.println(l.getQueueLength()); 13 l.unlock(); 14 System.out.println(l.getHoldCount()); 15 System.out.println(l.isLocked()); 16 l.unlock(); 17 System.out.println(l.isLocked()); 18 System.out.println(l.isFair()); 19 } 20 }</pre>		<pre>java -cp /tmp/U7yc5L5tG1 ReentrantLockDemo 2 true true 0 1 true false false</pre>

Let's understand the use of non-synchronized method, synchronized method and Reentrant Lock with the same example.


```

2- class Display{
3     ReentrantLock l = new ReentrantLock();
4     public void wish(String name){
5         for(int i=1; i <=10; i++){
6             l.lock();
7             System.out.print("Hello!! ");
8             try{
9                 Thread.sleep(2000);
10            }catch(InterruptedException e){
11                e.getMessage();
12            }
13            System.out.println(name);
14            l.unlock();
15        }
16    }
17 }
18- class MyThread extends Thread{
19     Display display;
20     String name;
21     MyThread(Display display, String name){
22         this.display = display;
23         this.name = name;
24     }
25     public void run(){
26         display.wish(name);
27     }
28 }
29- class ReentrantLockDemo {
30-     public static void main(String[] args) {
31         Display d = new Display();
32         MyThread t1 = new MyThread(d, "King");
33         MyThread t2 = new MyThread(d, "Queen");
34         t1.start();
35         t2.start();
36     }

```

```

* java -cp /tmp/U7yc5L5tG1 ReentrantLockDemo
Hello!! King
Hello!! King
Hello!! King
Hello!! King
Hello!! King
Hello!! King
Hello!! King
Hello!! King
Hello!! King
Hello!! Queen
Hello!! Queen
Hello!! Queen
Hello!! Queen
Hello!! Queen
Hello!! Queen
Hello!! Queen
Hello!! Queen

```

Let's understand tryLock() method with the simple Java Program.

```

Main.java
1- import java.util.concurrent.locks.*;
2
3- class MyThread extends Thread{
4     static ReentrantLock l = new ReentrantLock();
5     MyThread(String name){
6         super(name);
7     }
8     public void run(){
9         if(l.tryLock()){
10             System.out.println(Thread.currentThread().getName()+"...Got Lock and Performing safe
operations");
11             try{
12                 Thread.sleep(2000);
13             }catch(InterruptedException e){
14                 e.getMessage();
15             }
16             l.unlock();
17         }
18         else{
19             System.out.println(Thread.currentThread().getName()+"...Unable to got lock and performing
alternative Operations");
20         }
21     }
22 }
23- class ReentrantLockDemo {
24     public static void main(String[] args) {
25         MyThread t1 = new MyThread("King");
26         MyThread t2 = new MyThread("Queen");
27         t1.start();
28         t2.start();
29     }
30 }

```

```

* java -cp /tmp/U7yc5L5tG1 ReentrantLockDemo
King...Got Lock and Performing safe operations
Queen...Unable to got lock and performing alternative Operations

```

Let's understand how to use tryLock with TimeUnit.


```
Main.java
3> class MyThread extends Thread{
4     static ReentrantLock l = new ReentrantLock();
5     MyThread(String name){
6         super(name);
7     }
8     public void run(){
9         do{
10             try{
11                 if(l.tryLock(5000, TimeUnit.MILLISECONDS)){
12                     System.out.println(Thread.currentThread().getName()+"...Got Lock and Performing
13                         safe operations");
14                     Thread.sleep(30000);
15                     l.unlock();
16                     System.out.println(Thread.currentThread().getName()+"...Releases Lock");
17                     break;
18                 }
19                 else{
20                     System.out.println(Thread.currentThread().getName()+"...Unable to get lock and will
21                         try again");
22                 }
23             }catch(InterruptedException e){
24                 e.getMessage();
25             }
26         }while(true);
27     }
28 }
29 class ReentrantLockDemo {
30     public static void main(String[] args) {
31         MyThread t1 = new MyThread("King");
32         MyThread t2 = new MyThread("Queen");
33         t1.start();
34         t2.start();
35     }
36 }
```

Output

```
* java -cp /tmp/U7ycSL5tG1 #reentrantLockDemo
King...Got Lock and Performing safe operationsQueen...Unable to get lock and will try again
Queen...Unable to get lock and will try again
Queen...Unable to get lock and will try again
Queen...Unable to get lock and will try again
Queen...Unable to get lock and will try again
Queen...Unable to get lock and will try again
Queen...Got Lock and Performing safe operations
King...Releases Lock
```



13.. Thread Pools (Executor Framework)

Creating a new Thread for every job may cause performance and memory related problems. To Overcome this, we should go for thread pool. In thread pools we have some already created threads which are ready to execute our job. Java 1.5 Version introduces thread pool framework also known as executor framework to create thread pools.

To Create Thread Pool

ExecutorService service = Executors.newFixedThreadPool(int numberOfThreads);

Example:

To Creates a Thread pool of 3 Threads.

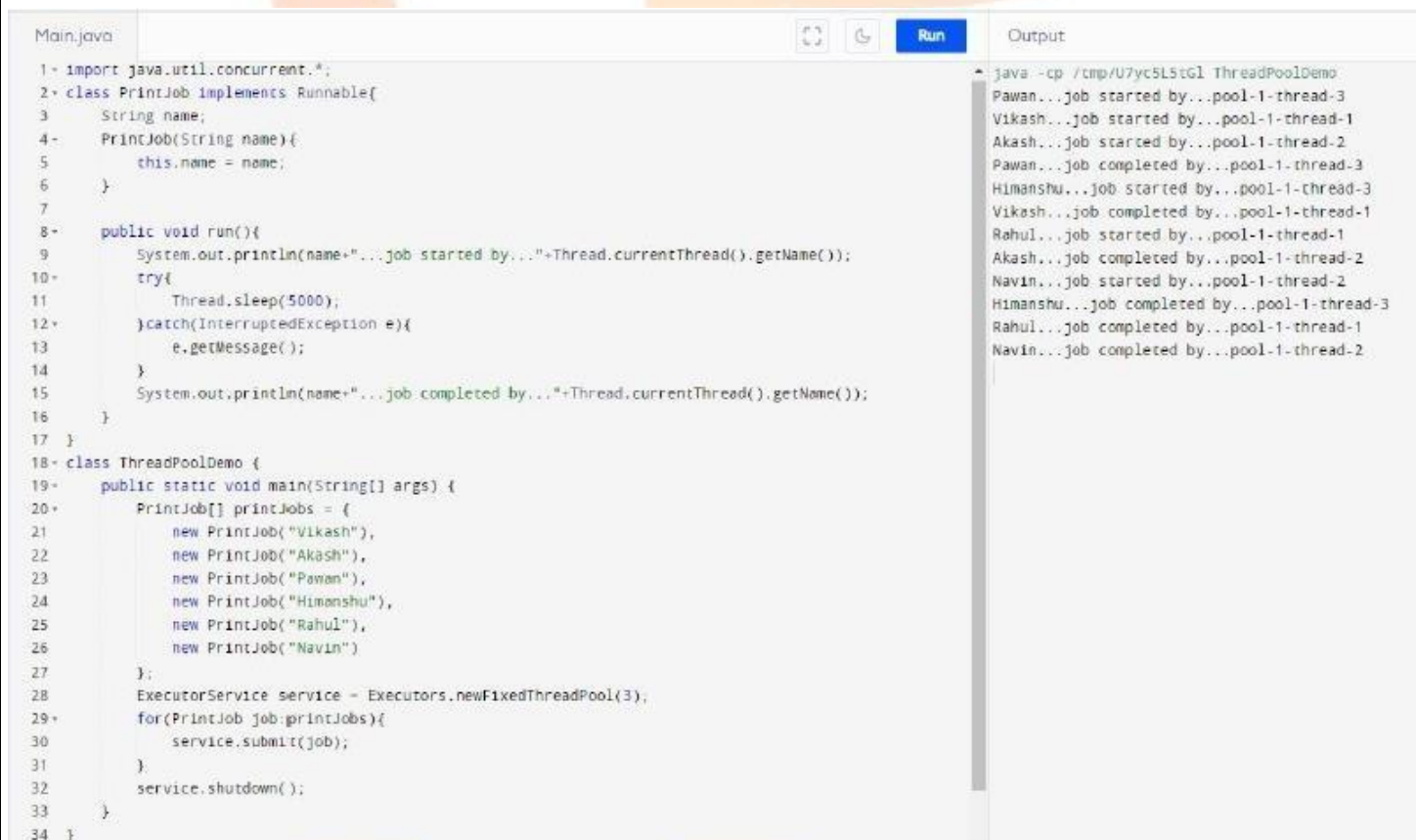
ExecutorService service = Executors.newFixedThreadPool(3);

We can submit a runnable job to the ExecutorService with following method:

service.submit(job);

We can shut down ExecutorService with below Method:

service.shutdown();



The screenshot shows an IDE with a file named 'Main.java'. The code defines a 'PrintJob' class implementing 'Runnable' and a 'ThreadPoolDemo' class. The 'ThreadPoolDemo' class creates a thread pool of size 3 and submits six 'PrintJob' objects. The output window shows the execution results, indicating that jobs are started and completed by different threads in the pool.

```
1- import java.util.concurrent.*;
2- class PrintJob implements Runnable{
3-     String name;
4-     PrintJob(String name){
5-         this.name = name;
6-     }
7-
8-     public void run(){
9-         System.out.println(name+"...job started by..." + Thread.currentThread().getName());
10-        try{
11-            Thread.sleep(5000);
12-        }catch (InterruptedException e){
13-            e.getMessage();
14-        }
15-        System.out.println(name+"...job completed by..." + Thread.currentThread().getName());
16-    }
17- }
18- class ThreadPoolDemo {
19-     public static void main(String[] args) {
20-         PrintJob[] printJobs = {
21-             new PrintJob("Vikash"),
22-             new PrintJob("Akash"),
23-             new PrintJob("Pawan"),
24-             new PrintJob("Himanshu"),
25-             new PrintJob("Rahul"),
26-             new PrintJob("Navin")
27-         };
28-         ExecutorService service = Executors.newFixedThreadPool(3);
29-         for(PrintJob job:printJobs){
30-             service.submit(job);
31-         }
32-         service.shutdown();
33-     }
34- }
```

Output:

```
java -cp /tmp/U7yc5LSstG1 ThreadPoo1Demo
Pawan...job started by...pool-1-thread-3
Vikash...job started by...pool-1-thread-1
Akash...job started by...pool-1-thread-2
Pawan...job completed by...pool-1-thread-3
Himanshu...job started by...pool-1-thread-3
Vikash...job completed by...pool-1-thread-1
Rahul...job started by...pool-1-thread-1
Akash...job completed by...pool-1-thread-2
Navin...job started by...pool-1-thread-2
Himanshu...job completed by...pool-1-thread-3
Rahul...job completed by...pool-1-thread-1
Navin...job completed by...pool-1-thread-2
```

Here we have created a thread pool of size 3. So at a time 3 threads are created to perform 3 print jobs.

Note: While developing web or application servers we can use Thread Pool concept.

14.. ThreadLocal

ThreadLocal class provides thread local variables. ThreadLocal class maintains values per Thread basis. Each ThreadLocal Object maintains a separate value like UserId, TransactionId etc. for each thread that access that object. Thread can access its local value, manipulate 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 variable.

Example:

Consider a Servlet which invokes some business method. We have a requirement to generate a unique Transaction ID for each and every request and we have to pass this transaction Id to the business methods. For this requirement, we can use a ThreadLocal to maintain a separate Transaction ID for every request i.e., for every Thread.

ThreadLocal class introduced in Java version 1.2 and enhanced in 1.5 version. ThreadLocal can be associated with Thread Scope. Total code which is executed by the Thread has access to the corresponding ThreadLocal Variable.

A Thread can access its own local variables and can't access other thread local variables.

Once thread enters into dead state, all its values are still by default eligible for garbage collection.

How to Create ThreadLocal Variable now?

```
ThreadLocal tl = new ThreadLocal();
```

Methods of ThreadLocal Variables:

- **Object get():** Returns the value of ThreadLocal Variable associated with Current Thread.
- **Object initialValue():** Returns initial value of ThreadLocal Variable associated with Current Thread. Default implementation of this method return null. To Customize our own initialValue we need to override this method.
- **void set(Object newValue):** To set a new value to the ThreadLocal Variable of a Current Thread.
- **void remove():** To remove the value of ThreadLocal variable associated to a Current Thread. It is newly added method in version 1.5. After removal if we try to access it will internally call its initial value which is now null.

<pre>Main.java 1 class ThreadLocalDemo { 2 public static void main(String[] args) throws Exception { 3 ThreadLocal tl = new ThreadLocal(); 4 System.out.println(tl.get()); 5 tl.set("Vikash"); 6 System.out.println(tl.get()); 7 tl.remove(); 8 System.out.println(tl.get()); 9 } 10 }</pre>	<pre>Run</pre>	<pre>Output java -cp /tmp/VeLtvJVwNC ThreadLocalDemo null Vikash null</pre>
--	----------------	---

Let's Customize our own initial value instead of null.

Main.java	Run	Output
<pre> 1- class ThreadLocalDemo { 2 public static void main(String[] args) throws Exception 3 { 4 ThreadLocal t1 = new ThreadLocal() 5 { 6 public Object initialValue() 7 { 8 return "Mr"; 9 } 10 }; 11 System.out.println(t1.get()); 12 t1.set("Vikash"); 13 System.out.println(t1.get()); 14 t1.remove(); 15 System.out.println(t1.get()); 16 } 17 } </pre>	Run	<pre> java -cp /tmp/VeLTVJYwNC ThreadLocalDemo Mr Vikash Mr </pre>

Let's now write one program where every thread has their own separate Value which is an actual functionality of ThreadLocal Class. Here in the below program for each CustomerThread, Separate Customer ID is maintained by ThreadLocal.

Main.java	Run	Output
<pre> 1- class CustomerThread extends Thread { 2 static int custId = 0; 3 private static ThreadLocal t1 = new ThreadLocal() { 4 public Object initialValue(){ 5 return ++custId; 6 } 7 }; 8 CustomerThread(String name){ 9 super(name); 10 } 11 public void run(){ 12 System.out.println(Thread.currentThread().getName() + " executing with customer id " + t1.get()); 13 } 14 } 15- class ThreadLocalDemo { 16 public static void main(String[] args) { 17 CustomerThread c1 = new CustomerThread("Customer-Thread-1"); 18 CustomerThread c2 = new CustomerThread("Customer-Thread-2"); 19 CustomerThread c3 = new CustomerThread("Customer-Thread-3"); 20 CustomerThread c4 = new CustomerThread("Customer-Thread-4"); 21 c1.start(); 22 c2.start(); 23 c3.start(); 24 c4.start(); 25 } 26 } </pre>	Run	<pre> java -cp /tmp/VeLTVJYwNC ThreadLocalDemo Customer-Thread-1 executing with customer id 1 Customer-Thread-3 executing with customer id 3 Customer-Thread-2 executing with customer id 2 Customer-Thread-4 executing with customer id 4 </pre>

Parent Thread's ThreadLocal value is not available to Child Thread. Let's Demonstrate this with one simple example:

Main.java	Run	Output
<pre> 1- class ParentThread extends Thread { 2 public static ThreadLocal t1 = new ThreadLocal(); 3 4 public void run(){ 5 t1.set("PP"); 6 System.out.println("Parent Thread Value : " + t1.get()); 7 ChildThread ct = new ChildThread(); 8 ct.start(); 9 } 10 } 11- class ChildThread extends Thread { 12 public void run(){ 13 System.out.println("Child Thread Value : " + ParentThread.t1.get()); 14 } 15 } 16- class ThreadLocalDemo { 17 public static void main(String[] args) { 18 ParentThread pt = new ParentThread(); 19 pt.start(); 20 } 21 } </pre>	Run	<pre> java -cp /tmp/VeLTVJYwNC ThreadLocalDemo Parent Thread Value : PP Child Thread Value : null </pre>

If we want to make the Parent thread Value also available to Child Thread then instead of ThreadLocal we should go for InheritableThreadLocal Class.

Main.java	Output
<pre>1- class ParentThread extends Thread { 2- public static InheritableThreadLocal t1 = new InheritableThreadLocal(); 3- 4- public void run(){ 5- t1.set("PP"); 6- System.out.println("Parent Thread Value : " + t1.get()); 7- ChildThread ct = new ChildThread(); 8- ct.start(); 9- } 10- } 11- class ChildThread extends Thread { 12- public void run(){ 13- System.out.println("Child Thread Value : " + ParentThread.t1.get()); 14- } 15- } 16- class ThreadLocalDemo { 17- public static void main(String[] args) { 18- ParentThread pt = new ParentThread(); 19- pt.start(); 20- } 21- }</pre>	<pre>java -cp /tmp/VeLTVJYwNC ThreadLocalDemo Parent Thread Value : PP Child Thread Value : PP</pre>

Here in above program we have got the parent thread value also in child Value i.e., both values are "PP" what if I want to override the child Value. This we can do by overriding childValue().

Main.java	Output
<pre>1- class ParentThread extends Thread { 2- public static InheritableThreadLocal t1 = new InheritableThreadLocal() 3- { 4- public Object childValue(Object P){ 5- return "CC"; 6- } 7- }; 8- public void run(){ 9- t1.set("PP"); 10- System.out.println("Parent Thread Value : " + t1.get()); 11- ChildThread ct = new ChildThread(); 12- ct.start(); 13- } 14- } 15- class ChildThread extends Thread { 16- public void run(){ 17- System.out.println("Child Thread Value : " + ParentThread.t1.get()); 18- } 19- } 20- class ThreadLocalDemo { 21- public static void main(String[] args) { 22- ParentThread pt = new ParentThread(); 23- pt.start(); 24- } 25- }</pre>	<pre>java -cp /tmp/VeLTVJYwNC ThreadLocalDemo Parent Thread Value : PP Child Thread Value : CC</pre>

Thread Scheduler

Thread scheduler is the one which is responsible for scheduling the threads waiting for execution. Thread scheduler can internally use any of the algorithms like FIFO, Round Robin for scheduling the Threads. This order of scheduling of main thread and child thread changes for every run and for every JVM. This is the reason why we might not get the same output for every run. Any combination of Main and Child thread is the possible output.

