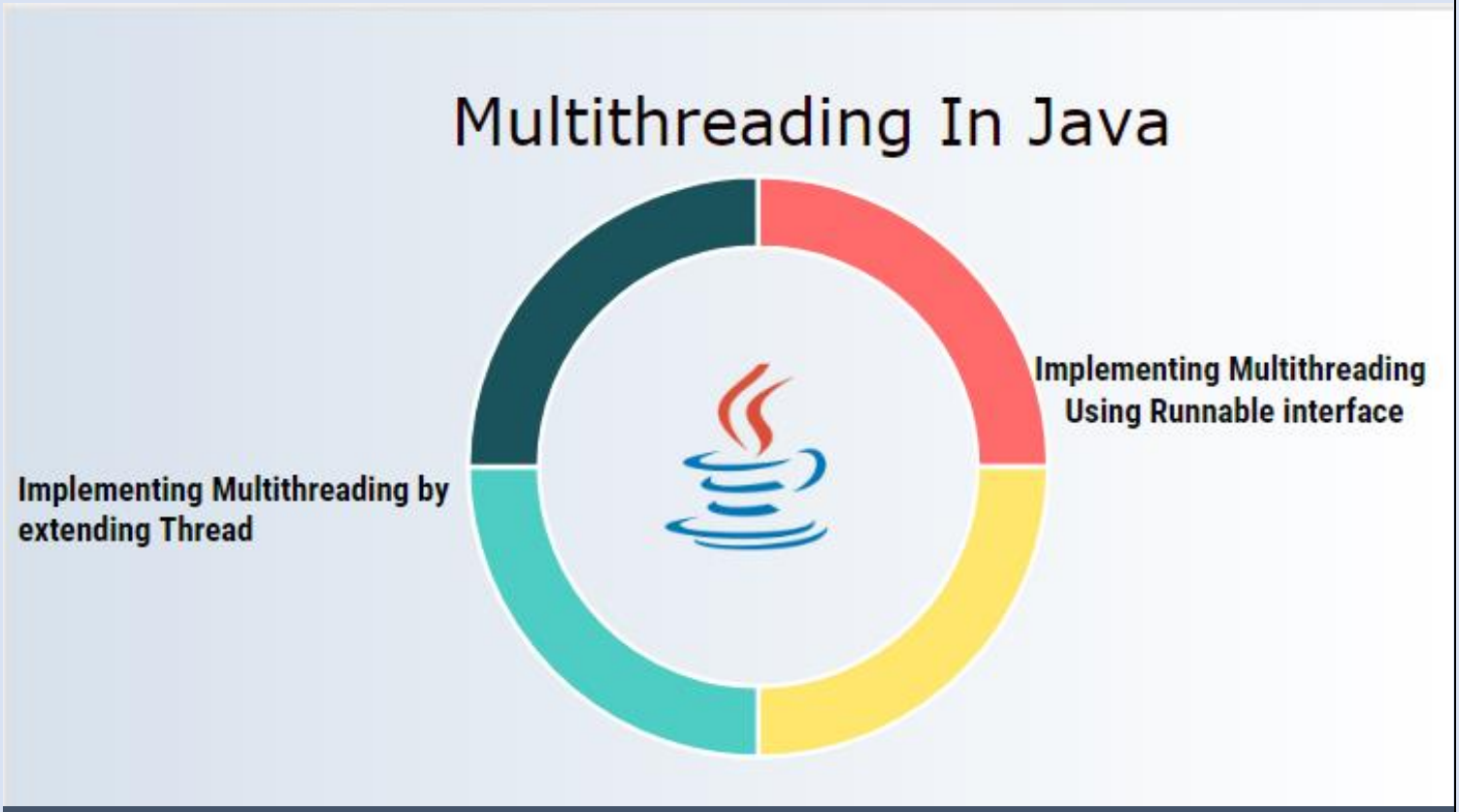


Multithreading

Uday Sharma

mrudaysharma4600@gmail.com

JAVA

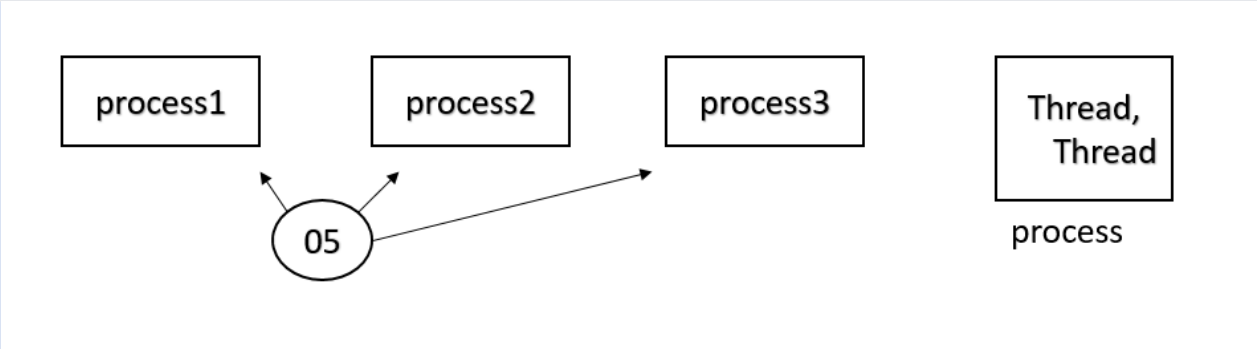
Full Stack Java Developer Series

Topics Covered:-

Multithreading in Java:-
Multitasking
What is Thread ?
Java Thread Class
Ways To Create A Thread In Java:-
Life Cycle of Thread
Thread Scheduler
Constructor In Multithreading
Sleeping a Thread
Start a Thread twice
Calling run() method
Joining a Thread
Naming a Thread
Thread Priority
Daemon Thread
Thread Pool
Thread Group
Shutdown Hook
Garbage Collection
Runtime Class

- ***Multithreading in Java:-***

- Multiprocessing and multithreading both are used to achieve multitasking.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.



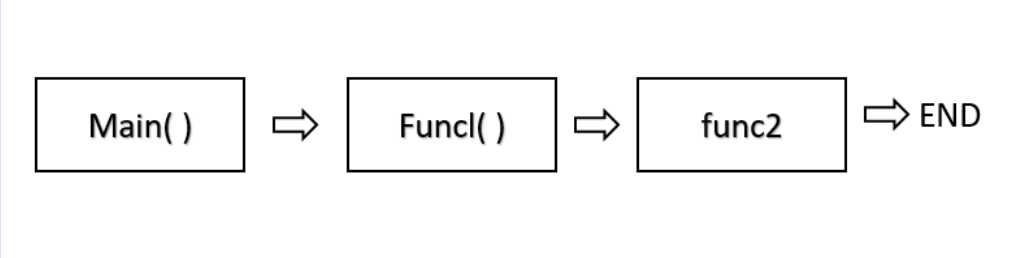
In a nut shell.....

- threads use shared memory area
- threads = faster content switching
- A thread is light-weight where a process is heavyweight

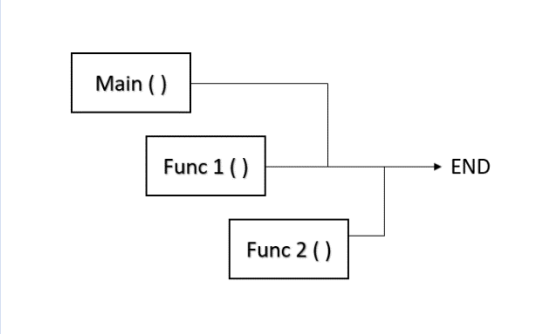
for example = A word processor can have one thread running in foreground as an editor and another in the background auto saving the document !

- ***Flow of control in java***

- Without threading :



- With threading :



- ***Creating a Threading:-***

There are two ways to create a thread in java

1. By extending thread class .
2. By implementing Runnable interface .

- ***Advantages of Java Multithreading: -***

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
2) You **can perform many operations together, so it saves time.**
3) Threads are **independent,** so it doesn't affect other threads if an exception occurs in a single thread.

- ***Multitasking***

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (**Multiprocessing**).
- Thread-based Multitasking (**Multithreading**).

1) ***Process-based Multitasking (Multiprocessing):-***

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading):-

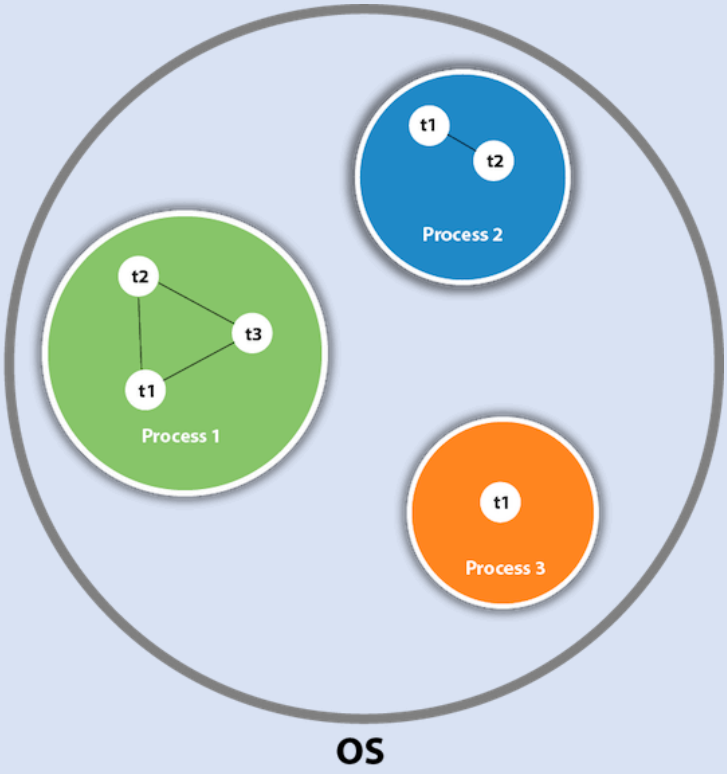
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

- ***What is Thread in java:-***

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Note: At a time one thread is executed only.

- ***Java Thread class:-***

Java provides Thread class to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

- ***Creating a Thread by Extending Thread class***

- ***Multithreading In Java:-***
  - Multithreading shows the concurrency. Concurrency is task of running and managing the multiple computation at the same time. While parallelism is the task of running multiple computation simultaneously. concurrency can be done by using a sinlge processing unit.
  - Used to maximize the CPU utilization.
  - We don't want our CPU to be in a free state; for example, Func1() comes into the memory and demands any input/output process. The CPU will need to wait for unit Func1() to complete its input/output operation in such a condition. But, while Func1() completes its I/O operation, the CPU is free and not executing any thread. So, the efficiency of the CPU is decreased in the absence of multithreading.
  - In the case of multithreading, if a thread demands any I/O operation, then the CPU will let the thread perform its I/O operation, but it will start the execution of a new thread parallelly. So, in this case, two threads are executing at the same time.

1*. Without threading:*

```
class ThreadExample{
   public static void main(String[] args) {
      Func1();
      Func2();
   }
}
```

In the above code, you can see that Func1() and Func2() are called inside the **main()** function. But the execution of Func2() will start only after the completion of the Func1().

2. With threading:-

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi func1=new Multi();
func1.start();
Multi func2=new Multi();
func2.start();
}}
```

Again, Func1() and Func2() are called inside the main function, but none of the two functions is waiting for the execution of the other function. Both the functions are getting executed concurrently.

- ***Ways To Create A Thread In Java:-***
  1. By extending the thread class

2. By implementing a Runnable interface

Let's see how we can create a thread by extending the thread class.

- ***Extending Thread Class :- (using the " extends" and "Thread" keywords)***

To create a thread using the thread class, we need to extend the thread class. Java's multithreading system is based on the thread class.

```
class MyThread extends Thread{
   @Override
   public void run(){
                  //code that we want to get executed on running the thread   }
   }
```

- In the above code, we're first inheriting the Thread class and then overriding the **run()** method.
- The code you want to execute on the thread's execution goes inside the run() method.

Code:-

```
package MultiThreading_in_Java;
// Creating a Thread by Extending the class
class MyTHread1 extends Thread{
    @Override
    public void run() {
        int i=0;
        while(i<5) {
            System.out.println("the thread of class 1 is running ");
            System.out.println(i);
            i++;
        }}}
//Creating another Thread by Extending the class
class MyTHread2 extends Thread{
    @Override
    public void run() {
        int i=0;
        while(i<5) {
            System.out.println("the thread of class 2 is running ");
            System.out.println(i+1);
            i++;
        }}}
public class Creating_a_Thread {
    public static void main(String[] args) {
//   creating an oblect of the thread classes
        MyTHread1 th1=new MyTHread1();
        MyTHread2 th2=new MyTHread2();
//      Both the Functions is running simultaneously
        th1.start();    // start method is given by the jvm
        th2.start();    // it is used start thread
    }
}
```

```
the thread of class 2 is running
1
the thread of class 2 is running
2
the thread of class 2 is running
3
the thread of class 2 is running
4
the thread of class 2 is running
5
the thread of class 1 is running
0
the thread of class 1 is running
1
the thread of class 1 is running
2
the thread of class 1 is running
3
the thread of class 1 is running
4
```

Output:-

By using the Thread in java by extending the class Both function
Runs Simultaneously

- If we don't use the thread method then the method **run()** of class MyThread1 runs first than

the method run()of class 2nd runs .

- ***start() :-***

Causes this thread to begin execution ; the java virtual machine calls the run method of this thread. This method is use using the public void type.

- ***Creating a Java Thread Using "Runnable" Interface:-***

In the previous tutorial, I told you that there are two ways to create a thread in java :

1. By Extending Thread Class
2. By implementing Runnable interface

- ***Steps To Create A Java Thread Using "Runnable" Interface:-***

1. Create a class and implement the Runnable interface by using the implements keyword.
2. Override the run() method inside the implementer class.
3. Create an object of the implementer class in the main() method.
4. Instantiate the Thread class and pass the object to the Thread constructor.
5. Call start() on the thread. start()will call the run()method.

Code:-

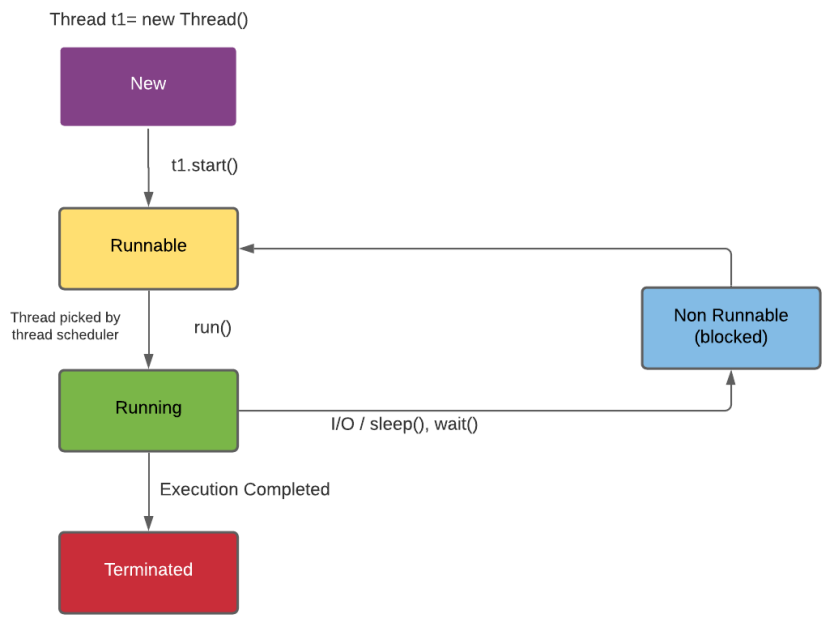
```
package MultiThreading_in_Java;
//Creating a Runnable interface by Extending the class
class MyRUNAbleTHread1 implements Runnable{
    @Override  // overriding the method so it runs freely
    public void run() {
        int i=0;
        while(i<5) {
            System.out.println("the thread of class 1 is running ");
            System.out.println(i);
            i++;
        }}}
//Creating another Runnable interface by Extending the class
class MyRUNAbleTHread2 implements Runnable{
    @Override               // overiding the method so it run freely
    public void run() {
        int i=0;
        while(i<5) {
            System.out.println("the thread of class 2 is running ");
            System.out.println(i+1);
            i++;
        }}}
public class Thread_using_runable_interface {
    public static void main(String[] args) {
//      craeting of the first Runnable interface object
        MyRUNAbleTHread1 sprint1 =new MyRUNAbleTHread1();
//      decalaring the object into thread
        Thread t1=new Thread(sprint1);
//      craeting of the second Runnable interface object
        MyRUNAbleTHread2 sprint2 =new MyRUNAbleTHread2();
//      decalaring the object into thread
        Thread t2=new Thread(sprint2);
        t1.start();
        t2.start();
    }}
```

Output:-

```
the thread of class 1 is running
0
the thread of class 1 is running
1
the thread of class 1 is running
2
the thread of class 1 is running
3
the thread of class 1 is running
4
the thread of class 2 is running
1
the thread of class 2 is running
2
the thread of class 2 is running
3
the thread of class 2 is running
4
the thread of class 2 is running
5
```

- We have to make a object of the runnable class , then declared the object as the thread the using the **start()** method we access the both the threads of the runnable interface simultaneously .

- ***Java Thread Life Cycle:-***



In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

1). New - Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Ex:- Instance of thread created which is not yet started by involving start(). In this state, the thread is also known as the born thread.

2). Active: -When a thread invokes the **start()** method, it moves from the new state to the active state. The active state contains two states within it: one is runnable, and the other is running.

3). Runnable - A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.

Ex:-After invocation of start()& before it is selected to be run by the scheduler.

4). Running - When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable. After the thread scheduler has selected it.

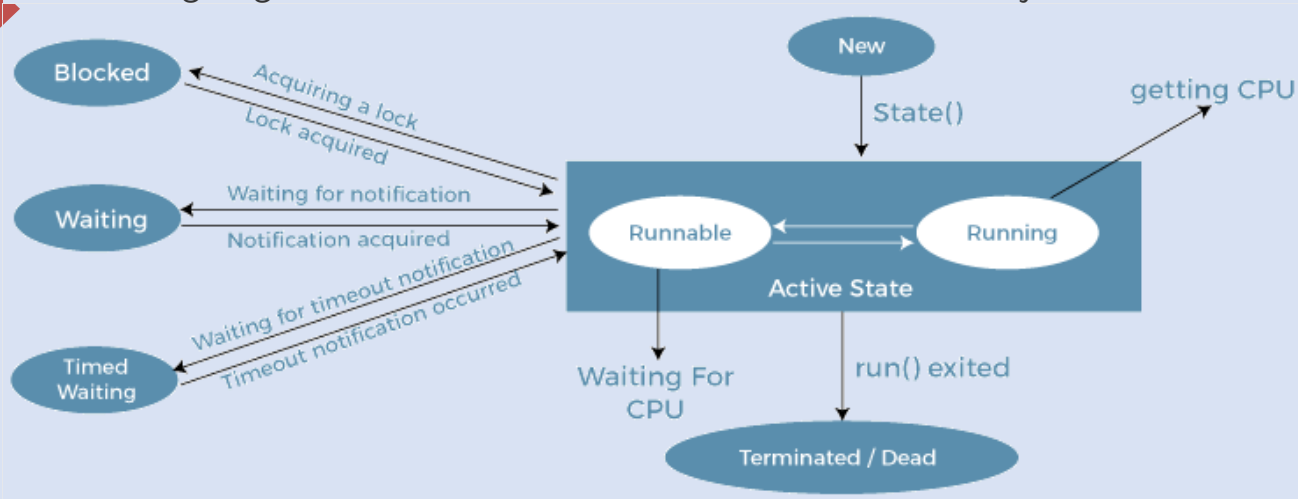
Blocked or Waiting: -Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

5).Non-runnable - thread alive, not eligible to run.

6). Terminated - run() method has exited.

The following diagram shows the different states involved in the life cycle of a thread.



Life Cycle of a Thread

Below code is show that the each states in the multithreading:-

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

- ***Constructors from Thread class in Java:-***
- ***The Thread class :-***

Below are the commonly used constructors of the thread class:

1. Thread ( )
2. Thread (string)
3. Thread (Runnable r)
4. Thread (Runnable r, String name)

Creating a thread Constructor which take some input from the user we using the super() ,super keyword to immediating the parent class .
We gonna perform some method which is mainly given by the Thread .
For Ex:- getId()

- ***Thread Scheduler in Java***

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

Priority: Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

Time of Arrival: Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

Thread Scheduler Algorithms

On the basis of the above-mentioned factors, the scheduling algorithm is followed by a Java thread scheduler.

First Come First Serve Scheduling:

In this scheduling algorithm, the scheduler picks the threads thar arrive first in the runnable queue. Observe the following table:

Threads	Time of Arrival
t1	0
t2	1
t3	2
t4	3

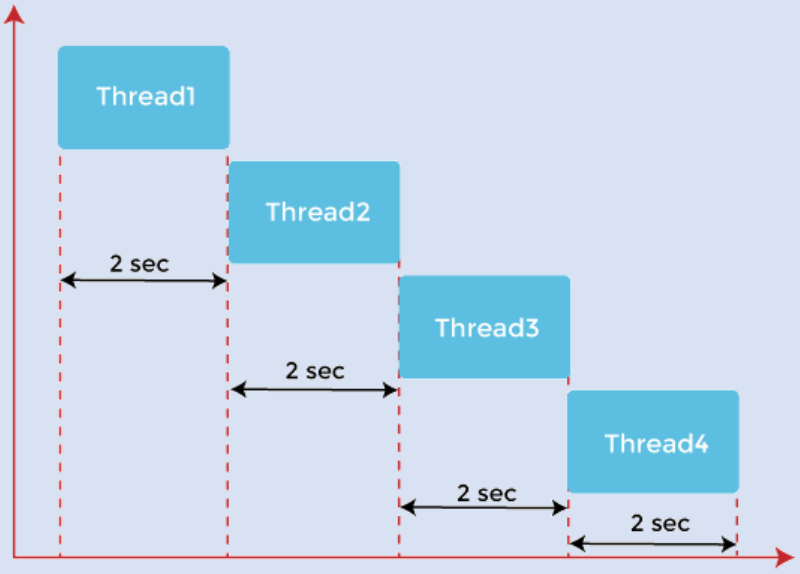
In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time of arrival of threads.

First Come First Serve Scheduling

Hence, Thread t1 will be processed first, and Thread t4 will be processed last.

Time-slicing scheduling:

Usually, the **First Come First Serve algorithm** is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.

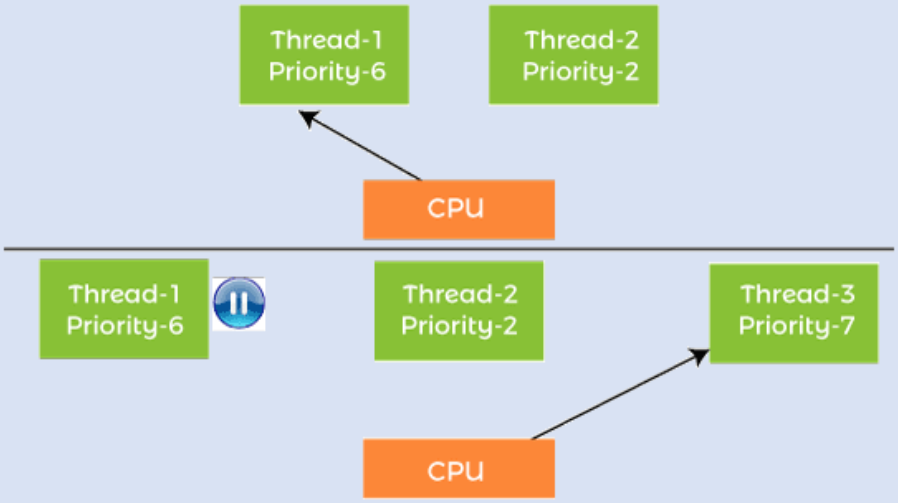


Time slicing scheduling

In the above diagram, each thread is given a time slice of 2 seconds. Thus, after 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2. The same process repeats for the other threads too.

- ***Preemptive-Priority Scheduling:***

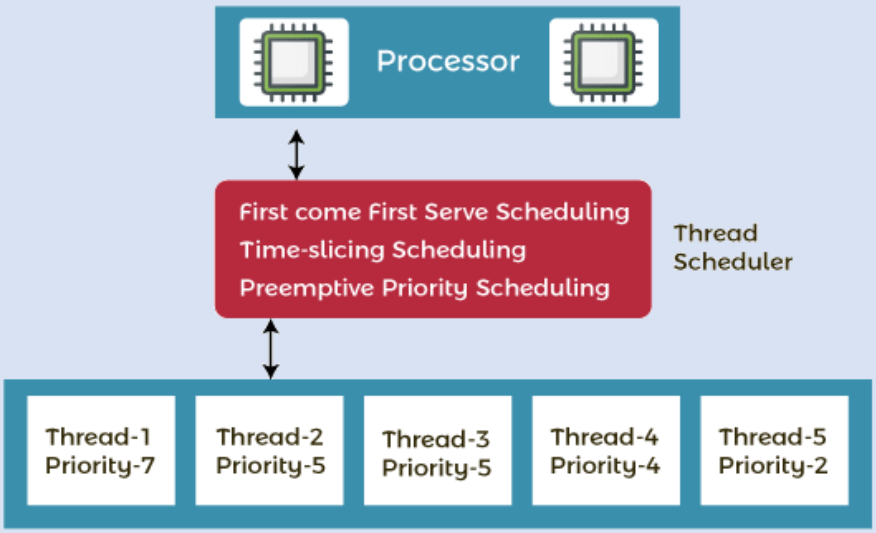
The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.



Preemptive-Priority Scheduling

Suppose there are multiple threads available in the runnable state. The thread scheduler picks that thread that has the highest priority. Since the algorithm is also preemptive, therefore, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it has to release the CPU because of preemption.

- ***Working of the Java Thread Scheduler***



Working of Thread Scheduler

Let's understand the working of the Java thread scheduler. Suppose, there are five threads that have different arrival times and different priorities. Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.

The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.

When two threads (Thread 2 and Thread 3) having the same priorities and arrival time, the scheduling will be decided on the basis of **FCFS algorithm**. Thus, the thread that arrives first gets the opportunity to execute first.

- ***Thread.sleep() in Java:-***

The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

- ***The sleep() Method Syntax:***

Following are the syntax of the **sleep()** method.

1. **public static void sleep(long mls) throws InterruptedException**
2. **public static void sleep(long mls, int n) throws InterruptedException**

- The method **sleep()** with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language.
- The other methods having the two parameters are not the native method. That is, its implementation is accomplished in Java. We can access the **sleep()** methods with the help of the Thread class, as the signature of the **sleep()** methods contain the static keyword.
- The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.
- The **Thread.sleep()** method can be used with any thread. It means any other thread or the main thread can invoke the **sleep()** method.

Parameters:

The following are the parameters used in the **sleep()** method.

- **mls:** The time in milliseconds is represented by the parameter **mls**. The duration for which the thread will sleep is given by the method **sleep().**

- **n:** It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

The method does not return anything.

- ***Important Points to Remember About the Sleep() Method:-***
- Whenever the **Thread.sleep()** methods execute, it always halts the execution of the current thread.
- Whenever another thread does interruption while the current thread is already in the sleep mode, then the **InterruptedException** is thrown.
- If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the **sleep()** method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

ThreadState.java | Sleeping_in_Thread.java

```
1 class SleepThread extends Thread{
2   public void run() {
3       int n=5;
4       for(int i=1;i<n;i++) {
5           try {
6 // Each Threads waits for 1sec = 100msec
7               Thread.sleep(1000);
8           }catch(Exception e) {
9               System.out.println(e);   }
10          System.out.println(i);   }
11   }
12 }
15 public class Sleeping_in_Thread {
16    public static void main(String[] args) {
17        // TODO Auto-generated method stub
18        SleepThread t1=new SleepThread();
19        SleepThread t2=new SleepThread();
20        t1.start();// after this Thread is on sleep
21 // its picks the t2 Thread and after t2 goes Sleep its
22  //  pick again t1 and this  goes on until all the Thread is printed
23        t2.start();
24    }
```

Console

<terminated> Sle

```
1
1
2
2
3
3
4
4
```

- In this the Thread t1 is starts and print 1 and the Thread t1 is going to sleep for 1000milisec
- Due to this the Thread Schedule picks the Thread t2 and t2 is invoked by the start method and print the 1
- And then Thread t2 goes to sleep for 1000msec , Then again the Thread Schedule picks the Thread t1 and print the next number 2
- And This is process continues until all the number is printed:-
- And we can see in the output that the number printed is 1 ,1,2,2,3,3,4,4 .
- Because when t1 ,t2 Thread runs then the Thread is in sleep for 1sec and the number printed like 1,1 and 2,2 are printed in the 1sec gap.
- ***If we give the Negative time to the Thread than its give the error:-***

"java.lang.IllegalArgumentException: timeout value is negative"

Code:-

```java
class NegativeSleepThread extends Thread{
      public void run() {
          int n=5;
          for(int i=1;i<n;i++) {
              try {
    // Giving the -ve time of the Thread so its gives the error
                  Thread.sleep(-500);
              }catch(Exception e1) {
                  System.out.println(e1);   }
              System.out.println(i);    }
      }
    }
public class Sleeping_in_Thread {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        SleepThread t1=new SleepThread();
        SleepThread t2=new SleepThread();
        t1.start();// after this Thread is on sleep
// its picks the t2 Thread and after t2 goes Sleep its
 //  pick again t1 and this  goes on until all the Thread is printed
        t2.start();
// Making the Object of the NegativeSleepThread class
        NegativeSleepThread t3=new NegativeSleepThread();
        t3.start();
    }
}
```

```
Console ×
<terminated> Sleeping_in_Thread [Java Application] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe  (Apr 12, 2023, 6:33:31 PM –
java.lang.IllegalArgumentException: timeout value is negative
1
java.lang.IllegalArgumentException: timeout value is negative
2
java.lang.IllegalArgumentException: timeout value is negative
3
java.lang.IllegalArgumentException: timeout value is negative
4
1
1
2
2
3
3
4
4
```

- ***Can we start a thread twice:-***

No. After starting a thread, it can never be started again. If you does so, an **IllegalThreadStateException** is thrown. In such case, thread will run once but for second time, it will throw exception.

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

Its gives the Error but we get the output:-

```
Console ×
<terminated> Sleeping_in_Thread [Java Application] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe  (Apr 12, 2023, 6:39:44 PM – 6
Exception in thread "main" java.
        at java.base/java.lang.Thread.start(Thread.java:793)
        at Sleeping_in_Thread.main(Sleeping_in_Thread.java:33)
1
2
3
4
```

- ***What if we call Java run() method directly instead start() method?***

- Each **thread** starts in a separate **call stack**.
- Invoking the **run()** method from the main thread, the **run()** method goes onto the current call stack rather than at the beginning of a **new call stack**.

As we can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

```
ThreadState.java   Sleeping_in_Thread.java ×
 1 class SleepThread extends Thread{
 2●  public void run() {
 3       int n=5;
 4       for(int i=1;i<n;i++) {
 5           try {
 6 // Each Threads waits for 1sec = 100msec
 7               Thread.sleep(1000);
 8           }catch(Exception e) {
 9               System.out.println(e);   }
10           System.out.println(i);   }
11   }
12 }
27 public class Sleeping_in_Thread {
28●    public static void main(String[] args) {
29         // TODO Auto-generated method stub
30         SleepThread t1=new SleepThread();
31         SleepThread t2=new SleepThread();
 // using the direct run method
         t1.run();
         t2.run();
     }
 }
```

```
Console ×
<terminated>
1
2
3
4
1
2
3
4
```

As we can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

In this the t1 and t2 is Treated as object which run the method name run() , it will not Treats as Thread .

So in this t1.run() compiles first and gives the output then the t2.run() compiles and gives the output.

- ***Java join() method:-***

The **join()** method in Java is provided by the **java.lang.Thread** class that permits one thread to wait until the other thread to finish its execution. Suppose ***th*** be the object the class Thread whose thread is doing its execution currently, then the ***th.join();*** statement ensures that ***th*** is finished before the program does the execution of the next statement. When there are more than one thread invoking the **join()** method, then it leads to overloading on the **join()** method that permits the developer or programmer to mention the waiting period. However, similar to the **sleep()** method in

Java, the **join()** method is also dependent on the operating system for the timing, so we should not assume that the **join()** method waits equal to the time we mention in the parameters. The following are the three overloaded **join()** methods.

- ***Description of The Overloaded join() Method:-***

join(): When the **join()** method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the **join()** method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the **InterruptedException.**

Syntax:

1. **public final void join() throws InterruptedException**

join(long mls, int nanos): When the **join()** method is invoked, the current thread stops its execution and go into the wait state. The current thread remains in the wait state until the thread on which the **join()** method is invoked called is dead or the wait for the specified time frame(in milliseconds + nanos) is over.

Syntax:

1. **public final synchronized void join(long mls, int nanos) throws InterruptedException**, w here mls is in milliseconds.

Code:-

ThreadState.java | Sleeping_in_Thread.java | Join_method_MultiThreading.java

```
class Mythread extends Thread{
    public void run() {
        for(int i=1;i<5;i++) {
            try{
                Thread.sleep(300);
    // Its gives the number of Thread is Executed or in Execution
         System.out.println("The current State of the Thread is "+Thread.currentThread().getName());
            }catch(Exception e) {
                System.out.println("The Exception is caught as : "+e);
            }
            System.out.println(i+"\t");
        }
    }
}
public class Join_method_MultiThreading {
    public static void main(String[] args) {
        // Creating 3 threads
   Mythread t1=new Mythread();
   Mythread t2=new Mythread();
   Mythread t3=new Mythread();
// starts a t1 Thread
   t1.start();
    try{
        // starting the second thread after when
        // the first thread th1 has ended or died.
t1.join();
// Below line of code gives the output of the Thread declared is as main Thread
System.out.println("The current State of the Thread is "+Thread.currentThread().getName());
    }catch(Exception e) {
        System.out.println("The Exception is caught as : "+e);

    }
    // starts a t2 Thread
    t2.start();
        try{
            // starting the Third thread after when
            // the Second thread th1 has ended or died.
```

```
    t2.join();
    // Below line of code gives the output of the Thread declared is as main Thread
    System.out.println("The current State of the Thread is "+Thread.currentThread().getName());
        }catch(Exception e) {
            System.out.println("The Exception is caught as : "+e);

        }
        // starts a t3 Thread
            t3.start();
}}
```

```
Console ×
<terminated> Join_method_MultiThreading [Java Application] C:\Program Files\Java\jdk-18.0.
The current State of the Thread is Thread-0
1
The current State of the Thread is Thread-0
2
The current State of the Thread is Thread-0
3
The current State of the Thread is Thread-0
4
The current State of the Thread is main
The current State of the Thread is Thread-1
1
The current State of the Thread is Thread-1
2
The current State of the Thread is Thread-1
3
The current State of the Thread is Thread-1
4
The current State of the Thread is main
The current State of the Thread is Thread-2
1
The current State of the Thread is Thread-2
2
The current State of the Thread is Thread-2
3
The current State of the Thread is Thread-2
4
```

Output:-

Another Example:-

ThreadState.java | Sleeping_in_Thread.java | Join_method_MultiThreading.java ×

```
class Mythread extends Thread{
    public void run() {
        for(int i=1;i<5;i++) {
            try{
                Thread.sleep(300);
            }catch(Exception e) {
                System.out.println("The Exception is caught as : "+e);
            }
            System.out.print(i+"\t");
        }
    }
}
public class Join_method_MultiThreading {
    public static void main(String[] args) {
        // Creating 3 threads
   Mythread t1=new Mythread();
   Mythread t2=new Mythread();
   Mythread t3=new Mythread();
// starts a t1 Thread
   t1.start();
    try{
        // starting the second thread after when
        // the first thread th1 has ended or died.
t1.join();
    }catch(Exception e) {
        System.out.println("The Exception is caught as : "+e);

    }
    // starts a t2 Thread
    t2.start();
            t3.start();
}}
```

Output:-

```
Console ×
<terminated> Join_method_MultiThreading [Java Application] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe  (Apr 15, 2023, 6:22:57 PM – 6:23:00 PM) [pid: 12360]
1	2	3	4	1	1	2	2	3	3	4	4
```

- ***Naming Thread and Current Thread:-***

- ***Naming Thread:-***

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name, i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using the setName() method. The syntax of **setName()** and **getName()** methods are given below:

1. **public String getName():** is used to **return** the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

We can also set the name of a thread directly when we create a new thread using the constructor of the class.

Code:-

ThreadState.java | Sleeping_in_Thread.java | Join_method_MultiThreading.java | Naming_Thread.java ×

```java
class TestMultiNaming extends Thread{
    public void run() {
        for(int i=1;i<5;i++) {
            try{
                // Thread Prints Between 1s gaps
Thread.sleep(1000);
            }catch(Exception e) {
                System.out.println("The Exception is caught as : "+e);
            }
            System.out.print(i+"\t");
        }
    }
}
public class Naming_Thread {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        TestMultiNaming t1=new TestMultiNaming();
        TestMultiNaming t2=new TestMultiNaming();
        TestMultiNaming t3=new TestMultiNaming();
          System.out.println("Name of t1:"+t1.getName());
          System.out.println("Name of t2:"+t2.getName());
          System.out.println("Name of t1:"+t3.getName());
          t1.start();
          t2.start();
          t3.start();
          t1.setName("Uday Sharma");
          t2.setName("Anshul Lakher");
          t3.setName("Yashas Gupta");
          System.out.println("After changing name of t1:"+t1.getName());
          System.out.println("After changing name of t2:"+t2.getName());
          System.out.println("After changing name of t3:"+t3.getName());
    }
}
```

Output:-

```
Console ×
<terminated> Naming_Thread [Java Application] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe  (Apr 15, 2023, 6:38:08 PM – 6:38:13 PM)
Name of t1:Thread-0
Name of t2:Thread-1
Name of t1:Thread-2
After changing name of t1:Uday Sharma
After changing name of t2:Anshul Lakher
After changing name of t3:Yashas Gupta
1       1       1       2       2       2       3       3       3       4       4       4
```

- ***: Without Using setName() Method:-***

One can also set the name of a thread at the time of the creation of a thread, without using the **setName()** method. Observe the following code.

Note:- **For this we can use the constructors.**

- ***Priority of a Thread (Thread Priority):-***

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

- ***Setter & Getter Method of Thread Priority:-***

Let's discuss the **setter and getter method** of the thread priority.

public final int getPriority(): The **java.lang.Thread.getPriority()** method returns the priority of the given thread.

public final void setPriority(int newPriority): The **java.lang.Thread.setPriority()** method updates or assign the priority of the thread to **newPriority**. The method throws **IllegalArgumentException** if the value **newPriority** goes out of the range, which is 1 (minimum) to 10 (maximum).

- ***3 constants defined in Thread class:***

1. **public static int MIN_PRIORITY**
2. **public static int NORM_PRIORITY**
3. **public static int MAX_PRIORITY**

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Code:-

```
1  class TestMultiPriority extends Thread{
2      public void run() {
3          for(int i=1;i<5;i++) {
4              try{
5                  // Thread Prints Between 1s gaps
6  Thread.sleep(1000);
7  // This help us to know get the current state of the Thread is running
8  System.out.println("The current State of the Thread is "+Thread.currentThread().getName());
9              }catch(Exception e) {
10                 System.out.println("The Exception is caught as : "+e);
11             }
12             System.out.print(i+"\t");   }
13     }
14 }
15 public class Priority_thread {
16     public static void main(String[] args) {
17         TestMultiPriority t1=new TestMultiPriority();
18         TestMultiPriority t2=new TestMultiPriority();
19         TestMultiPriority t3=new TestMultiPriority();
```

```
20          System.out.println("Priority of the thread t1 is : " + t1.getPriority());
21          System.out.println("Priority of the thread t2 is : " + t2.getPriority());
22          System.out.println("Priority of the thread t3 is : " + t3.getPriority());
23          // Setting the priority of the Thread
24          t1.setPriority(9);
25          t2.setPriority(4);
26          t3.setPriority(7);
27          System.out.println("Priority of the thread t1 is : " + t1.getPriority());
28          System.out.println("Priority of the thread t2 is : " + t2.getPriority());
29          System.out.println("Priority of the thread t3 is : " + t3.getPriority());
30          t1.start();
31          t2.start();
32          t3.start();
33      }
34  }
```

Output:-

```
Console ×
<terminated> Priority_thread [Java Application] C:\Program Files\Java\jdk-18.0.
Priority of the thread t1 is : 5
Priority of the thread t2 is : 5
Priority of the thread t3 is : 5
Priority of the thread t1 is : 9
Priority of the thread t2 is : 4
Priority of the thread t3 is : 7
The current State of the Thread is Thread-0
1       The current State of the Thread is Thread-2
1       The current State of the Thread is Thread-1
1       The current State of the Thread is Thread-0
2       The current State of the Thread is Thread-2
2       The current State of the Thread is Thread-1
2       The current State of the Thread is Thread-0
3       The current State of the Thread is Thread-2
3       The current State of the Thread is Thread-1
3       The current State of the Thread is Thread-0
4       The current State of the Thread is Thread-2
4       The current State of the Thread is Thread-1
4
```

- As we can see in the output the Thread-0(which is t1 Thread) has the Highest priority and the Thread-1 (which is t2 Thread) has the lowest priority.
- So we can see the Highest priority Thread run first then the Other priority Thread
- In the output the Thread-0 is highest, then Thread-2 then Thread-1 .
- We can see that Thread-0 runs first then Thread-2 then Thread-1 . and takes 1000milisec sleep then run the other part of its Thread.

  Note:-If you set the priority Greater than 10 its gives the error on a console .

• *Daemon Thread in Java:-*

Daemon thread in Java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

- ***Points to remember for Daemon Thread in Java:-***
  - It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
  - Its life depends on user threads.
  - It is a low priority thread.
- ***Why JVM terminates the daemon thread if there is no user thread?***

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

- ***Methods for Java Daemon thread by Thread class:-***

The class provides two methods for java daemon thread.

No.	Method	Description
1)	**public void setDaemon(boolean status)**	is used to mark the current thread as daemon thread or user thread.
2)	**public boolean isDaemon()**	is used to check that current is daemon.

Code:-

```
ThreadState.java  Sleeping_in_Thread.java  Join_method_MultiThreading.java  Naming_Thread.java  Priority_thr
 1 class DeamonMultiThreading extends Thread{
 2     public void run() {
 3             if(Thread.currentThread().isDaemon()){//checking for daemon thread
 4                   System.out.println("daemon thread work");
 5                 }
 6                 else{
 7                 System.out.println("user thread work");
 8                 }
 9 }}
10 public class Daemon_Thread {
11     public static void main(String[] args) {
12         DeamonMultiThreading t1=new DeamonMultiThreading();
13         DeamonMultiThreading t2=new DeamonMultiThreading();
14         DeamonMultiThreading t3=new DeamonMultiThreading();
15         t1.setDaemon(true);//now t1 is daemon thread
16           t1.start();//starting threads
17           t2.start();
18           t3.start();
19     }
20 }
21
```

Output:-

```
Console ×
<terminated> Daemon_Threa
daemon thread work
user thread work
user thread work
```

- ***Java Thread Pool:-***

Java Thread pool represents a group of worker threads that are waiting for the job and reused many times.

In the case of a thread pool, a group of fixed-size threads is created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, the thread is contained in the thread pool again.

- ***Thread Pool Methods:-***
- **newFixedThreadPool(int s):** The method creates a thread pool of the fixed size s.
- **newCachedThreadPool():** The method creates a new thread pool that creates the new threads when needed but will still use the previously created thread whenever they are available to use.
- **newSingleThreadExecutor():** The method creates a new thread.
- ***Advantage of Java Thread Pool:-***

Better performance It saves time because there is no need to create a new thread.

- ***Real time usage:-***

It is used in Servlet and JSP where the container creates a thread pool to process the request.

Code:-

ThreadState.java | Sleeping_in_Thread.java | Join_method_MultiThreading.java | Naming_Thread.java | Priority_thread.java | Daemon_Thread.java | *Thread_pool_E

```
// important import statements
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.text.SimpleDateFormat;
class WorkAssign implements Runnable {
    private String workassign;
// constructor of the class Tasks
    public WorkAssign(String str) {
// initializing the field taskName
        workassign = str;
    }
// Printing the task name and then sleeps for 1 sec
// The complete process is getting repeated five times
    public void run() {
        try {
            for (int j = 0; j <= 5; j++) {
                if (j == 0) {
                    // Creating a object of the Date package
                    Date dt = new Date();
//Creating a object of the SimpleDateFormat package
                    SimpleDateFormat sdf = new SimpleDateFormat("hh : mm : ss");
//prints the initialization time for every task
                    System.out.println("Initialization time for the task name: " + workassign + " = " + sdf.format(dt));
                } else {
                    Date dt = new Date();
                    SimpleDateFormat sdf = new SimpleDateFormat("hh : mm : ss");
// prints the execution time for every task
                    System.out.println("Time of execution for the task name: " + workassign + " = " + sdf.format(dt));
                }
// 1000ms = 1 sec
                Thread.sleep(1000);
            }
            System.out.println(workassign + " is complete.");
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
}
public class Thread_pool_Example {
// Maximum number of threads in the thread pool
    static final int MAX_TH = 3;
// main method
    public static void main(String argvs[]) {
// Creating five new tasks
        Runnable rb1 = new WorkAssign("WorkAssign 1");
        Runnable rb2 = new WorkAssign("WorkAssign 2");
        Runnable rb3 = new WorkAssign("WorkAssign 3");
        Runnable rb4 = new WorkAssign("WorkAssign 4");
        Runnable rb5 = new WorkAssign("WorkAssign 5");
```

```
// creating a thread pool with MAX_TH number of
// threads size the pool size is fixed
        ExecutorService pl = Executors.newFixedThreadPool(MAX_TH);
// passes the Task objects to the pool to execute (Step 3)
        pl.execute(rb1);
        pl.execute(rb2);
        pl.execute(rb3);
        pl.execute(rb4);
        pl.execute(rb5);
// pool is shutdown
        pl.shutdown();
    }
}
```

Output:-

Explanation: It is evident by looking at the output of the program that tasks 4 and 5 are executed only when the thread has an idle thread. Until then, the extra tasks are put in the queue.

The takeaway from the above example is when one wants to execute 50 tasks but is not willing to create 50 threads. In such a case, one can create a pool of 10 threads. Thus, 10 out of 50 tasks are assigned, and the rest are put in the queue. Whenever any thread out of 10 threads becomes idle, it picks up the 11th task. The other pending tasks are treated the same way.

- ***Risks involved in Thread Pools:-***

The following are the risk involved in the thread pools.

- ***Deadlock:*** It is a known fact that deadlock can come in any program that involves multithreading, and a thread pool introduces another scenario of deadlock. Consider a scenario where all the threads that are executing are waiting for the results from the threads that are blocked and waiting in the queue because of the non-availability of threads for the execution.
- ***Thread Leakage:*** Leakage of threads occurs when a thread is being removed from the pool to execute a task but is not returning to it after the completion of the task. For example, when a thread throws the exception and the pool class is not able to catch this exception, then the thread exits and reduces the thread pool size by 1. If the same thing repeats a number of times, then there are fair chances that the pool will become empty, and hence, there are no threads available in the pool for executing other requests.
- ***Resource Thrashing:*** A lot of time is wasted in context switching among threads when the size of the thread pool is very large. Whenever there are more threads than the optimal number may cause the starvation problem, and it leads to resource thrashing.

- ***Points to Remember:-***

Do not queue the tasks that are concurrently waiting for the results obtained from the other tasks. It may lead to a deadlock situation, as explained above.

Care must be taken whenever threads are used for the operation that is long-lived. It may result in the waiting of thread forever and will finally lead to the leakage of the resource.

In the end, the thread pool has to be ended explicitly. If it does not happen, then the program continues to execute, and it never ends. Invoke the shutdown() method on the thread pool to terminate the executor. Note that if someone tries to send another task to the executor after shutdown, it will throw a RejectedExecutionException.

One needs to understand the tasks to effectively tune the thread pool. If the given tasks are contrasting, then one should look for pools for executing different varieties of tasks so that one can properly tune them.

To reduce the probability of running JVM out of memory, one can control the maximum threads that can run in JVM. The thread pool cannot create new threads after it has reached the maximum limit.

A thread pool can use the same used thread if the thread has finished its execution. Thus, the time and resources used for the creation of a new thread are saved.

- ***Tuning the Thread Pool:-***

The accurate size of a thread pool is decided by the number of available processors and the type of tasks the threads have to execute. If a system has the P processors that have only got the computation type processes, then the maximum size of the thread pool of P or P + 1 achieves the maximum efficiency. However, the tasks may have to wait for I/O, and in such a scenario, one has to take into consideration the ratio of the waiting time (W) and the service time (S) for the request; resulting in the maximum size of the pool P * (1 + W / S) for the maximum efficiency.

- ***Conclusion:-***

A thread pool is a very handy tool for organizing applications, especially on the server-side. Concept-wise, a thread pool is very easy to comprehend. However, one may have to look at a lot of issues when dealing with a thread pool. It is because the thread pool comes with some risks involved it (risks are discussed above).

- ***ThreadGroup in Java:-***

- Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.

Note: Now suspend(), resume() and stop() methods are deprecated.

  - Java thread group is implemented by ***java.lang.ThreadGroup*** class.
- A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

- ***Constructors of ThreadGroup class:-***

There are only two constructors of **ThreadGroup** class.

No.	Constructor	Description
1)	**ThreadGroup(String name)**	creates a thread group with given name.
2)	**ThreadGroup(ThreadGroup parent, String name)**	creates a thread group with a given parent group and name.

- ***Methods of ThreadGroup class:-***

There are many methods in ThreadGroup class. A list of ThreadGroup methods is given below.

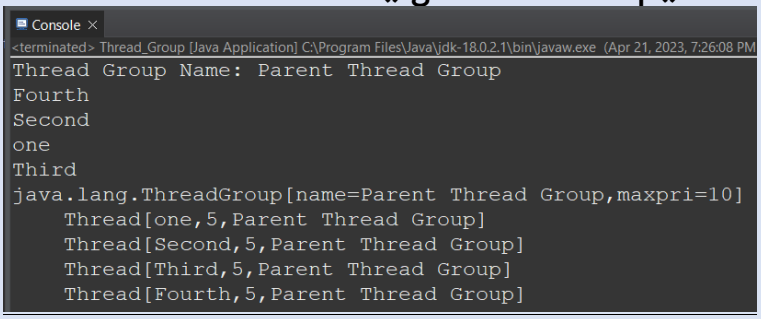
S.N.	Modifier and Type	Method	Description
1)	**void**	**checkAccess()**	This method determines if the currently running thread has permission to modify the thread group.
2)	**int**	**activeCount()**	This method returns an estimate of the number of active threads in the thread group and its subgroups.
3)	**int**	**activeGroupCount()**	This method returns an estimate of the number of active groups in the thread group and its subgroups.
4)	**void**	**destroy()**	This method destroys the thread group and all of its subgroups.

5)	**int**	**enumerate(Thread[] list)**	This method copies into the specified array every active thread in the thread group and its subgroups.
6)	**int**	**getMaxPriority()**	This method returns the maximum priority of the thread group.
7)	**String**	**getName()**	This method returns the name of the thread group.
8)	**ThreadGroup**	**getParent()**	This method returns the parent of the thread group.
9)	**void**	**interrupt()**	This method interrupts all threads in the thread group.
10)	**boolean**	**isDaemon()**	This method tests if the thread group is a daemon thread group.
11)	**void**	**setDaemon(boolean daemon)**	This method changes the daemon status of the thread group.
12)	**boolean**	**isDestroyed()**	This method tests if this thread group has been destroyed.
13)	**void**	**list()**	This method prints information about the thread group to the standard output.
14)	**boolean**	**parentOf(ThreadGroup g**	This method tests if the thread group is either the thread group argument or one of its ancestor thread groups.
15)	**void**	**suspend()**	This method is used to suspend all threads in the thread group.
16)	**void**	**resume()**	This method is used to resume all threads in the thread group which was suspended using suspend() method.
17)	**void**	**setMaxPriority(int pri)**	This method sets the maximum priority of the group.
18)	**void**	**stop()**	This method is used to stop all threads in the thread group.
19)	**String**	**toString()**	This method returns a string representation of the Thread group.

Code:- Simple Thread Group Example:-

Now all 4 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two","three"and “fourth” are the thread names. Now we can interrupt all threads by a single line of code only.

Thread.currentThread().getThreadGroup().interrupt();



```
public class Thread_Group implements Runnable{
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
  Thread_Group runnable=new Thread_Group();
  //Creating the object of the Thread group and name them
  ThreadGroup tg1=new ThreadGroup("Parent Thread Group");
  //Creating a Thread and put in the Group name tg1
    Thread t1=new Thread(tg1,runnable,"one");
    t1.start();
    Thread t2=new Thread(tg1,runnable,"Second");
    t2.start();
    Thread t3=new Thread(tg1,runnable,"Third");
    t3.start();
    Thread t4=new Thread(tg1,runnable,"Fourth");
    t4.start();
      System.out.println("Thread Group Name: "+tg1.getName());
      tg1.list();
    }
}
```

Example-2 :- ActiveGroupCount and ChildOfTHreadGroup:-

```
// For using the THread Group we have to import the package
import java.lang.ThreadGroup;
// Thread Group ActiveGroupCount Example
class TG_activeGroupCount extends Thread{
    TG_activeGroupCount (String tName, ThreadGroup tgrp)
    {
    super(tgrp, tName);
    start();
    }
    public void run() {
        for(int j=0;j<5;j++) {
            try {
                Thread.sleep(1000);
            }catch(Exception e) {System.out.println("The exception has been encountered " + e);  }
        }
    }
}
// Simple Thread Example
```

```
public class Thread_Group implements Runnable{
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
  Thread_Group runnable=new Thread_Group();
  //Creating the object of the Thread group and name them
  ThreadGroup tg1=new ThreadGroup("Parent Thread Group");
  //Creating a Thread and put in the Group name tg1
    Thread t1=new Thread(tg1,runnable,"one");
    t1.start();
    Thread t2=new Thread(tg1,runnable,"Second");
    t2.start();
    Thread t3=new Thread(tg1,runnable,"Third");
    t3.start();
    Thread t4=new Thread(tg1,runnable,"Fourth");
    t4.start();
      System.out.println("Thread Group Name: "+tg1.getName());
      tg1.list();
```

```
    // creating the a another thread group tg2
       ThreadGroup tg2 = new ThreadGroup("The parent group of threads");
       //Creating a Thread and put in the Group name tg2
       TG_activeGroupCount th1 = new TG_activeGroupCount("first", tg2);
       System.out.println("Starting the first");
       TG_activeGroupCount th2 = new TG_activeGroupCount("second", tg2);
       System.out.println("Starting the second");
       // checking the number of active thread by invoking the activeCount() method
       // tg2 Thread Group contains two Thread
       System.out.println("The total number of active threads are: " + tg2.activeCount());
     // Making child of the Thread tg2
       ThreadGroup childoftg2=new ThreadGroup(tg2,"Child of the THreadGroup-2(tg2)");
       TG_activeGroupCount childoftg2 th1 = new TG_activeGroupCount("first", childoftg2);
       System.out.println("The total number of active threads are: " + childoftg2.activeCount());
       System.out.println("Thread Group Name: "+childoftg2.getName());
    }
 }
```

Output:-

```
Console ×
Thread_Group [Java Application] C:\Program Files\Java\jdk-18.0.2.1\bin\javaw.exe  (Apr 21, 2023, 7:38:12 PM) [pid: 8108]
Thread Group Name: Parent Thread Group
Fourth
java.lang.ThreadGroup[name=Parent Thread Group,maxpri=10]
    Third
Second
one
Thread[one,5,Parent Thread Group]
    Thread[Second,5,Parent Thread Group]
    Thread[Third,5,Parent Thread Group]
    Thread[Fourth,5,Parent Thread Group]
Starting the first
Starting the second
The total number of active threads are: 2
The total number of active threads are: 1
Thread Group Name: Child of the THreadGroup-2(tg2)
```

And There are my other Method in the Thread Group of Multithreading in Java like:-

- **destroy()**
- **enumerate()**
- **getMaxPriority()**
- **ThreadGroup getParent()**
- **interrupt()**
- **isDaemon()**
- **isDestroyed()**

• ***Java Shutdown Hook:-***

- A special construct that facilitates the developers to add some code that has to be run when the **Java Virtual Machine (JVM)** is **shutting down** is known as the **Java shutdown hook**. The Java shutdown hook comes in very handy in the cases where one needs to perform some special **cleanup work** when the JVM is shutting down. Note that handling an operation such as invoking a special method before the JVM terminates does not work using a general construct when the JVM is shutting down due to some external factors. For example, whenever a kill request is generated by the operating system or due to resource is not allocated because of the lack of free memory, then in such a case, it is not possible to invoke the procedure. The shutdown hook solves this problem comfortably by providing an arbitrary block of code.
- Taking at a surface level, learning about the shutdown hook is straightforward. All one has to do is to derive a class using the **java.lang.Thread** class, and then provide the code for the task one

wants to do in the **run()** method when the JVM is going to shut down. For registering the instance of the derived class as the shutdown hook, one has to invoke the method **Runtime.getRuntime().addShutdownHook(Thread)**, whereas for removing the already registered shutdown hook, one has to invoke the **removeShutdownHook(Thread)** method.

In nutshell, the shutdown hook can be used to perform cleanup resources or save the state when JVM shuts down normally or abruptly. Performing clean resources means closing log files, sending some alerts, or something else. So if you want to execute some code before JVM shuts down, use the shutdown hook.

- ***When does the JVM shut down?***

The JVM shuts down when:

- user presses ctrl+c on the command prompt
- System.exit(int) method is invoked
- user logoff
- user shutdown etc.

- ***The addShutdownHook(Thread hook) method:-***

The **addShutdownHook()** method of the Runtime class is used to register the thread with the Virtual Machine.

Syntax:

1. **public void addShutdownHook(Thread hook){}**

The object of the Runtime class can be obtained by calling the static factory method **getRuntime()**. For example:

Runtime r = Runtime.getRuntime();

- ***The removeShutdownHook(Thread hook) method:-***

The **removeShutdownHook()** method of the Runtime class is invoked to remove the registration of the already registered shutdown hooks.

Syntax:

1. **public boolean removeShutdownHook(Thread hook){ }**

True value is returned by the method, when the method successfully de-register the registered hooks; otherwise returns false.

- ***Factory method:-***

The method that returns the instance of a class is known as factory method.

Code:-Simple example of Shutdown Hook:-

ThreadState.java | Sleeping_in_Thr... | Join_method_Mul... | Naming_Thread.ja... | Priority_thread... | Daemon_Thread.ja...

```
1  class ShutDown_EX1 extends Thread{
2      public void run(){
3          System.out.println("shut down hook task completed..");
4      }
5  }
6  public class ShutDown_hook {
7      public static void main(String[] args) {
8          // TODO Auto-generated method stub
9  // Create RunTime Object r
10 Runtime r=Runtime.getRuntime();
11 r.addShutdownHook(new ShutDown_EX1());
12 System.out.println("Now main sleeping... press ctrl+c to exit");
13 try{
14     System.out.println("The Thread wait For 5 sec then Shut Down");
15     Thread.sleep(5000);
16 }catch (Exception e) {System.out.println("the Execption is "+e);}
17     }
18 }
```

Output:-

```
Console ×
<terminated> ShutDown_hook [Java Application] C:\Program Files\Java\jdk-18.0.2.1\bin\
Now main sleeping... press ctrl+c to exit
The Thread wait For 5 sec then Shut Down
shut down hook task completed..
```

Java Garbage Collection

- In java, garbage means unreferenced objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
- To do so, we were using **free()** function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

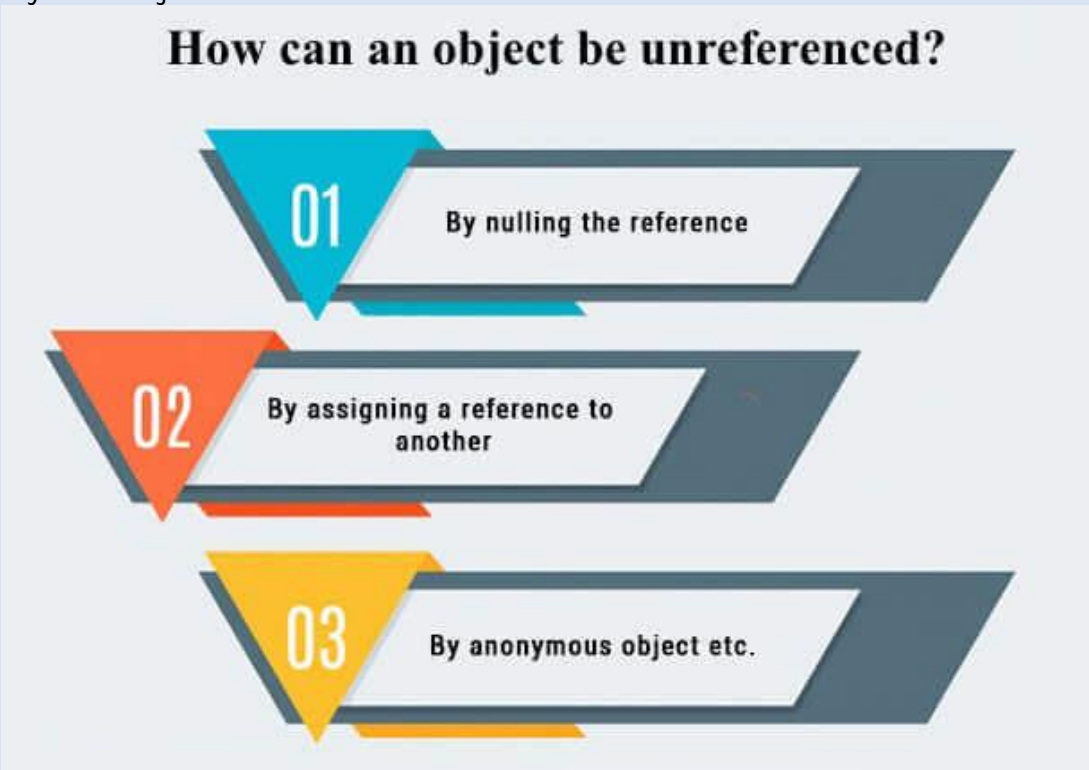
Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.



- ***By nulling a reference:***

```
1. Employee e=new Employee();
2. e=null;
```

2) By assigning a reference to another:-

```
1. Employee e1=new Employee();
2. Employee e2=new Employee();
```

3. e1=e2;//now the first object referred by e1 is available for garbage collection

3) By anonymous object:-

1. **new** Employee();

- ***finalize() method:-***

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

protected void finalize(){}

- **Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).**

gc() method:-

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

1. **public static void gc(){}**

- **Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.**

Code:-

```
1  public class Grabage_Collection {
2  protected void finalize() {
3      System.out.println("The Grabage is Collected!!!...");
4  }
5      public static void main(String[] args) {
6    Grabage_Collection gc1=new Grabage_Collection();
7    Grabage_Collection gc2 =new Grabage_Collection();
8    gc1=null;
9    gc2=null;
10 // gc() method run Twice Because The There is Two Object Which is Pointing NULL
11   System.gc();
12     }
13 }
```

Output:-

```
Console ×
<terminated> Grabage_Collection [Java Application] C:\Program F
The Grabage is Collected!!!. <term
The Grabage is Collected!!!...
```

- ***Java Runtime class***

Java Runtime class is used *to interact with java runtime environment*. Java Runtime class provides methods to execute a process, invoke **GC**, get total and free memory etc. There is only one instance of **java.lang.Runtime** class is available for one java application.

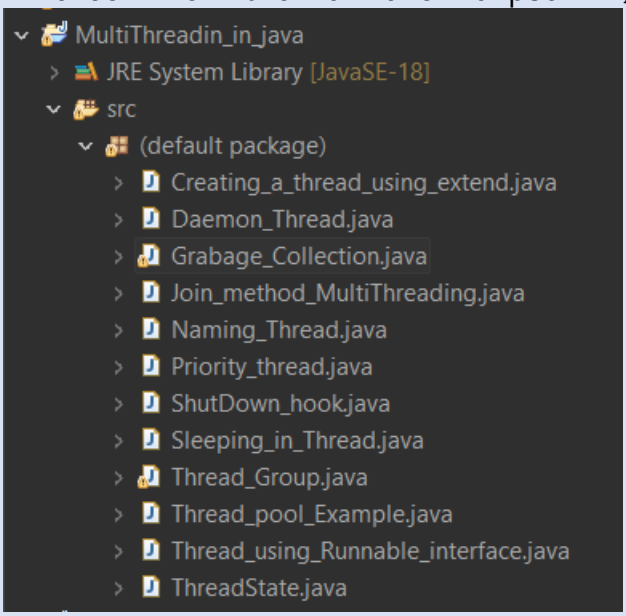
The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

- ***Important methods of Java Runtime class***

No.	Method	Description
1)	**public static Runtime getRuntime()**	returns the instance of Runtime class.
2)	**public void exit(int status)**	terminates the current virtual machine.

3)	**public void addShutdownHook(Thread hook)**	registers new hook thread.
4)	**public Process exec(String command)throws IOException**	executes given command in a separate process.
5)	**public int availableProcessors()**	returns no. of available processors.
6)	**public long freeMemory()**	returns amount of free memory in JVM.
7)	**public long totalMemory()**	returns amount of total memory in JVM.

This Topics Cover Are look like in the Manner of Eclipse IDE ,JDK-18 , JAVA EE.



- Go check out my ***LinkedIn profile*** for more notes and other resources content

@Uday Sharma
mrudaysharma4600@gmail.com
https://www.linkedin.com/in/uday-sharma-602b33267