Multithreading in Java

Multithreading in Java

is a process of executing multiple threads simultaneously.

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

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

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)

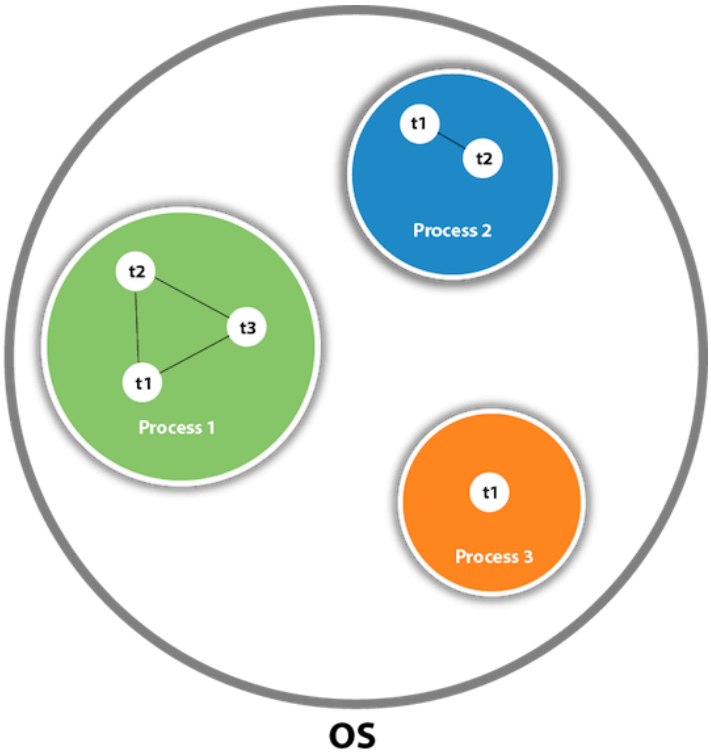
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

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.

Java Thread Methods

S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.

2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	suspend()	It is used to suspend the thread.
14)	void	resume()	It is used to resume the suspended thread.
15)	void	stop()	It is used to stop the thread.
16)	void	destroy()	It is used to destroy the thread group and all of its subgroups.
17)	boolean	isDaemon()	It tests if the thread is a daemon thread.
18)	void	setDaemon()	It marks the thread as daemon or user thread.
19)	void	interrupt()	It interrupts the thread.
20)	boolean	isinterrupted()	It tests whether the thread has been interrupted.
21)	static boolean	interrupted()	It tests whether the current thread has been interrupted.
22)	static int	activeCount()	It returns the number of active threads in the current thread's thread group.

23)	void	checkAccess()	It determines if the currently running thread has permission to modify the thread.
24)	static boolean	holdLock()	It returns true if and only if the current thread holds the monitor lock on the specified object.
25)	static void	dumpStack()	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	getStackTrace()	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	enumerate()	It is used to copy every active thread's thread group and its subgroup into the specified array.
28)	Thread.State	getState()	It is used to return the state of the thread.
29)	ThreadGroup	getThreadGroup()	It is used to return the thread group to which this thread belongs
30)	String	toString()	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.
31)	void	notify()	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	notifyAll()	It is used to give the notification to all waiting threads of a particular object.
33)	void	setContextClassLoader()	It sets the context ClassLoader for the Thread.
34)	ClassLoader	getContextClassLoader()	It returns the context ClassLoader for the thread.
35)	static Thread.UncaughtExceptionHandler	getDefaultUncaughtExceptionHandler()	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	setDefaultUncaughtExceptionHandler()	It sets the default handler invoked when a thread

			abruptly terminates due to an uncaught exception.

Do You Know

- How to perform two tasks by two threads?
- How to perform multithreading by anonymous class?
- What is the Thread Scheduler and what is the difference between preemptive scheduling and time slicing?
- What happens if we start a thread twice?
- What happens if we call the run() method instead of start() method?
- What is the purpose of join method?
- Why JVM terminates the daemon thread if no user threads are remaining?
- What is the shutdown hook?
- What is garbage collection?
- What is the purpose of finalize() method?
- What does the gc() method?
- What is synchronization and why use synchronization?
- What is the difference between synchronized method and synchronized block?
- What are the two ways to perform static synchronization?
- What is deadlock and when it can occur?
- What is interthread-communication or cooperation?

Life cycle of a Thread (Thread States)

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

Explanation of Different Thread States

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.

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**.

- **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.
  A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.

- **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.

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.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

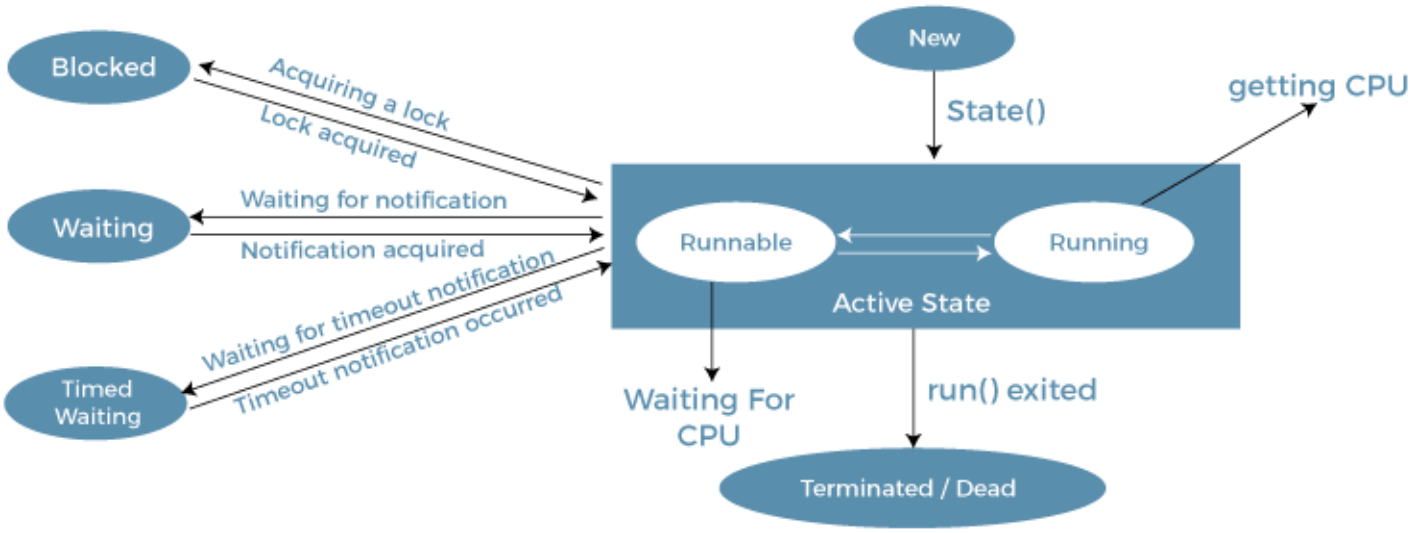
Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such 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.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

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



Life Cycle of a Thread

Implementation of Thread States

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

1. public static final Thread.State NEW

It represents the first state of a thread that is the NEW state.

1. public static final Thread.State RUNNABLE

It represents the runnable state.It means a thread is waiting in the queue to run.

1. public static final Thread.State BLOCKED

It represents the blocked state. In this state, the thread is waiting to acquire a lock.

1. public static final Thread.State WAITING

It represents the waiting state. A thread will go to this state when it invokes the Object.wait() method, or Thread.join() method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

1. public static final Thread.State TIMED_WAITING

It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.

- sleep
- join with timeout
- wait with timeout

- parkUntil
- parkNanos

1. public static final Thread.State TERMINATED

It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

Java Program for Demonstrating Thread States

The following Java program shows some of the states of a thread defined above.

FileName: ThreadState.java

```
1. // ABC class implements the interface Runnable
2. class ABC implements Runnable
3. {
4. public void run()
5. {
6.
7. // try-catch block
8. try
9. {
10. // moving thread t2 to the state timed waiting
11. Thread.sleep(100);
12. }
13. catch (InterruptedException ie)
14. {
15. ie.printStackTrace();
16. }
17.
18.
19. System.out.println("The state of thread t1 while it invoked the method join() on thread t2 -
    "+ ThreadState.t1.getState());
20.
21. // try-catch block
22. try
23. {
24. Thread.sleep(200);
25. }
26. catch (InterruptedException ie)
27. {
28. ie.printStackTrace();
29. }
30. }
31. }
32.
33. // ThreadState class implements the interface Runnable
34. public class ThreadState implements Runnable
35. {
```

```
36. public static Thread t1;
37. public static ThreadState obj;
38.
39. // main method
40. public static void main(String argvs[])
41. {
42. // creating an object of the class ThreadState
43. obj = new ThreadState();
44. t1 = new Thread(obj);
45.
46. // thread t1 is spawned
47. // The thread t1 is currently in the NEW state.
48. System.out.println("The state of thread t1 after spawning it - " + t1.getState());
49.
50. // invoking the start() method on
51. // the thread t1
52. t1.start();
53.
54. // thread t1 is moved to the Runnable state
55. System.out.println("The state of thread t1 after invoking the method start() on it - " + t1.getState());
56. }
57.
58. public void run()
59. {
60. ABC myObj = new ABC();
61. Thread t2 = new Thread(myObj);
62.
63. // thread t2 is created and is currently in the NEW state.
64. System.out.println("The state of thread t2 after spawning it - "+ t2.getState());
65. t2.start();
66.
67. // thread t2 is moved to the runnable state
68. System.out.println("the state of thread t2 after calling the method start() on it - " + t2.getState());
69.
70. // try-catch block for the smooth flow of the  program
71. try
72. {
73. // moving the thread t1 to the state timed waiting
74. Thread.sleep(200);
75. }
76. catch (InterruptedException ie)
77. {
78. ie.printStackTrace();
79. }
80.
81. System.out.println("The state of thread t2 after invoking the method sleep() on it - "+ t2.getState() );
82.
83. // try-catch block for the smooth flow of the  program
84. try
85. {
86. // waiting for thread t2 to complete its execution
```

```
87. t2.join();
88. }
89. catch (InterruptedException ie)
90. {
91. ie.printStackTrace();
92. }
93. System.out.println("The state of thread t2 when it has completed it's execution - " + t2.getState());
94. }
95.
96. }
```

Output:

```
The state of thread t1 after spawning it - NEW
The state of thread t1 after invoking the method start() on it - RUNNABLE
The state of thread t2 after spawning it - NEW
the state of thread t2 after calling the method start() on it - RUNNABLE
The state of thread t1 while it invoked the method join() on thread t2 -TIMED_WAITING
The state of thread t2 after invoking the method sleep() on it - TIMED_WAITING
The state of thread t2 when it has completed it's execution - TERMINATED
```

Explanation: Whenever we spawn a new thread, that thread attains the new state. When the method start() is invoked on a thread, the thread scheduler moves that thread to the runnable state. Whenever the join() method is invoked on any thread instance, the current thread executing that statement has to wait for this thread to finish its execution, i.e., move that thread to the terminated state. Therefore, before the final print statement is printed on the console, the program invokes the method join() on thread t2, making the thread t1 wait while the thread t2 finishes its execution and thus, the thread t2 get to the terminated or dead state. Thread t1 goes to the waiting state because it is waiting for thread t2 to finish it's execution as it has invoked the method join() on thread t2.

Java Threads | How to create a thread in Java

There are two ways to create a thread:

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

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

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.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

FileName: Multi.java

```
1. class Multi extends Thread{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5. public static void main(String args[]){
6. Multi t1=new Multi();
7. t1.start();
8.  }
```

```
}
```

Output:

```
thread is running...
```

2) Java Thread Example by implementing Runnable interface

FileName: Multi3.java

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
t1.start();
 }
}
```

Output:

```
thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

FileName: MyThread1.java

```
public class MyThread1
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
}
```

Output:

```
My first thread
```

4) Using the Thread Class: Thread(Runnable r, String name)

Observe the following program.

FileName: MyThread2.java

```
1. public class MyThread2 implements Runnable
2. {
3. public void run()
4. {
5. System.out.println("Now the thread is running ...");
6. }
7.
8. // main method
9. public static void main(String argvs[])
10. {
11. // creating an object of the class MyThread2
12. Runnable r1 = new MyThread2();
13.
14. // creating an object of the class Thread using Thread(Runnable r, String name)
15. Thread th1 = new Thread(r1, "My new thread");
16.
17. // the start() method moves the thread to the active state
18. th1.start();
19.
20. // getting the thread name by invoking the getName() method
21. String str = th1.getName();
22. System.out.println(str);
23. }
24. }
```

Output:

```
My new thread
Now the thread is running ...
```

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

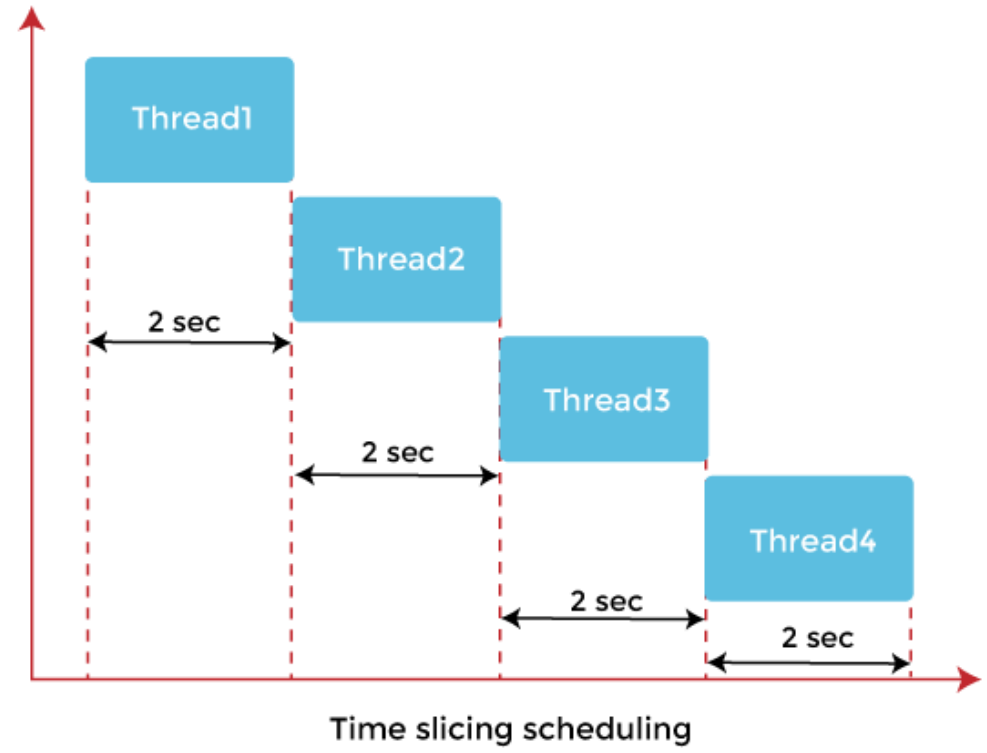
In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time of arrival of threads.

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.

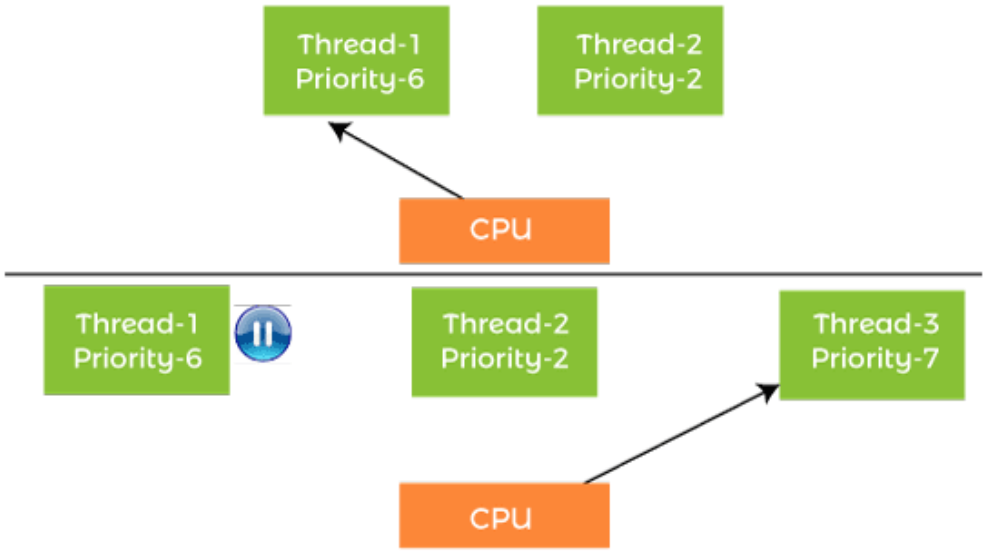


Time slicing scheduling

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:

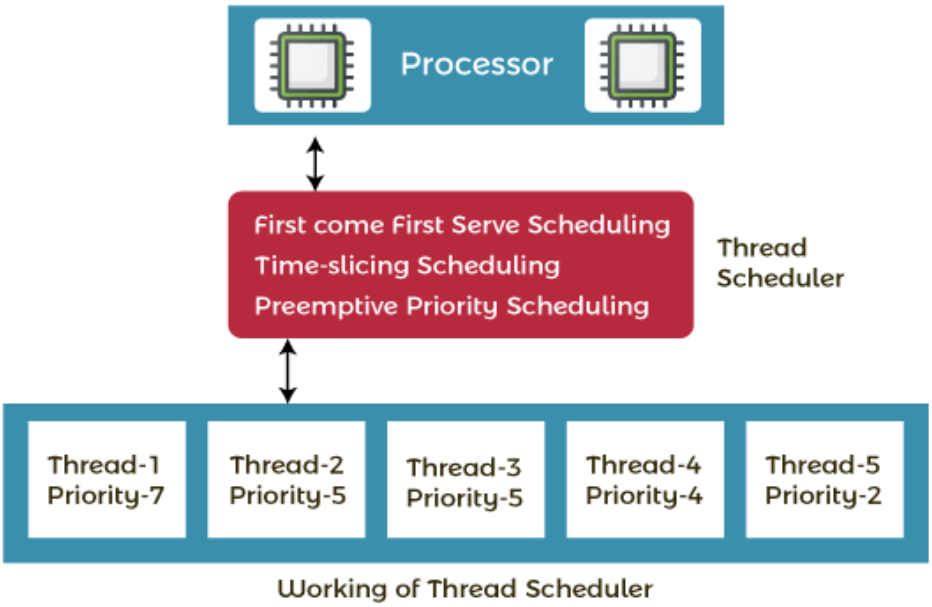
The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.



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 with Examples

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.

Example of the sleep() method in Java : on the custom thread

The following example shows how one can use the sleep() method on the custom thread.

FileName: TestSleepMethod1.java

```
1.  class TestSleepMethod1 extends Thread{
2.   public void run(){
3.    for(int i=1;i<5;i++){
4.    // the thread will sleep for the 500 milli seconds
5.      try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
6.      System.out.println(i);
7.    }
8.   }
9.   public static void main(String args[]){
10.   TestSleepMethod1 t1=new TestSleepMethod1();
11.   TestSleepMethod1 t2=new TestSleepMethod1();
12.
13.   t1.start();
14.   t2.start();
```

```
15.  }
16. }
```

Output:

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

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

Example of the sleep() Method in Java : on the main thread

FileName: TestSleepMethod2.java

```
1.  // important import statements
2.  import java.lang.Thread;
3.  import java.io.*;
4.
5.
6.  public class TestSleepMethod2
7.  {
8.      // main method
9.  public static void main(String argvs[])
10. {
11.
12. try {
13. for (int j = 0; j < 5; j++)
14. {
15.
16. // The main thread sleeps for the 1000 milliseconds, which is 1 sec
17. // whenever the loop runs
18. Thread.sleep(1000);
19.
20. // displaying the value of the variable
21. System.out.println(j);
22. }
23. }
24. catch (Exception expn)
25. {
26. // catching the exception
27. System.out.println(expn);
28. }
29. }
30. }
```

Output:

```
0
1
2
3
4
```

Example of the sleep() Method in Java: When the sleeping time is -ive

The following example throws the exception IllegalArguementException when the time for sleeping is negative.

FileName: TestSleepMethod3.java

```
1.  // important import statements
2.  import java.lang.Thread;
3.  import java.io.*;
4.
5.  public class TestSleepMethod3
6.  {
7.  // main method
8.  public static void main(String argvs[])
9.  {
10. // we can also use throws keyword followed by
11. // exception name for throwing the exception
12. try
13. {
14. for (int j = 0; j < 5; j++)
15. {
16.
17. // it throws the exception IllegalArgumentException
18. // as the time is -ive which is -100
19. Thread.sleep(-100);
20.
21. // displaying the variable's value
22. System.out.println(j);
23. }
24. }
25. catch (Exception expn)
26. {
27.
28. // the exception iscaught here
29. System.out.println(expn);
30. }
31. }
32. }
```

Output:

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

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.

Let's understand it by the example given below:

```
1. public class TestThreadTwice1 extends Thread{
2.  public void run(){
3.    System.out.println("running...");
4.  }
5.  public static void main(String args[]){
6.   TestThreadTwice1 t1=new TestThreadTwice1();
7.   t1.start();
8.   t1.start();
9.  }
10. }
```

Output:

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

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.

FileName: TestCallRun1.java

```
1. class TestCallRun1 extends Thread{
2.  public void run(){
3.    System.out.println("running...");
4.  }
5.  public static void main(String args[]){
6.   TestCallRun1 t1=new TestCallRun1();
7.   t1.run();//fine, but does not start a separate call stack
8.  }
9. }
```

Output:

```
running...
```

Problem if you direct call run() method

FileName: TestCallRun2.java

```
1.  class TestCallRun2 extends Thread{
2.   public void run(){
3.    for(int i=1;i<5;i++){
4.      try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
5.      System.out.println(i);
6.    }
7.   }
8.   public static void main(String args[]){
9.    TestCallRun2 t1=new TestCallRun2();
10.   TestCallRun2 t2=new TestCallRun2();
11.
12.   t1.run();
13.   t2.run();
14.  }
15. }
```

Output:

```
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.

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): 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 called is dead or the wait for the specified time frame(in milliseconds) is over.

Syntax:

1. public final synchronized void join(long mls) throws InterruptedException, where mls is in milliseconds

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, where mls is in mi lliseconds.

Example of join() Method in Java

The following program shows the usage of the join() method.

FileName: ThreadJoinExample.java

```
1. // A Java program for understanding
2. // the joining of threads
3.
4. // import statement
5. import java.io.*;
6.
7. // The ThreadJoin class is the child class of the class Thread
```

```
8.  class ThreadJoin extends Thread
9.  {
10. // overriding the run method
11. public void run()
12. {
13. for (int j = 0; j < 2; j++)
14. {
15. try
16. {
17. // sleeping the thread for 300 milli seconds
18. Thread.sleep(300);
19. System.out.println("The current thread name is: " + Thread.currentThread().getName());
20. }
21. // catch block for catching the raised exception
22. catch(Exception e)
23. {
24. System.out.println("The exception has been caught: " + e);
25. }
26. System.out.println( j );
27. }
28. }
29. }
30.
31. public class ThreadJoinExample
32. {
33. // main method
34. public static void main (String argvs[])
35. {
36.
37. // creating 3 threads
38. ThreadJoin th1 = new ThreadJoin();
39. ThreadJoin th2 = new ThreadJoin();
40. ThreadJoin th3 = new ThreadJoin();
41.
42. // thread th1 starts
43. th1.start();
44.
45. // starting the second thread after when
46. // the first thread th1 has ended or died.
47. try
48. {
49. System.out.println("The current thread name is: "+ Thread.currentThread().getName());
50.
51. // invoking the join() method
52. th1.join();
53. }
54.
55. // catch block for catching the raised exception
56. catch(Exception e)
57. {
58. System.out.println("The exception has been caught " + e);
```

```
59. }
60.
61. // thread th2 starts
62. th2.start();
63.
64. // starting the th3 thread after when the thread th2 has ended or died.
65. try
66. {
67. System.out.println("The current thread name is: " + Thread.currentThread().getName());
68. th2.join();
69. }
70.
71. // catch block for catching the raised exception
72. catch(Exception e)
73. {
74. System.out.println("The exception has been caught " + e);
75. }
76.
77. // thread th3 starts
78. th3.start();
79. }
80. }
```

Output:

```
The current thread name is: main
The current thread name is: Thread - 0
0
The current thread name is: Thread - 0
1
The current thread name is: main
The current thread name is: Thread - 1
0
The current thread name is: Thread - 1
1
The current thread name is: Thread - 2
0
The current thread name is: Thread - 2
1
```

Explanation: The above program shows that the second thread th2 begins after the first thread th1 has ended, and the thread th3 starts its work after the second thread th2 has ended or died.

The Join() Method: InterruptedException

We have learnt in the description of the join() method that whenever the interruption of the thread occurs, it leads to the throwing of InterruptedException. The following example shows the same.

FileName: ThreadJoinExample1.java

```
1. class ABC extends Thread
2. {
3. Thread threadToInterrupt;
```

```
4.  // overriding the run() method
5.  public void run()
6.  {
7.  // invoking the method interrupt
8.  threadToInterrupt.interrupt();
9.  }
10. }
11.
12.
13. public class ThreadJoinExample1
14. {
15. // main method
16. public static void main(String[] argvs)
17. {
18. try
19. {
20. // creating an object of the class ABC
21. ABC th1 = new ABC();
22.
23. th1.threadToInterrupt = Thread.currentThread();
24. th1.start();
25.
26. // invoking the join() method leads
27. // to the generation of InterruptedException
28. th1.join();
29. }
30. catch (InterruptedException ex)
31. {
32. System.out.println("The exception has been caught. " + ex);
33. }
34. }
35. }
```

Output:

```
The exception has been caught. java.lang.InterruptedException
```

Some More Examples of the join() Method

Let' see some other examples.

Filename: TestJoinMethod1.java

```
1.  class TestJoinMethod1 extends Thread{
2.   public void run(){
3.    for(int i=1;i<=5;i++){
4.     try{
5.      Thread.sleep(500);
6.     }catch(Exception e){System.out.println(e);}
7.    System.out.println(i);
8.    }
```

```
9.    }
10. public static void main(String args[]){
11.  TestJoinMethod1 t1=new TestJoinMethod1();
12.  TestJoinMethod1 t2=new TestJoinMethod1();
13.  TestJoinMethod1 t3=new TestJoinMethod1();
14.  t1.start();
15.  try{
16.   t1.join();
17.  }catch(Exception e){System.out.println(e);}
18.
19.  t2.start();
20.  t3.start();
21.  }
22. }
```

Output:

```
1
2
3
4
5
1
1
2
2
3
3
4
4
5
5
```

We can see in the above example, when t1 completes its task then t2 and t3 starts executing.

join(long miliseconds) Method Example

Filename: TestJoinMethod2.jav

```
1.   class TestJoinMethod2 extends Thread{
2.    public void run(){
3.     for(int i=1;i<=5;i++){
4.      try{
5.       Thread.sleep(500);
6.      }catch(Exception e){System.out.println(e);}
7.     System.out.println(i);
8.     }
9.    }
10. public static void main(String args[]){
11.  TestJoinMethod2 t1=new TestJoinMethod2();
12.  TestJoinMethod2 t2=new TestJoinMethod2();
13.  TestJoinMethod2 t3=new TestJoinMethod2();
14.  t1.start();
15.  try{
```

```
16.   t1.join(1500);
17.  }catch(Exception e){System.out.println(e);}
18.
19.  t2.start();
20.  t3.start();
21.  }
22. }
```

Output:

```
 1
2
3
1
4
1
2
5
2
3
3
4
4
5
5
```

In the above example, when t1 completes its task for 1500 milliseconds(3 times), then t2 and t3 start executing.

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.

Example of naming a thread : Using setName() Method

FileName: TestMultiNaming1.java

```
1.  class TestMultiNaming1 extends Thread{
2.    public void run(){
3.     System.out.println("running...");
4.    }
5.   public static void main(String args[]){
6.    TestMultiNaming1 t1=new TestMultiNaming1();
7.    TestMultiNaming1 t2=new TestMultiNaming1();
```

```
8.     System.out.println("Name of t1:"+t1.getName());
9.     System.out.println("Name of t2:"+t2.getName());
10.
11.   t1.start();
12.   t2.start();
13.
14.   t1.setName("Sonoo Jaiswal");
15.   System.out.println("After changing name of t1:"+t1.getName());
16.  }
17. }
```

Output:

```
Name of t1:Thread-0
Name of t2:Thread-1
After changing name of t1:Sonoo Jaiswal
running...
running...
```

Example of naming a thread : 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.

FileName: ThreadNamingExample.java

```
1.  // A Java program that shows how one can
2.  // set the name of a thread at the time
3.  // of creation of the thread
4.
5.  // import statement
6.  import java.io.*;
7.
8.  // The ThreadNameClass is the child class of the class Thread
9.  class ThreadName extends Thread
10. {
11.
12. // constructor of the class
13. ThreadName(String threadName)
14. {
15. // invoking the constructor of
16. // the superclass, which is Thread class.
17. super(threadName);
18. }
19.
20. // overriding the method run()
21. public void run()
22. {
23. System.out.println(" The thread is executing....");
24. }
25. }
26.
```

```
27. public class ThreadNamingExample
28. {
29. // main method
30. public static void main (String argvs[])
31. {
32. // creating two threads and settting their name
33. // using the contructor of the class
34. ThreadName th1 = new ThreadName("JavaTpoint1");
35. ThreadName th2 = new ThreadName("JavaTpoint2");
36.
37. // invoking the getName() method to get the names
38. // of the thread created above
39. System.out.println("Thread - 1: " + th1.getName());
40. System.out.println("Thread - 2: " + th2.getName());
41.
42.
43. // invoking the start() method on both the threads
44. th1.start();
45. th2.start();
46. }
47. }
```

Output:

```
Thread - 1: JavaTpoint1
Thread - 2: JavaTpoint2
 The thread is executing....
 The thread is executing....
```

Current Thread

The currentThread() method returns a reference of the currently executing thread.

1. public static Thread currentThread()

Example of currentThread() method

FileName: TestMultiNaming2.java

```
1. class TestMultiNaming2 extends Thread{
2.  public void run(){
3.   System.out.println(Thread.currentThread().getName());
4.  }
5.  public static void main(String args[]){
6.   TestMultiNaming2 t1=new TestMultiNaming2();
7.   TestMultiNaming2 t2=new TestMultiNaming2();
8.
9.   t1.start();
10.  t2.start();
11. }
12. }
```

Output:

```
Thread-0
Thread-1
```

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.

Example of priority of a Thread:

FileName: ThreadPriorityExample.java

```
1. // Importing the required classes
2. import java.lang.*;
3.
4. public class ThreadPriorityExample extends Thread
5. {
6.
7. // Method 1
8. // Whenever the start() method is called by a thread
9. // the run() method is invoked
10. public void run()
11. {
12. // the print statement
13. System.out.println("Inside the run() method");
```

```
14. }
15.
16. // the main method
17. public static void main(String argvs[])
18. {
19. // Creating threads with the help of ThreadPriorityExample class
20. ThreadPriorityExample th1 = new ThreadPriorityExample();
21. ThreadPriorityExample th2 = new ThreadPriorityExample();
22. ThreadPriorityExample th3 = new ThreadPriorityExample();
23.
24. // We did not mention the priority of the thread.
25. // Therefore, the priorities of the thread is 5, the default value
26.
27. // 1st Thread
28. // Displaying the priority of the thread
29. // using the getPriority() method
30. System.out.println("Priority of the thread th1 is : " + th1.getPriority());
31.
32. // 2nd Thread
33. // Display the priority of the thread
34. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
35.
36. // 3rd Thread
37. // // Display the priority of the thread
38. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
39.
40. // Setting priorities of above threads by
41. // passing integer arguments
42. th1.setPriority(6);
43. th2.setPriority(3);
44. th3.setPriority(9);
45.
46. // 6
47. System.out.println("Priority of the thread th1 is : " + th1.getPriority());
48.
49. // 3
50. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
51.
52. // 9
53. System.out.println("Priority of the thread th3 is : " + th3.getPriority());
54.
55. // Main thread
56.
57. // Displaying name of the currently executing thread
58. System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());
59.
60. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
61.
62. // Priority of the main thread is 10 now
63. Thread.currentThread().setPriority(10);
64.
```

```
65. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
66. }
67. }
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java thread scheduler. Refer to the following example to comprehend what will happen if two threads have the same priority.

FileName: ThreadPriorityExample1.java

```
1.  // importing the java.lang package
2.  import java.lang.*;
3.
4.  public class ThreadPriorityExample1 extends Thread
5.  {
6.
7.  // Method 1
8.  // Whenever the start() method is called by a thread
9.  // the run() method is invoked
10. public void run()
11. {
12. // the print statement
13. System.out.println("Inside the run() method");
14. }
15.
16.
17. // the main method
18. public static void main(String argvs[])
19. {
20.
21. // Now, priority of the main thread is set to 7
22. Thread.currentThread().setPriority(7);
23.
24. // the current thread is retrieved
25. // using the currentThread() method
26.
27. // displaying the main thread priority
28. // using the getPriority() method of the Thread class
29. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
30.
```

```
31. // creating a thread by creating an object of the class ThreadPriorityExample1
32. ThreadPriorityExample1 th1 = new ThreadPriorityExample1();
33.
34. // th1 thread is the child of the main thread
35. // therefore, the th1 thread also gets the priority 7
36.
37. // Displaying the priority of the current thread
38. System.out.println("Priority of the thread th1 is : " + th1.getPriority());
39. }
40. }
```

Output:

```
Priority of the main thread is : 7
Priority of the thread th1 is : 7
```

Explanation: If there are two threads that have the same priority, then one can not predict which thread will get the chance to execute first. The execution then is dependent on the thread scheduler's algorithm (First Come First Serve, Round-Robin, etc.)

Example of IllegalArgumentException

We know that if the value of the parameter *newPriority* of the method getPriority() goes out of the range (1 to 10), then we get the IllegalArgumentException. Let's observe the same with the help of an example.

FileName: IllegalArgumentException.java

```
1. // importing the java.lang package
2. import java.lang.*;
3.
4. public class IllegalArgumentException extends Thread
5. {
6.
7. // the main method
8. public static void main(String argvs[])
9. {
10.
11. // Now, priority of the main thread is set to 17, which is greater than 10
12. Thread.currentThread().setPriority(17);
13.
14. // The current thread is retrieved
15. // using the currentThread() method
16.
17. // displaying the main thread priority
18. // using the getPriority() method of the Thread class
19. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
20.
21. }
22. }
```

When we execute the above program, we get the following exception:

```
Exception in thread "main" java.lang.IllegalArgumentException
        at java.base/java.lang.Thread.setPriority(Thread.java:1141)
        at IllegalArgumentException.main(IllegalArgumentException.java:12)
```