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

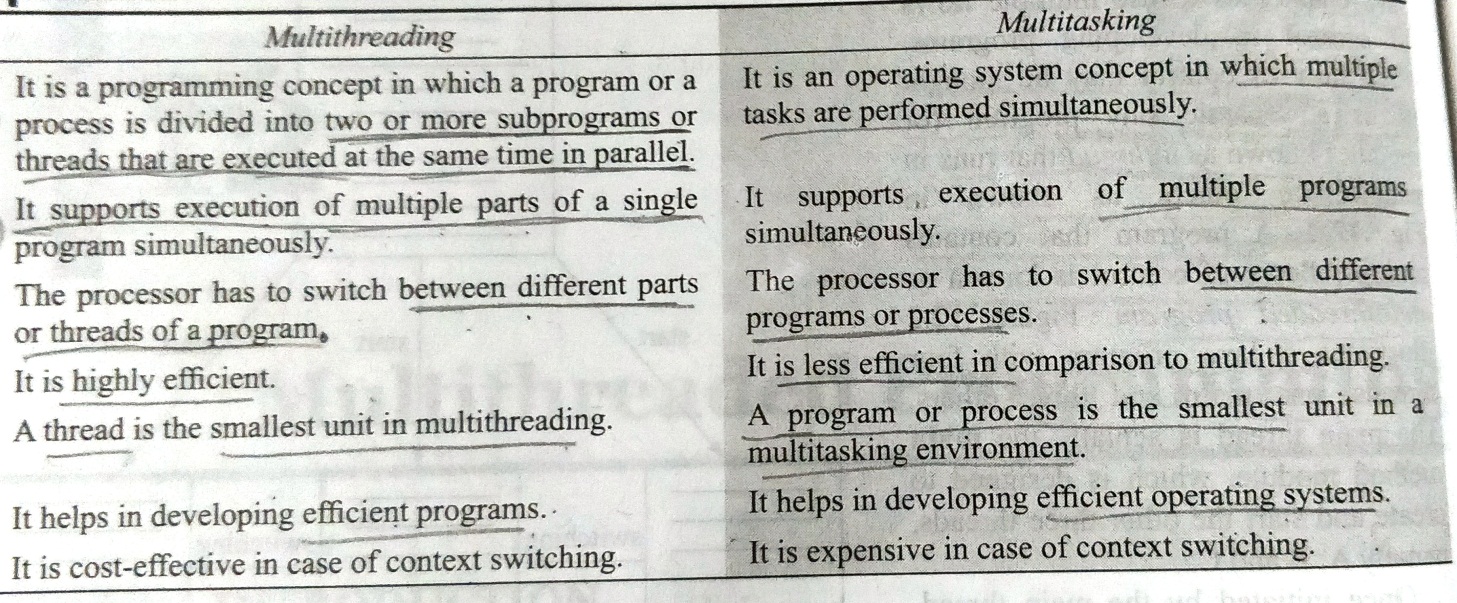
A thread is a lightweight subprocess, **the smallest unit of processing**. It is a separate path of execution.

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.

****



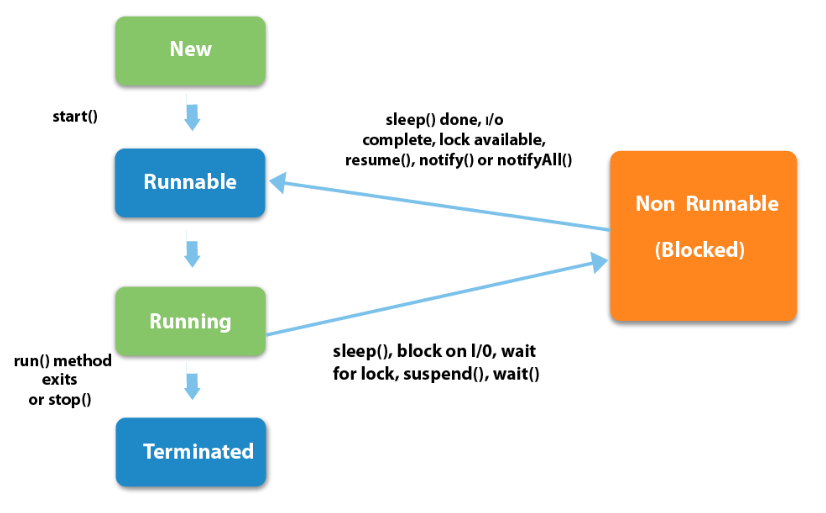
# Life cycle of a Thread (Thread States)

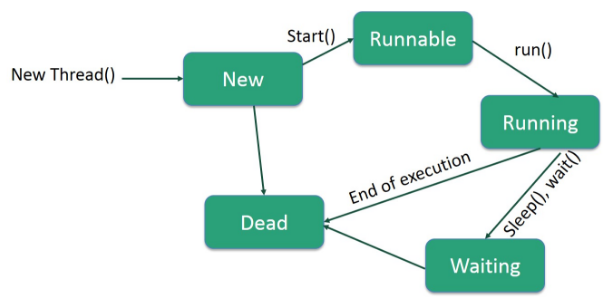
A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated





|  |
| --- |
| 1) New The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

### 2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### 3) Running

The thread is in running state if the thread scheduler has selected it.

### 4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

### 5) Terminated

A thread is in terminated or dead state when its run() method exits.

# 

# 

# 

### New

When we create a new Thread object using new operator, thread state is New Thread. At this point, thread is not alive and it’s a state internal to Java programming.

### Runnable

When we call start() function on Thread object, it’s state is changed to Runnable. The control is given to Thread scheduler to finish it’s execution. Whether to run this thread instantly or keep it in runnable thread pool before running, depends on the OS implementation of thread scheduler.

### Running

When thread is executing, it’s state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it’s state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

### Blocked/Waiting

A thread can be waiting for other thread to finish using [thread join](https://www.journaldev.com/1024/java-thread-join-example) or it can be waiting for some resources to available. For example [producer consumer problem](https://www.journaldev.com/1034/java-blockingqueue-example) or [waiter notifier implementation](https://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example) or IO resources, then it’s state is changed to Waiting. Once the thread wait state is over, it’s state is changed to Runnable and it’s moved back to runnable thread pool.

### Dead

Once the thread finished executing, it’s state is changed to Dead and it’s considered to be not alive.

# How to create thread

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:

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs 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

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. }
9. }

Output:thread is running...

### 2) Java Thread Example by implementing Runnable interface

1. **class** Multi3 **implements** Runnable{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
6. **public** **static** **void** main(String args[]){
7. Multi3 m1=**new** Multi3();
8. Thread t1 =**new** Thread(m1);
9. t1.start();
10. }
11. }

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 explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute. |

# Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

## Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

* public static void sleep(long miliseconds)throws InterruptedException
* public static void sleep(long miliseconds, int nanos)throws InterruptedException

## Example of sleep method in java

1. **class** TestSleepMethod1 **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. TestSleepMethod1 t1=**new** TestSleepMethod1();
10. TestSleepMethod1 t2=**new** TestSleepMethod1();
12. t1.start();
13. t2.start();
14. }
15. }

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 shedular picks up another thread and so on.

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. }

running

Exception in thread "main" java.lang.IllegalThreadStateException

# What if we call run() method directly instead start() method?

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

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. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun1)

Output:running...

***Problem if you direct call run() method***

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();
12. t1.run();
13. t2.run();
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun2)

Output:1

2

3

4

5

1

2

3

4

5

|  |
| --- |
| As you 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. |

# The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

### Syntax:

|  |
| --- |
| public void join()throws InterruptedException |
| public void join(long milliseconds)throws InterruptedException |

***Example of join() method***

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);}
19. t2.start();
20. t3.start();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod1)

Output:1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

|  |
| --- |
| As you can see in the above example,when t1 completes its task then t2 and t3 starts executing. |

***Example of join(long miliseconds) method***

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);}
19. t2.start();
20. t3.start();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod2)

Output:1

2

3

1

4

1

2

5

2

3

3

4

4

5

5

|  |
| --- |
| In the above example,when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing. |

### getName(),setName(String) and getId() method:

|  |
| --- |
| public String getName() |
| public void setName(String name) |
| public long getId() |

1. **class** TestJoinMethod3 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestJoinMethod3 t1=**new** TestJoinMethod3();
7. TestJoinMethod3 t2=**new** TestJoinMethod3();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
10. System.out.println("id of t1:"+t1.getId());
12. t1.start();
13. t2.start();
15. t1.setName("Sonoo Jaiswal");
16. System.out.println("After changing name of t1:"+t1.getName());
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod3)

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changling name of t1:Sonoo Jaiswal

running...

### The currentThread() method:

|  |
| --- |
| The currentThread() method returns a reference to the currently executing thread object. |

### Syntax:

|  |
| --- |
| public static Thread currentThread() |

***Example of currentThread() method***

1. **class** TestJoinMethod4 **extends** Thread{
2. **public** **void** run(){
3. System.out.println(Thread.currentThread().getName());
4. }
5. }
6. **public** **static** **void** main(String args[]){
7. TestJoinMethod4 t1=**new** TestJoinMethod4();
8. TestJoinMethod4 t2=**new** TestJoinMethod4();
10. t1.start();
11. t2.start();
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod4)

Output:Thread-0

Thread-1

# Priority of a Thread (Thread Priority):

|  |
| --- |
| Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular 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. |

## 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:

1. **class** TestMultiPriority1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running thread name is:"+Thread.currentThread().getName());
4. System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
6. }
7. **public** **static** **void** main(String args[]){
8. TestMultiPriority1 m1=**new** TestMultiPriority1();
9. TestMultiPriority1 m2=**new** TestMultiPriority1();
10. m1.setPriority(Thread.MIN\_PRIORITY);
11. m2.setPriority(Thread.MAX\_PRIORITY);
12. m1.start();
13. m2.start();
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiPriority1)

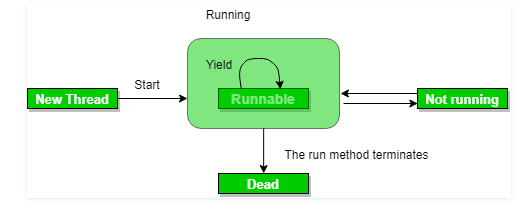
Output:running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1

**yield():** Suppose there are three threads t1, t2, and t3. Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state. Completion time for thread t1 is 5 hour and completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job. In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent execution of a thread in between if something important is pending. yeild() helps us in doing so.  
**yield()**basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.



**Use of yield method:**

* Whenever a thread calls java.lang.Thread.yield method, it gives hint to the thread scheduler that it is ready to pause its execution. Thread scheduler is free to ignore this hint.
* If any thread executes yield method , thread scheduler checks if there is any thread with same or high priority than this thread. If processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give processor to other thread and if not – current thread will keep executing.

**Syntax:**

**Syntax:**

public static native void yield()

|  |
| --- |
| // Java program to illustrate yield() method  // in Java  import java.lang.\*;    // MyThread extending Thread  class MyThread extends Thread  {      public void run()      {          for (int i=0; i<5 ; i++)              System.out.println(Thread.currentThread().getName()                                  + " in control");      }  }    // Driver Class  public class yieldDemo  {      public static void main(String[]args)      {          MyThread t = new MyThread();          t.start();            for (int i=0; i<5; i++)          {              // Control passes to child thread              Thread.yield();                // After execution of child Thread              // main thread takes over              System.out.println(Thread.currentThread().getName()                                  + " in control");          }      }  } |

**Output:**

Thread-0 in control

Thread-0 in control

Thread-0 in control

Thread-0 in control

Thread-0 in control

main in control

main in control

main in control

main in control

main in control

**yield() vs sleep()**

**yield:()** indicates that the thread is not doing anything particularly important and if any other threads or processes need to be run, they can. Otherwise, the **current thread will continue to run.**

**sleep()**: causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, **the CPU will be idle** (and probably enter a power saving mode).