Multithreading in Java

**Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing multiple threads simultaneously.

P1 p2 p3 p1

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](https://www.javatpoint.com/register-memory), 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.**

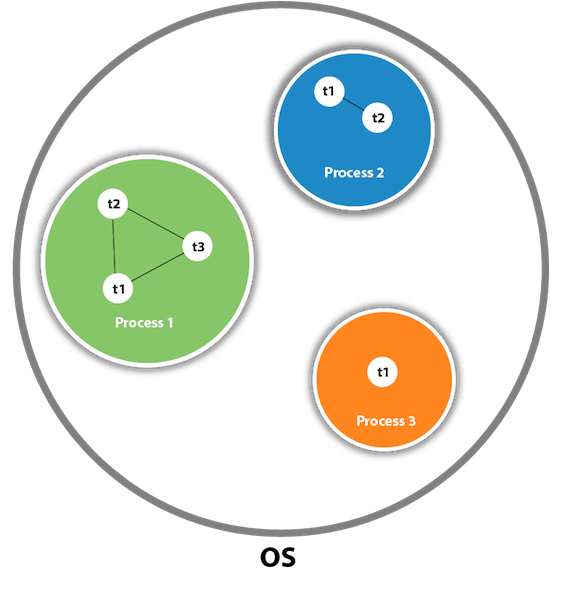
## **What is Thread in java**

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

Extends Thread class

Implements Runnable interface

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

Java Multithreading 

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](https://www.javatpoint.com/os-tutorial), 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](https://www.javatpoint.com/java-constructor) and methods to create and perform operations on a thread. Thread class extends [Object class](https://www.javatpoint.com/object-class) and implements Runnable interface.

## **Java Thread Methods**

| **S.N.** | **Modifier and Type** | **Method** | **Description** |
| --- | --- | --- | --- |
| 1) | void | [start()](https://www.javatpoint.com/java-thread-start-method) | It is used to start the execution of the thread. |
| 2) | void | [run()](https://www.javatpoint.com/java-thread-run-method) | It is used to do an action for a thread. |
| 3) | static void | [sleep()](https://www.javatpoint.com/java-thread-sleep-method) | It sleeps a thread for the specified amount of time. |
| 4) | static Thread | [currentThread()](https://www.javatpoint.com/java-thread-currentthread-method) | It returns a reference to the currently executing thread object. |
| 5) | void | [join()](https://www.javatpoint.com/java-thread-join-method) | It waits for a thread to die. |
| 6) | int | [getPriority()](https://www.javatpoint.com/java-thread-getpriority-method) | It returns the priority of the thread. |
| 7) | void | [setPriority()](https://www.javatpoint.com/java-thread-setpriority-method) | It changes the priority of the thread. |
| 8) | String | [getName()](https://www.javatpoint.com/java-thread-getname-method) | It returns the name of the thread. |
| 9) | void | [setName()](https://www.javatpoint.com/java-thread-setname-method) | It changes the name of the thread. |
| 10) | long | [getId()](https://www.javatpoint.com/java-thread-getid-method) | It returns the id of the thread. |
| 11) | boolean | [isAlive()](https://www.javatpoint.com/java-thread-isalive-method) | It tests if the thread is alive. |
| 12) | static void | [yield()](https://www.javatpoint.com/java-thread-yield-method) | It causes the currently executing thread object to pause and allow other threads to execute temporarily. |
| 13) | void | [suspend()](https://www.javatpoint.com/java-thread-suspend-method) | It is used to suspend the thread. |
| 14) | void | [resume()](https://www.javatpoint.com/java-thread-resume-method) | It is used to resume the suspended thread. |
| 15) | void | [stop()](https://www.javatpoint.com/java-thread-stop-method) | It is used to stop the thread. |
| 16) | void | [destroy()](https://www.javatpoint.com/java-thread-destroy-method) | It is used to destroy the thread group and all of its subgroups. |
| 17) | void | [notify()](https://www.javatpoint.com/java-thread-notify-method) | It is used to give the notification for only one thread which is waiting for a particular object. |
| 18) | void | [notifyAll()](https://www.javatpoint.com/java-thread-notifyall-method) | It is used to give the notification to all waiting threads of a particular object. |

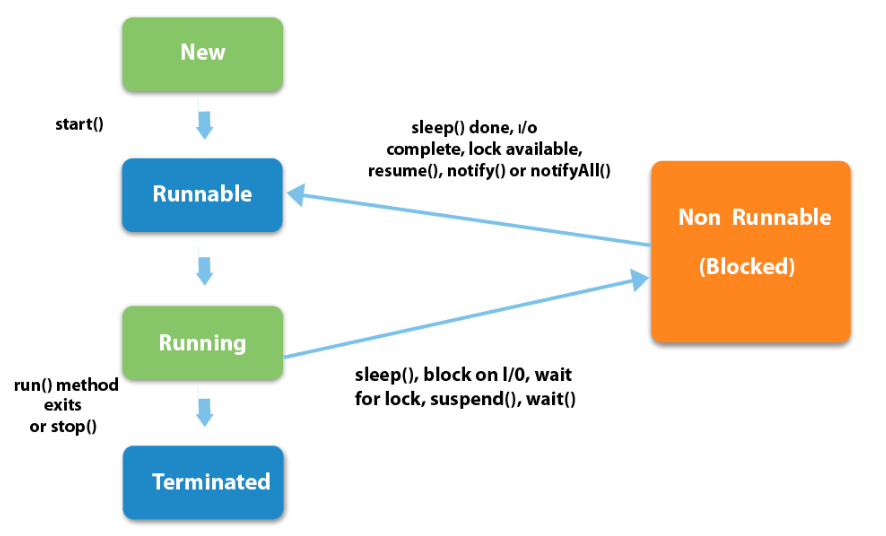
# Life cycle of a Thread (Thread States)

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 a new state if you create an instance of Thread class butbefore 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.

# 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 boolean isAlive():**tests if the thread is alive. 9. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 10. **public void suspend():**is used to suspend the thread(depricated). 11. **public void resume():**is used to resume the suspended thread(depricated). 12. **public void stop():**is used to stop the thread(depricated). |
| --- |

### **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. |
| --- |

### **Java Thread Example by extending Thread class**

**class** Multi **extends** Thread{

**public** **void** run(){

System.out.println("thread is running...");

}

**public** **static** **void** main(String args[]){

Multi t1=**new** Multi();

  Multi t2=**new** Multi();

t1.start();

t2.start();

 }

}

Output:thread is running..

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

**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);

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

# Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

### **Difference between preemptive scheduling and time slicing**

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

# 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

## **for(1 to 3)**

## **{ sop i**

sleep (100)

## **}**

## **t1->1,2**

## **t2->1,**

## **1 1 1 2 2 3 3 3**

## **Example of sleep method in java**

**class** TestSleepMethod1 **extends** Thread{

**public** **void** run(){

**for**(**int** i=1;i<5;i++){

**try**{Thread.sleep(500);}

**catch**(InterruptedException e){System.out.println(e);}

    System.out.println(i);

  }

 }

**public** **static** **void** main(String args[]){

  TestSleepMethod1 t1=**new** TestSleepMethod1();

  TestSleepMethod1 t2=**new** TestSleepMethod1();

  t1.start();

  t2.start();

 }

}

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

**public** **class** TestThreadTwice1 **extends** Thread{

**public** **void** run(){

   System.out.println("running...");

 }

**public** **static** **void** main(String args[]){

  TestThreadTwice1 t1=**new** TestThreadTwice1();

  t1.start();

  t1.start();

 }

}

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.

**class** TestCallRun1 **extends** Thread{

**public** **void** run(){

   System.out.println("running...");

 }

**public** **static** **void** main(String args[]){

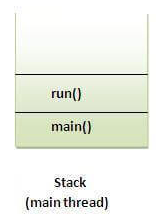
  TestCallRun1 t1=**new** TestCallRun1();

  t1.run();//fine, but does not start a separate call stack

 }

}

Output:running...



**class** TestCallRun2 **extends** Thread{

**public** **void** run(){

**for**(**int** i=1;i<5;i++){

**try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}

    System.out.println(i);

  }

 }

**public** **static** **void** main(String args[]){

  TestCallRun2 t1=**new** TestCallRun2();

  TestCallRun2 t2=**new** TestCallRun2();

  t1.run();

  t2.run();

 }

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***

**class** TestJoinMethod1 **extends** Thread{

**public** **void** run(){

**for**(**int** i=1;i<=5;i++){

**try**{

    Thread.sleep(500);

   }**catch**(Exception e){System.out.println(e);}

  System.out.println(i);

  }

 }

**public** **static** **void** main(String args[]){

 TestJoinMethod1 t1=**new** TestJoinMethod1();

 TestJoinMethod1 t2=**new** TestJoinMethod1();

 TestJoinMethod1 t3=**new** TestJoinMethod1();

 t1.start();

**try**{

  t1.join();

 }**catch**(Exception e){System.out.println(e);}

 t2.start();

 t3.start();

 }

}

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***

**class** TestJoinMethod2 **extends** Thread{

**public** **void** run(){

**for**(**int** i=1;i<=5;i++){

**try**{

    Thread.sleep(500);

   }**catch**(Exception e){System.out.println(e);}

  System.out.println(i);

  }

 }

**public** **static** **void** main(String args[]){

 TestJoinMethod2 t1=**new** TestJoinMethod2();

 TestJoinMethod2 t2=**new** TestJoinMethod2();

 TestJoinMethod2 t3=**new** TestJoinMethod2();

 t1.start();

**try**{

  t1.join(1500);

 }**catch**(Exception e){System.out.println(e);}

 t2.start();

 t3.start();

 }

}

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() |

**class** TestJoinMethod3 **extends** Thread{

**public** **void** run(){

   System.out.println("running...");

  }

**public** **static** **void** main(String args[]){

  TestJoinMethod3 t1=**new** TestJoinMethod3();

  TestJoinMethod3 t2=**new** TestJoinMethod3();

  System.out.println("Name of t1:"+t1.getName());

  System.out.println("Name of t2:"+t2.getName());

  System.out.println("id of t1:"+t1.getId());

  t1.start();

  t2.start();

  t1.setName("Sonoo Jaiswal");

  System.out.println("After changing name of t1:"+t1.getName());

 }

}

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***

**class** TestJoinMethod4 **extends** Thread{

**public** **void** run(){

  System.out.println(Thread.currentThread().getName());

 }

 }

**public** **static** **void** main(String args[]){

  TestJoinMethod4 t1=**new** TestJoinMethod4();

  TestJoinMethod4 t2=**new** TestJoinMethod4();

  t1.start();

  t2.start();

 }

}

Output:Thread-0

Thread-1

# 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 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.

## **Example of naming a thread**

**class** TestMultiNaming1 **extends** Thread{

**public** **void** run(){

   System.out.println("running...");

  }

**public** **static** **void** main(String args[]){

  TestMultiNaming1 t1=**new** TestMultiNaming1();

  TestMultiNaming1 t2=**new** TestMultiNaming1();

  System.out.println("Name of t1:"+t1.getName());

  System.out.println("Name of t2:"+t2.getName());

  t1.start();

  t2.start();

  t1.setName("Sonoo Jaiswal");

  System.out.println("After changing name of t1:"+t1.getName());

 }

}

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changeling name of t1:Sonoo Jaiswal

running...

## **Current Thread**

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

**public** **static** Thread currentThread()

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

**class** TestMultiNaming2 **extends** Thread{

**public** **void** run(){

  System.out.println(Thread.currentThread().getName());

 }

**public** **static** **void** main(String args[]){

  TestMultiNaming2 t1=**new** TestMultiNaming2();

  TestMultiNaming2 t2=**new** TestMultiNaming2();

  t1.start();

  t2.start();

 }

}

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:**

**class** TestMultiPriority1 **extends** Thread{

**public** **void** run(){

   System.out.println("running thread name is:"+Thread.currentThread().getName());

   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }

**public** **static** **void** main(String args[]){

  TestMultiPriority1 m1=**new** TestMultiPriority1();

  TestMultiPriority1 m2=**new** TestMultiPriority1();

  m1.setPriority(Thread.MIN\_PRIORITY);

  m2.setPriority(Thread.MAX\_PRIORITY);

  m1.start();

  m2.start();

 }

}

Output:running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1