Multithreading in Java

1. [Multithreading](https://www.javatpoint.com/multithreading-in-java)
2. [Multitasking](https://www.javatpoint.com/multithreading-in-java" \l "multitasing)
3. [Process-based multitasking](https://www.javatpoint.com/multithreading-in-java" \l "multiprocessing)
4. [Thread-based multitasking](https://www.javatpoint.com/multithreading-in-java" \l "multithreading)
5. [What is Thread](https://www.javatpoint.com/multithreading-in-java" \l "thread)

**Multithreading in [Java](https://www.javatpoint.com/java-tutorial)** 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.

### 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.

## Java Thread Methods

Life cycle of a Thread (Thread States)

1. [Life cycle of a thread](https://www.javatpoint.com/life-cycle-of-a-thread)
   1. [New](https://www.javatpoint.com/life-cycle-of-a-thread" \l "threadstatenew)
   2. [Runnable](https://www.javatpoint.com/life-cycle-of-a-thread" \l "threadstaterunnable)
   3. [Running](https://www.javatpoint.com/life-cycle-of-a-thread" \l "threadstaterunning)
   4. [Non-Runnable (Blocked)](https://www.javatpoint.com/life-cycle-of-a-thread" \l "threadstateblocked)
   5. [Terminated](https://www.javatpoint.com/life-cycle-of-a-thread" \l "threadstateterminated)

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.

# 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) |

### 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

**class** Multi **extends** Thread{

**public** **void** run(){

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

}

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

Multi t1=**new** Multi();

t1.start();

 }

}

### 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. }

|  |  |  |  |
| --- | --- | --- | --- |
| **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) | boolean | [isDaemon()](https://www.javatpoint.com/java-thread-isdaemon-method) | It tests if the thread is a daemon thread. |
| 18) | void | [setDaemon()](https://www.javatpoint.com/java-thread-setdaemon-method) | It marks the thread as daemon or user thread. |
| 19) | void | [interrupt()](https://www.javatpoint.com/java-thread-interrupt-method) | It interrupts the thread. |
| 20) | boolean | [isinterrupted()](https://www.javatpoint.com/java-thread-isinterrupted-method) | It tests whether the thread has been interrupted. |
| 21) | static boolean | [interrupted()](https://www.javatpoint.com/java-thread-interrupted-method) | It tests whether the current thread has been interrupted. |
| 22) | static int | [activeCount()](https://www.javatpoint.com/java-thread-activecount-method) | It returns the number of active threads in the current thread's thread group. |
| 23) | void | [checkAccess()](https://www.javatpoint.com/java-thread-checkaccess-method) | It determines if the currently running thread has permission to modify the thread. |
| 24) | static boolean | [holdLock()](https://www.javatpoint.com/java-thread-holdlock-method) | It returns true if and only if the current thread holds the monitor lock on the specified object. |
| 25) | static void | [dumpStack()](https://www.javatpoint.com/java-thread-dumpstack-method) | It is used to print a stack trace of the current thread to the standard error stream. |
| 26) | StackTraceElement[] | [getStackTrace()](https://www.javatpoint.com/java-thread-getstacktrace-method) | It returns an array of stack trace elements representing the stack dump of the thread. |
| 27) | static int | [enumerate()](https://www.javatpoint.com/java-thread-enumerate-method) | It is used to copy every active thread's thread group and its subgroup into the specified array. |
| 28) | Thread.State | [getState()](https://www.javatpoint.com/java-thread-getstate-method) | It is used to return the state of the thread. |
| 29) | ThreadGroup | [getThreadGroup()](https://www.javatpoint.com/java-thread-getthreadgroup-method) | It is used to return the thread group to which this thread belongs |
| 30) | String | [toString()](https://www.javatpoint.com/java-thread-tostring-method) | It is used to return a string representation of this thread, including the thread's name, priority, and thread group. |
| 31) | 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. |
| 32) | 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. |
| 33) | void | [setContextClassLoader()](https://www.javatpoint.com/java-thread-setcontextclassloader-method) | It sets the context ClassLoader for the Thread. |
| 34) | ClassLoader | [getContextClassLoader()](https://www.javatpoint.com/java-thread-getcontextclassloader-method) | It returns the context ClassLoader for the thread. |
| 35) | static Thread.UncaughtExceptionHandler | [getDefaultUncaughtExceptionHandler()](https://www.javatpoint.com/java-thread-getdefaultuncaughtexceptionhandler-method) | It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception. |
| 36) | static void | [setDefaultUncaughtExceptionHandler()](https://www.javatpoint.com/java-thread-setdefaultuncaughtexceptionhandler-method) | It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception. |

* Cost of communication between the thread is low.

## 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.

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.

# 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.

**[next →](https://www.javatpoint.com/can-we-start-a-thread-twice)[← prev](https://www.javatpoint.com/Thread-Schedular)**

# 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. }

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. }

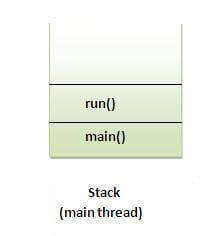
# What if we call run() method directly instead start() method?

|  |
| --- |
| * 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" \t "_blank)**

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. }

# 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. }

***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. }