# Multithreading in Java

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

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.

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](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.

AD

## 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) | 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. |

# 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.

AD

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.



## 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. // try-catch block
7. **try**
8. {
9. // moving thread t2 to the state timed waiting
10. Thread.sleep(100);
11. }
12. **catch** (InterruptedException ie)
13. {
14. ie.printStackTrace();
15. }

18. System.out.println("The state of thread t1 while it invoked the method join() on thread t2 -"+ ThreadState.t1.getState());
20. // try-catch block
21. **try**
22. {
23. Thread.sleep(200);
24. }
25. **catch** (InterruptedException ie)
26. {
27. ie.printStackTrace();
28. }
29. }
30. }
32. // ThreadState class implements the interface Runnable
33. **public** **class** ThreadState **implements** Runnable
34. {
35. **public** **static** Thread t1;
36. **public** **static** ThreadState obj;
38. // main method
39. **public** **static** **void** main(String argvs[])
40. {
41. // creating an object of the class ThreadState
42. obj = **new** ThreadState();
43. t1 = **new** Thread(obj);
45. // thread t1 is spawned
46. // The thread t1 is currently in the NEW state.
47. System.out.println("The state of thread t1 after spawning it - " + t1.getState());
49. // invoking the start() method on
50. // the thread t1
51. t1.start();
53. // thread t1 is moved to the Runnable state
54. System.out.println("The state of thread t1 after invoking the method start() on it - " + t1.getState());
55. }
57. **public** **void** run()
58. {
59. ABC myObj = **new** ABC();
60. Thread t2 = **new** Thread(myObj);
62. // thread t2 is created and is currently in the NEW state.
63. System.out.println("The state of thread t2 after spawning it - "+ t2.getState());
64. t2.start();
66. // thread t2 is moved to the runnable state
67. System.out.println("the state of thread t2 after calling the method start() on it - " + t2.getState());
69. // try-catch block for the smooth flow of the  program
70. **try**
71. {
72. // moving the thread t1 to the state timed waiting
73. Thread.sleep(200);
74. }
75. **catch** (InterruptedException ie)
76. {
77. ie.printStackTrace();
78. }
79. System.out.println("The state of thread t2 after invoking the method sleep() on it - "+ t2.getState() );
81. // try-catch block for the smooth flow of the  program
82. **try**
83. {
84. // waiting for thread t2 to complete its execution
85. t2.join();
86. }
87. **catch** (InterruptedException ie)
88. {
89. ie.printStackTrace();
90. }
91. System.out.println("The state of thread t2 when it has completed it's execution - " + t2.getState());
92. }
93. }

**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.

AD

### 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. }
9. }

**Output:**

thread is running...

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

**FileName:** Multi3.java

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);   // Using the constructor Thread(Runnable r)
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 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

1. **public** **class** MyThread1
2. {
3. // Main method
4. **public** **static** **void** main(String argvs[])
5. {
6. // creating an object of the Thread class using the constructor Thread(String name)
7. Thread t= **new** Thread("My first thread");
9. // the start() method moves the thread to the active state
10. t.start();
11. // getting the thread name by invoking the getName() method
12. String str = t.getName();
13. System.out.println(str);
14. }
15. }

**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. }
8. // main method
9. **public** **static** **void** main(String argvs[])
10. {
11. // creating an object of the class MyThread2
12. Runnable r1 = **new** MyThread2();
14. // creating an object of the class Thread using Thread(Runnable r, String name)
15. Thread th1 = **new** Thread(r1, "My new thread");
17. // the start() method moves the thread to the active state
18. th1.start();
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 ...

# 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. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestThreadTwice1)

**Output:**

running

Exception in thread "main" java.lang.IllegalThreadStateException

# 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 milliseconds.

## 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
4. // import statement
5. **import** java.io.\*;
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. }
31. **public** **class** ThreadJoinExample
32. {
33. // main method
34. **public** **static** **void** main (String argvs[])
35. {
37. // creating 3 threads
38. ThreadJoin th1 = **new** ThreadJoin();
39. ThreadJoin th2 = **new** ThreadJoin();
40. ThreadJoin th3 = **new** ThreadJoin();
42. // thread th1 starts
43. th1.start();
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());
51. // invoking the join() method
52. th1.join();
53. }
55. // catch block for catching the raised exception
56. **catch**(Exception e)
57. {
58. System.out.println("The exception has been caught " + e);
59. }
61. // thread th2 starts
62. th2.start();
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. }
71. // catch block for catching the raised exception
72. **catch**(Exception e)
73. {
74. System.out.println("The exception has been caught " + e);
75. }
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. }

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();
23. th1.threadToInterrupt = Thread.currentThread();
24. th1.start();
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

AD

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

**Output:**

AD

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);}
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.AD

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

**Output:**

AD

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);}
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.

# Daemon Thread in Java

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

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

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

## Points to remember for Daemon Thread in Java

* It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
* Its life depends on user threads.
* It is a low priority thread.

### Why JVM terminates the daemon thread if there is no user thread?

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

### Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

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

### Simple example of Daemon thread in java

*File: MyThread.java*

1. **public** **class** TestDaemonThread1 **extends** Thread{
2. **public** **void** run(){
3. **if**(Thread.currentThread().isDaemon()){//checking for daemon thread
4. System.out.println("daemon thread work");
5. }
6. **else**{
7. System.out.println("user thread work");
8. }
9. }
10. **public** **static** **void** main(String[] args){
11. TestDaemonThread1 t1=**new** TestDaemonThread1();//creating thread
12. TestDaemonThread1 t2=**new** TestDaemonThread1();
13. TestDaemonThread1 t3=**new** TestDaemonThread1();
15. t1.setDaemon(**true**);//now t1 is daemon thread
17. t1.start();//starting threads
18. t2.start();
19. t3.start();
20. }
21. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread1)

**Output:**

daemon thread work

user thread work

user thread work

#### Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

*File: MyThread.java*

1. **class** TestDaemonThread2 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("Name: "+Thread.currentThread().getName());
4. System.out.println("Daemon: "+Thread.currentThread().isDaemon());
5. }
7. **public** **static** **void** main(String[] args){
8. TestDaemonThread2 t1=**new** TestDaemonThread2();
9. TestDaemonThread2 t2=**new** TestDaemonThread2();
10. t1.start();
11. t1.setDaemon(**true**);//will throw exception here
12. t2.start();
13. }
14. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread2)

**Output:**

exception in thread main: java.lang.IllegalThreadStateException

# Java Thread Pool

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

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

## Thread Pool Methods

**newFixedThreadPool(int s):** The method creates a thread pool of the fixed size s.

**newCachedThreadPool():** The method creates a new thread pool that creates the new threads when needed but will still use the previously created thread whenever they are available to use.

**newSingleThreadExecutor():** The method creates a new thread.

### Advantage of Java Thread Pool

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

### Real time usage

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

## Example of Java Thread Pool

Let's see a simple example of the Java thread pool using ExecutorService and Executors.

**File: WorkerThread.java**

1. **import** java.util.concurrent.ExecutorService;
2. **import** java.util.concurrent.Executors;
3. **class** WorkerThread **implements** Runnable {
4. **private** String message;
5. **public** WorkerThread(String s){
6. **this**.message=s;
7. }
8. **public** **void** run() {
9. System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);
10. processmessage();//call processmessage method that sleeps the thread for 2 seconds
11. System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name
12. }
13. **private** **void** processmessage() {
14. **try** {  Thread.sleep(2000);  } **catch** (InterruptedException e) { e.printStackTrace(); }
15. }
16. }

**File: TestThreadPool.java**

1. **public** **class** TestThreadPool {
2. **public** **static** **void** main(String[] args) {
3. ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads
4. **for** (**int** i = 0; i < 10; i++) {
5. Runnable worker = **new** WorkerThread("" + i);
6. executor.execute(worker);//calling execute method of ExecutorService
7. }
8. executor.shutdown();
9. **while** (!executor.isTerminated()) {   }
11. System.out.println("Finished all threads");
12. }
13. }

**Output:**

pool-1-thread-1 (Start) message = 0

pool-1-thread-2 (Start) message = 1

pool-1-thread-3 (Start) message = 2

pool-1-thread-5 (Start) message = 4

pool-1-thread-4 (Start) message = 3

pool-1-thread-2 (End)

pool-1-thread-2 (Start) message = 5

pool-1-thread-1 (End)

pool-1-thread-1 (Start) message = 6

pool-1-thread-3 (End)

pool-1-thread-3 (Start) message = 7

pool-1-thread-4 (End)

pool-1-thread-4 (Start) message = 8

pool-1-thread-5 (End)

pool-1-thread-5 (Start) message = 9

pool-1-thread-2 (End)

pool-1-thread-1 (End)

pool-1-thread-4 (End)

pool-1-thread-3 (End)

pool-1-thread-5 (End)

Finished all threads

[download this example](https://static.javatpoint.com/src/multi/threadpool.zip)

## Thread Pool Example: 2

Let's see another example of the thread pool.

**FileName:** ThreadPoolExample.java

1. // important import statements
2. **import** java.util.Date;
3. **import** java.util.concurrent.ExecutorService;
4. **import** java.util.concurrent.Executors;
5. **import** java.text.SimpleDateFormat;

8. **class** Tasks **implements** Runnable
9. {
10. **private** String taskName;
12. // constructor of the class Tasks
13. **public** Tasks(String str)
14. {
15. // initializing the field taskName
16. taskName = str;
17. }
19. // Printing the task name and then sleeps for 1 sec
20. // The complete process is getting repeated five times
21. **public** **void** run()
22. {
23. **try**
24. {
25. **for** (**int** j = 0; j <= 5; j++)
26. {
27. **if** (j == 0)
28. {
29. Date dt = **new** Date();
30. SimpleDateFormat sdf = **new** SimpleDateFormat("hh : mm : ss");
32. //prints the initialization time for every task
33. System.out.println("Initialization time for the task name: "+ taskName + " = " + sdf.format(dt));
35. }
36. **else**
37. {
38. Date dt = **new** Date();
39. SimpleDateFormat sdf = **new** SimpleDateFormat("hh : mm : ss");
41. // prints the execution time for every task
42. System.out.println("Time of execution for the task name: " + taskName + " = " +sdf.format(dt));
44. }
46. // 1000ms = 1 sec
47. Thread.sleep(1000);
48. }
50. System.out.println(taskName + " is complete.");
51. }
53. **catch**(InterruptedException ie)
54. {
55. ie.printStackTrace();
56. }
57. }
58. }
60. **public** **class** ThreadPoolExample
61. {
62. // Maximum number of threads in the thread pool
63. **static** **final** **int** MAX\_TH = 3;
65. // main method
66. **public** **static** **void** main(String argvs[])
67. {
68. // Creating five new tasks
69. Runnable rb1 = **new** Tasks("task 1");
70. Runnable rb2 = **new** Tasks("task 2");
71. Runnable rb3 = **new** Tasks("task 3");
72. Runnable rb4 = **new** Tasks("task 4");
73. Runnable rb5 = **new** Tasks("task 5");
75. // creating a thread pool with MAX\_TH number of
76. // threads size the pool size is fixed
77. ExecutorService pl = Executors.newFixedThreadPool(MAX\_TH);
79. // passes the Task objects to the pool to execute (Step 3)
80. pl.execute(rb1);
81. pl.execute(rb2);
82. pl.execute(rb3);
83. pl.execute(rb4);
84. pl.execute(rb5);
86. // pool is shutdown
87. pl.shutdown();
88. }
89. }

**Output:**

Initialization time for the task name: task 1 = 06 : 13 : 02

Initialization time for the task name: task 2 = 06 : 13 : 02

Initialization time for the task name: task 3 = 06 : 13 : 02

Time of execution for the task name: task 1 = 06 : 13 : 04

Time of execution for the task name: task 2 = 06 : 13 : 04

Time of execution for the task name: task 3 = 06 : 13 : 04

Time of execution for the task name: task 1 = 06 : 13 : 05

Time of execution for the task name: task 2 = 06 : 13 : 05

Time of execution for the task name: task 3 = 06 : 13 : 05

Time of execution for the task name: task 1 = 06 : 13 : 06

Time of execution for the task name: task 2 = 06 : 13 : 06

Time of execution for the task name: task 3 = 06 : 13 : 06

Time of execution for the task name: task 1 = 06 : 13 : 07

Time of execution for the task name: task 2 = 06 : 13 : 07

Time of execution for the task name: task 3 = 06 : 13 : 07

Time of execution for the task name: task 1 = 06 : 13 : 08

Time of execution for the task name: task 2 = 06 : 13 : 08

Time of execution for the task name: task 3 = 06 : 13 : 08

task 2 is complete.

Initialization time for the task name: task 4 = 06 : 13 : 09

task 1 is complete.

Initialization time for the task name: task 5 = 06 : 13 : 09

task 3 is complete.

Time of execution for the task name: task 4 = 06 : 13 : 10

Time of execution for the task name: task 5 = 06 : 13 : 10

Time of execution for the task name: task 4 = 06 : 13 : 11

Time of execution for the task name: task 5 = 06 : 13 : 11

Time of execution for the task name: task 4 = 06 : 13 : 12

Time of execution for the task name: task 5 = 06 : 13 : 12

Time of execution for the task name: task 4 = 06 : 13 : 13

Time of execution for the task name: task 5 = 06 : 13 : 13

Time of execution for the task name: task 4 = 06 : 13 : 14

Time of execution for the task name: task 5 = 06 : 13 : 14

task 4 is complete.

task 5 is complete.

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

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

Risks involved in Thread Pools

The following are the risk involved in the thread pools.

**Deadlock:** It is a known fact that deadlock can come in any program that involves multithreading, and a thread pool introduces another scenario of deadlock. Consider a scenario where all the threads that are executing are waiting for the results from the threads that are blocked and waiting in the queue because of the non-availability of threads for the execution.

**Thread Leakage:** Leakage of threads occurs when a thread is being removed from the pool to execute a task but is not returning to it after the completion of the task. For example, when a thread throws the exception and the pool class is not able to catch this exception, then the thread exits and reduces the thread pool size by 1. If the same thing repeats a number of times, then there are fair chances that the pool will become empty, and hence, there are no threads available in the pool for executing other requests.

**Resource Thrashing:** A lot of time is wasted in context switching among threads when the size of the thread pool is very large. Whenever there are more threads than the optimal number may cause the starvation problem, and it leads to resource thrashing.

## Points to Remember

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

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

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

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

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

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

## Tuning the Thread Pool

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

## Conclusion

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

# ThreadGroup in Java

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

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

Java thread group is implemented by java.lang.ThreadGroup class.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

## Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

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

## Methods of ThreadGroup class

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

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Modifier and Type** | **Method** | **Description** |
| 1) | void | [checkAccess()](https://www.javatpoint.com/java-threadgroup-checkaccess-method) | This method determines if the currently running thread has permission to modify the thread group. |
| 2) | int | [activeCount()](https://www.javatpoint.com/java-threadgroup-activecount-method) | This method returns an estimate of the number of active threads in the thread group and its subgroups. |
| 3) | int | [activeGroupCount()](https://www.javatpoint.com/java-threadgroup-activegroupcount-method) | This method returns an estimate of the number of active groups in the thread group and its subgroups. |
| 4) | void | [destroy()](https://www.javatpoint.com/java-threadgroup-destroy-method) | This method destroys the thread group and all of its subgroups. |
| 5) | int | [enumerate(Thread[] list)](https://www.javatpoint.com/java-threadgroup-enumerate-method) | This method copies into the specified array every active thread in the thread group and its subgroups. |
| 6) | int | [getMaxPriority()](https://www.javatpoint.com/java-threadgroup-getmaxpriority-method) | This method returns the maximum priority of the thread group. |
| 7) | String | [getName()](https://www.javatpoint.com/java-threadgroup-getname-method) | This method returns the name of the thread group. |
| 8) | ThreadGroup | [getParent()](https://www.javatpoint.com/java-threadgroup-getparent-method) | This method returns the parent of the thread group. |
| 9) | void | [interrupt()](https://www.javatpoint.com/java-threadgroup-interrupt-method) | This method interrupts all threads in the thread group. |
| 10) | boolean | [isDaemon()](https://www.javatpoint.com/java-threadgroup-isdaemon-method) | This method tests if the thread group is a daemon thread group. |
| 11) | void | [setDaemon(boolean daemon)](https://www.javatpoint.com/java-threadgroup-setdaemon-method) | This method changes the daemon status of the thread group. |
| 12) | boolean | [isDestroyed()](https://www.javatpoint.com/java-threadgroup-isdestroyed-method) | This method tests if this thread group has been destroyed. |
| 13) | void | [list()](https://www.javatpoint.com/java-threadgroup-list-method) | This method prints information about the thread group to the standard output. |
| 14) | boolean | [parentOf(ThreadGroup g](https://www.javatpoint.com/java-threadgroup-parentof-method) | This method tests if the thread group is either the thread group argument or one of its ancestor thread groups. |
| 15) | void | [suspend()](https://www.javatpoint.com/java-threadgroup-suspend-method) | This method is used to suspend all threads in the thread group. |
| 16) | void | [resume()](https://www.javatpoint.com/java-threadgroup-resume-method) | This method is used to resume all threads in the thread group which was suspended using suspend() method. |
| 17) | void | [setMaxPriority(int pri)](https://www.javatpoint.com/java-threadgroup-setmaxpriority-method) | This method sets the maximum priority of the group. |
| 18) | void | [stop()](https://www.javatpoint.com/java-threadgroup-stop-method) | This method is used to stop all threads in the thread group. |
| 19) | String | [toString()](https://www.javatpoint.com/java-threadgroup-tostring-method) | This method returns a string representation of the Thread group. |

Let's see a code to group multiple threads.

1. ThreadGroup tg1 = **new** ThreadGroup("Group A");
2. Thread t1 = **new** Thread(tg1,**new** MyRunnable(),"one");
3. Thread t2 = **new** Thread(tg1,**new** MyRunnable(),"two");
4. Thread t3 = **new** Thread(tg1,**new** MyRunnable(),"three");

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

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

## ThreadGroup Example

*File: ThreadGroupDemo.java*

1. **public** **class** ThreadGroupDemo **implements** Runnable{
2. **public** **void** run() {
3. System.out.println(Thread.currentThread().getName());
4. }
5. **public** **static** **void** main(String[] args) {
6. ThreadGroupDemo runnable = **new** ThreadGroupDemo();
7. ThreadGroup tg1 = **new** ThreadGroup("Parent ThreadGroup");
9. Thread t1 = **new** Thread(tg1, runnable,"one");
10. t1.start();
11. Thread t2 = **new** Thread(tg1, runnable,"two");
12. t2.start();
13. Thread t3 = **new** Thread(tg1, runnable,"three");
14. t3.start();
16. System.out.println("Thread Group Name: "+tg1.getName());
17. tg1.list();
19. }
20. }

**Output:**

one

two

three

Thread Group Name: Parent ThreadGroup

java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]

AD

## Thread Pool Methods Example: int activeCount()

Let's see how one can use the method activeCount().

**FileName:** ActiveCountExample.java

1. // code that illustrates the activeCount() method
3. // import statement
4. **import** java.lang.\*;

7. **class** ThreadNew **extends** Thread
8. {
9. // constructor of the  class
10. ThreadNew(String tName, ThreadGroup tgrp)
11. {
12. **super**(tgrp, tName);
13. start();
14. }
16. // overriding the run method
17. **public** **void** run()
18. {
20. **for** (**int** j = 0; j < 1000; j++)
21. {
22. **try**
23. {
24. Thread.sleep(5);
25. }
26. **catch** (InterruptedException e)
27. {
28. System.out.println("The exception has been encountered " + e);
29. }
30. }
31. }
32. }
34. **public** **class** ActiveCountExample
35. {
36. // main method
37. **public** **static** **void** main(String argvs[])
38. {
39. // creating the thread group
40. ThreadGroup tg = **new** ThreadGroup("The parent group of threads");
42. ThreadNew th1 = **new** ThreadNew("first", tg);
43. System.out.println("Starting the first");
45. ThreadNew th2 = **new** ThreadNew("second", tg);
46. System.out.println("Starting the second");
48. // checking the number of active thread by invoking the activeCount() method
49. System.out.println("The total number of active threads are: " + tg.activeCount());
50. }
51. }

**Output:**

Starting the first

Starting the second

The total number of active threads are: 2

## Thread Pool Methods Example: int activeGroupCount()

Now, we will learn how one can use the activeGroupCount() method in the code.

**FileName:** ActiveGroupCountExample.java

1. // Java code illustrating the activeGroupCount() method
3. // import statement
4. **import** java.lang.\*;

7. **class** ThreadNew **extends** Thread
8. {
9. // constructor of the  class
10. ThreadNew(String tName, ThreadGroup tgrp)
11. {
12. **super**(tgrp, tName);
13. start();
14. }
16. // overriding the run() method
17. **public** **void** run()
18. {
20. **for** (**int** j = 0; j < 100; j++)
21. {
22. **try**
23. {
24. Thread.sleep(5);
25. }
26. **catch** (InterruptedException e)
27. {
28. System.out.println("The exception has been encountered " + e);
29. }
31. }
33. System.out.println(Thread.currentThread().getName() + " thread has finished executing");
34. }
35. }
37. **public** **class** ActiveGroupCountExample
38. {
39. // main method
40. **public** **static** **void** main(String argvs[])
41. {
42. // creating the thread group
43. ThreadGroup tg = **new** ThreadGroup("The parent group of threads");
45. ThreadGroup tg1 = **new** ThreadGroup(tg, "the child group");
47. ThreadNew th1 = **new** ThreadNew("the first", tg);
48. System.out.println("Starting the first");
50. ThreadNew th2 = **new** ThreadNew("the second", tg);
51. System.out.println("Starting the second");
53. // checking the number of active thread by invoking the activeGroupCount() method
54. System.out.println("The total number of active thread groups are: " + tg.activeGroupCount());
55. }
56. }

**Output:**

Starting the first

Starting the second

The total number of active thread groups are: 1

the second thread has finished executing

the first thread has finished executing

### Thread Pool Methods Example: void destroy()

Now, we will learn how one can use the destroy() method in the code.

**FileName:** DestroyExample.java

1. // Code illustrating the destroy() method
3. // import statement
4. **import** java.lang.\*;

7. **class** ThreadNew **extends** Thread
8. {
9. // constructor of the  class
10. ThreadNew(String tName, ThreadGroup tgrp)
11. {
12. **super**(tgrp, tName);
13. start();
14. }
16. // overriding the run() method
17. **public** **void** run()
18. {
20. **for** (**int** j = 0; j < 100; j++)
21. {
22. **try**
23. {
24. Thread.sleep(5);
25. }
26. **catch** (InterruptedException e)
27. {
28. System.out.println("The exception has been encountered " + e);
29. }
31. }
33. System.out.println(Thread.currentThread().getName() + " thread has finished executing");
34. }
35. }
37. **public** **class** DestroyExample
38. {
39. // main method
40. **public** **static** **void** main(String argvs[]) **throws** SecurityException, InterruptedException
41. {
42. // creating the thread group
43. ThreadGroup tg = **new** ThreadGroup("the parent group");
45. ThreadGroup tg1 = **new** ThreadGroup(tg, "the child group");
47. ThreadNew th1 = **new** ThreadNew("the first", tg);
48. System.out.println("Starting the first");
50. ThreadNew th2 = **new** ThreadNew("the second", tg);
51. System.out.println("Starting the second");
53. // waiting until the other threads has been finished
54. th1.join();
55. th2.join();
57. // destroying the child thread group
58. tg1.destroy();
59. System.out.println(tg1.getName() + " is destroyed.");
61. // destroying the parent thread group
62. tg.destroy();
63. System.out.println(tg.getName() + " is destroyed.");
64. }
65. }

**Output:**

Starting the first

Starting the second

the first thread has finished executing

the second thread has finished executing

the child group is destroyed.

the parent group is destroyed.

## Thread Pool Methods Example: int enumerate()

Now, we will learn how one can use the enumerate() method in the code.

**FileName:** EnumerateExample.java

1. // Code illustrating the enumerate() method
2. // import statement
3. **import** java.lang.\*;
4. **class** ThreadNew **extends** Thread
5. {
6. // constructor of the class
7. ThreadNew(String tName, ThreadGroup tgrp)
8. {
9. **super**(tgrp, tName);
10. start();
11. }
13. // overriding the run() method
14. **public** **void** run()
15. {
16. **for** (**int** j = 0; j < 100; j++)
17. {
18. **try**
19. {
20. Thread.sleep(5);
21. }
22. **catch** (InterruptedException e)
23. {
24. System.out.println("The exception has been encountered " + e);
25. }
27. }
29. System.out.println(Thread.currentThread().getName() + " thread has finished executing");
30. }
31. }
33. **public** **class** EnumerateExample
34. {
35. // main method
36. **public** **static** **void** main(String argvs[]) **throws** SecurityException, InterruptedException
37. {
38. // creating the thread group
39. ThreadGroup tg = **new** ThreadGroup("the parent group");
41. ThreadGroup tg1 = **new** ThreadGroup(tg, "the child group");
43. ThreadNew th1 = **new** ThreadNew("the first", tg);
44. System.out.println("Starting the first");
46. ThreadNew th2 = **new** ThreadNew("the second", tg);
47. System.out.println("Starting the second");
49. // returning the number of threads kept in this array
50. Thread[] grp = **new** Thread[tg.activeCount()];
51. **int** cnt = tg.enumerate(grp);
52. **for** (**int** j = 0; j < cnt; j++)
53. {
54. System.out.println("Thread " + grp[j].getName() + " is found.");
55. }
56. }
57. }

**Output:**

Starting the first

Starting the second

Thread the first is found.

Thread the second is found.

the first thread has finished executing

the second thread has finished executing

## Thread Pool Methods Example: int getMaxPriority()

The following code shows the working of the getMaxPriority() method.

**FileName:** GetMaxPriorityExample.java

1. // Code illustrating the getMaxPriority() method
3. // import statement
4. **import** java.lang.\*;

7. **class** ThreadNew **extends** Thread
8. {
9. // constructor of the class
10. ThreadNew(String tName, ThreadGroup tgrp)
11. {
12. **super**(tgrp, tName);
13. start();
14. }
16. // overriding the run() method
17. **public** **void** run()
18. {
20. **for** (**int** j = 0; j < 100; j++)
21. {
22. **try**
23. {
24. Thread.sleep(5);
25. }
26. **catch** (InterruptedException e)
27. {
28. System.out.println("The exception has been encountered " + e);
29. }
31. }
33. System.out.println(Thread.currentThread().getName() + " thread has finished executing");
34. }
35. }
37. **public** **class** GetMaxPriorityExample
38. {
39. // main method
40. **public** **static** **void** main(String argvs[]) **throws** SecurityException, InterruptedException
41. {
42. // creating the thread group
43. ThreadGroup tg = **new** ThreadGroup("the parent group");
45. ThreadGroup tg1 = **new** ThreadGroup(tg, "the child group");
47. ThreadNew th1 = **new** ThreadNew("the first", tg);
48. System.out.println("Starting the first");
50. ThreadNew th2 = **new** ThreadNew("the second", tg);
51. System.out.println("Starting the second");
53. **int** priority = tg.getMaxPriority();
55. System.out.println("The maximum priority of the parent ThreadGroup: " + priority);

58. }
59. }

**Output:**

Starting the first

Starting the second

The maximum priority of the parent ThreadGroup: 10

the first thread has finished executing

the second thread has finished executing

## Thread Pool Methods Example: ThreadGroup getParent()

Now, we will learn how one can use the getParent() method in the code.

**FileName:** GetParentExample.java

1. // Code illustrating the getParent() method
3. // import statement
4. **import** java.lang.\*;

7. **class** ThreadNew **extends** Thread
8. {
9. // constructor of the class
10. ThreadNew(String tName, ThreadGroup tgrp)
11. {
12. **super**(tgrp, tName);
13. start();
14. }
16. // overriding the run() method
17. **public** **void** run()
18. {
20. **for** (**int** j = 0; j < 100; j++)
21. {
22. **try**
23. {
24. Thread.sleep(5);
25. }
26. **catch** (InterruptedException e)
27. {
28. System.out.println("The exception has been encountered" + e);
29. }
31. }
33. System.out.println(Thread.currentThread().getName() + " thread has finished executing");
34. }
35. }
37. **public** **class** GetMaxPriorityExample
38. {
39. // main method
40. **public** **static** **void** main(String argvs[]) **throws** SecurityException, InterruptedException
41. {
42. // creating the thread group
43. ThreadGroup tg = **new** ThreadGroup("the parent group");
45. ThreadGroup tg1 = **new** ThreadGroup(tg, "the child group");
47. ThreadNew th1 = **new** ThreadNew("the first", tg);
48. System.out.println("Starting the first");
50. ThreadNew th2 = **new** ThreadNew("the second", tg);
51. System.out.println("Starting the second");
53. // printing the parent ThreadGroup
54. // of both child and parent threads
55. System.out.println("The ParentThreadGroup for " + tg.getName() + " is " + tg.getParent().getName());
56. System.out.println("The ParentThreadGroup for " + tg1.getName() + " is " + tg1.getParent().getName());

59. }
60. }

**Output:**

Starting the first

Starting the second

The ParentThreadGroup for the parent group is main

The ParentThreadGroup for the child group is the parent group

the first thread has finished executing

the second thread has finished executing

## Thread Pool Methods Example: void interrupt()

The following program illustrates how one can use the interrupt() method.

**FileName:** InterruptExample.java

1. // Code illustrating the interrupt() method
3. // import statement
4. **import** java.lang.\*;

7. **class** ThreadNew **extends** Thread
8. {
9. // constructor of the class
10. ThreadNew(String tName, ThreadGroup tgrp)
11. {
12. **super**(tgrp, tName);
13. start();
14. }
16. // overriding the run() method
17. **public** **void** run()
18. {
20. **for** (**int** j = 0; j < 100; j++)
21. {
22. **try**
23. {
24. Thread.sleep(5);
25. }
26. **catch** (InterruptedException e)
27. {
28. System.out.println("The exception has been encountered " + e);
29. }
31. }
33. System.out.println(Thread.currentThread().getName() + " thread has finished executing");
34. }
35. }
37. **public** **class** InterruptExample
38. {
39. // main method
40. **public** **static** **void** main(String argvs[]) **throws** SecurityException, InterruptedException
41. {
42. // creating the thread group
43. ThreadGroup tg = **new** ThreadGroup("the parent group");
45. ThreadGroup tg1 = **new** ThreadGroup(tg, "the child group");
47. ThreadNew th1 = **new** ThreadNew("the first", tg);
48. System.out.println("Starting the first");
50. ThreadNew th2 = **new** ThreadNew("the second", tg);
51. System.out.println("Starting the second");
53. // invoking the interrupt method
54. tg.interrupt();
56. }
57. }

**Output:**

Starting the first

Starting the second

The exception has been encountered java.lang.InterruptedException: sleep interrupted

The exception has been encountered java.lang.InterruptedException: sleep interrupted

the second thread has finished executing

the first thread has finished executing

## Thread Pool Methods Example: boolean isDaemon()

The following program illustrates how one can use the isDaemon() method.

**FileName:** IsDaemonExample.java

1. // Code illustrating the isDaemon() method
3. // import statement
4. **import** java.lang.\*;

7. **class** ThreadNew **extends** Thread
8. {
9. // constructor of the class
10. ThreadNew(String tName, ThreadGroup tgrp)
11. {
12. **super**(tgrp, tName);
13. start();
14. }
16. // overriding the run() method
17. **public** **void** run()
18. {
20. **for** (**int** j = 0; j < 100; j++)
21. {
22. **try**
23. {
24. Thread.sleep(5);
25. }
26. **catch** (InterruptedException e)
27. {
28. System.out.println("The exception has been encountered" + e);
29. }
31. }
33. System.out.println(Thread.currentThread().getName() + " thread has finished executing");
34. }
35. }
37. **public** **class** IsDaemonExample
38. {
39. // main method
40. **public** **static** **void** main(String argvs[]) **throws** SecurityException, InterruptedException
41. {
42. // creating the thread group
43. ThreadGroup tg = **new** ThreadGroup("the parent group");
45. ThreadGroup tg1 = **new** ThreadGroup(tg, "the child group");
47. ThreadNew th1 = **new** ThreadNew("the first", tg);
48. System.out.println("Starting the first");
50. ThreadNew th2 = **new** ThreadNew("the second", tg);
51. System.out.println("Starting the second");
53. **if** (tg.isDaemon() == **true**)
54. {
55. System.out.println("The group is a daemon group.");
56. }
57. **else**
58. {
59. System.out.println("The group is not a daemon group.");
60. }
62. }
63. }

**Output:**

Starting the first

Starting the second

The group is not a daemon group.

the second thread has finished executing

the first thread has finished executing

## Thread Pool Methods Example: boolean isDestroyed()

The following program illustrates how one can use the isDestroyed() method.

**FileName:** IsDestroyedExample.java

1. // Code illustrating the isDestroyed() method
3. // import statement
4. **import** java.lang.\*;

7. **class** ThreadNew **extends** Thread
8. {
9. // constructor of the class
10. ThreadNew(String tName, ThreadGroup tgrp)
11. {
12. **super**(tgrp, tName);
13. start();
14. }
16. // overriding the run() method
17. **public** **void** run()
18. {
20. **for** (**int** j = 0; j < 100; j++)
21. {
22. **try**
23. {
24. Thread.sleep(5);
25. }
26. **catch** (InterruptedException e)
27. {
28. System.out.println("The exception has been encountered" + e);
29. }
31. }
33. System.out.println(Thread.currentThread().getName() + " thread has finished executing");
34. }
35. }
37. **public** **class** IsDestroyedExample
38. {
39. // main method
40. **public** **static** **void** main(String argvs[]) **throws** SecurityException, InterruptedException
41. {
42. // creating the thread group
43. ThreadGroup tg = **new** ThreadGroup("the parent group");
45. ThreadGroup tg1 = **new** ThreadGroup(tg, "the child group");
47. ThreadNew th1 = **new** ThreadNew("the first", tg);
48. System.out.println("Starting the first");
50. ThreadNew th2 = **new** ThreadNew("the second", tg);
51. System.out.println("Starting the second");
53. **if** (tg.isDestroyed() == **true**)
54. {
55. System.out.println("The group has been destroyed.");
56. }
57. **else**
58. {
59. System.out.println("The group has not been destroyed.");
60. }
62. }
63. }

**Output:**

Starting the first

Starting the second

The group has not been destroyed.

the first thread has finished executing

the second thread has finished executing

AD

# Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

### Advantage of Garbage Collection

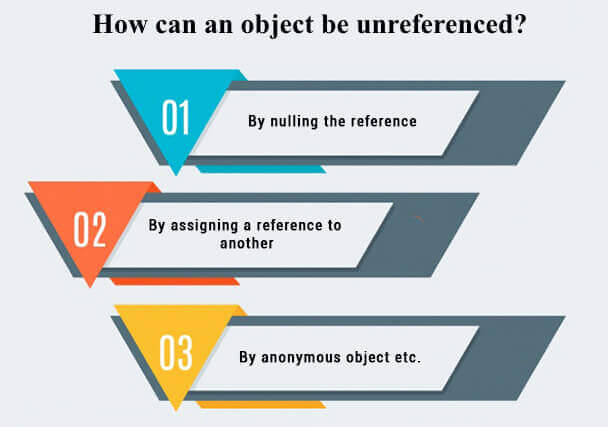
* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

## How can an object be unreferenced?

There are many ways:

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

AD



### 1) By nulling a reference:

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

### 2) By assigning a reference to another:

1. Employee e1=**new** Employee();
2. Employee e2=**new** Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection

### 3) By anonymous object:

1. **new** Employee();

## finalize() method

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

1. **protected** **void** finalize(){}

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

## gc() method

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

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

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

### Simple Example of garbage collection in java

1. **public** **class** TestGarbage1{
2. **public** **void** finalize(){System.out.println("object is garbage collected");}
3. **public** **static** **void** main(String args[]){
4. TestGarbage1 s1=**new** TestGarbage1();
5. TestGarbage1 s2=**new** TestGarbage1();
6. s1=**null**;
7. s2=**null**;
8. System.gc();
9. }
10. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestGarbage1)

object is garbage collected

object is garbage collected

#### Note: Neither finalization nor garbage collection is guaranteed.