**Multithreading in Java**

**Multithreading in java** is a process of executing multiple threads simultaneously.

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

But we use multithreading than multiprocessing because threads share a common 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 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 exception occur 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 by two ways:

* Process-based Multitasking(Multiprocessing)
* Thread-based Multitasking(Multithreading)

### 1) Process-based Multitasking (Multiprocessing)

* Each process have its own address in memory i.e. each process allocates separate memory area.
* Process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

### 2) Thread-based Multitasking (Multithreading)

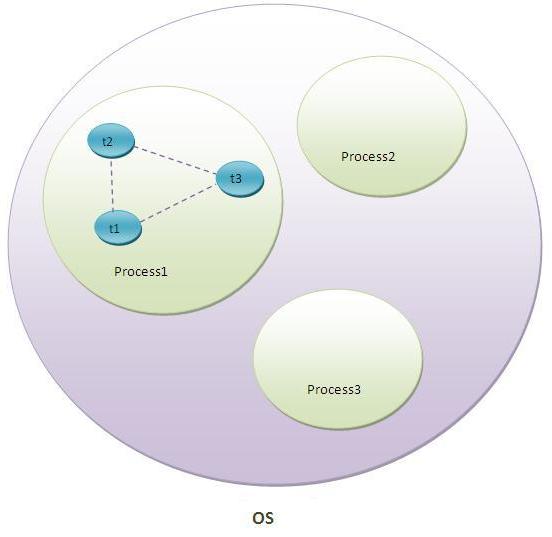
* Threads share the same address space.
* Thread is lightweight.
* Cost of communication between the thread is low.

#### Note: At least one process is required for each thread.

## What is Thread in java

A thread is a lightweight sub process, a 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 shares a common memory area



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

#### Note: At a time one thread is executed only.

Do You Know

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

What we will learn in Multithreading

* Multithreading
* Life Cycle of a Thread
* Two ways to create a Thread
* How to perform multiple tasks by multiple threads
* Thread Schedular
* Sleeping a thread
* Can we start a thread twice ?
* What happens if we call the run() method instead of start() method ?
* Joining a thread
* Naming a thread
* Priority of a thread
* Daemon Thread
* ShutdownHook
* Garbage collection
* Synchronization with synchronized method
* Synchronized block
* Static synchronization
* Deadlock
* Inter-thread communication

# 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   thread life cycle in java 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) |

### Commonly used methods of Thread class:

|  |
| --- |
| 1. **public void run():** is used to perform action for a thread. 2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread. 3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. 4. **public void join():** waits for a thread to die. 5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds. 6. **public int getPriority():** returns the priority of the thread. 7. **public int setPriority(int priority):** changes the priority of the thread. 8. **public String getName():** returns the name of the thread. 9. **public void setName(String name):** changes the name of the thread. 10. **public Thread currentThread():** returns the reference of currently executing thread. 11. **public int getId():** returns the id of the thread. 12. **public Thread.State getState():** returns the state of the thread. 13. **public boolean isAlive():** tests if the thread is alive. 14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public void suspend():** is used to suspend the thread(depricated). 16. **public void resume():** is used to resume the suspended thread(depricated). 17. **public void stop():** is used to stop the thread(depricated). 18. **public boolean isDaemon():** tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread. 20. **public void interrupt():** interrupts the thread. 21. **public boolean isInterrupted():** tests if the thread has been interrupted. 22. **public static boolean interrupted():** tests if the current thread has been interrupted. |

### Runnable interface:

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

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

### Starting a thread:

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:   * A new thread starts(with new callstack). * The thread moves from New state to the Runnable state. * When the thread gets a chance to execute, its target run() method will run. |

### 1) Java Thread Example by extending Thread class

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

Output:thread is running...

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

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

Output:thread is running...

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

# 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

## Example of sleep method in java

1. class TestSleepMethod1 extends Thread{
2. public void run(){
3. for(int i=1;i<5;i++){
4. try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. public static void main(String args[]){
9. TestSleepMethod1 t1=new TestSleepMethod1();
10. TestSleepMethod1 t2=new TestSleepMethod1();
12. t1.start();
13. t2.start();
14. }
15. }

Output:

1

1

2

2

3

3

4

4

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

**Can we start a thread twice**

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

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

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

running

Exception in thread "main" java.lang.IllegalThreadStateException

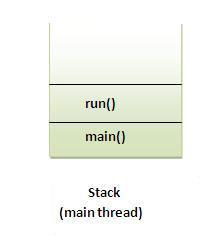
**What if we call run() method directly instead start() method?**

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

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

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

Output:running...

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

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

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

Output:1

2

3

4

5

1

2

3

4

5

|  |
| --- |
| As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object. |

# The join() method

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

### Syntax:

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

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

1. class TestJoinMethod1 extends Thread{
2. public void run(){
3. for(int i=1;i<=5;i++){
4. try{
5. Thread.sleep(500);
6. }catch(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. public static void main(String args[]){
11. TestJoinMethod1 t1=new TestJoinMethod1();
12. TestJoinMethod1 t2=new TestJoinMethod1();
13. TestJoinMethod1 t3=new TestJoinMethod1();
14. t1.start();
15. try{
16. t1.join();
17. }catch(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

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

Output:1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

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

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

1. class TestJoinMethod2 extends Thread{
2. public void run(){
3. for(int i=1;i<=5;i++){
4. try{
5. Thread.sleep(500);
6. }catch(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. public static void main(String args[]){
11. TestJoinMethod2 t1=new TestJoinMethod2();
12. TestJoinMethod2 t2=new TestJoinMethod2();
13. TestJoinMethod2 t3=new TestJoinMethod2();
14. t1.start();
15. try{
16. t1.join(1500);
17. }catch(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

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

Output:1

2

3

1

4

1

2

5

2

3

3

4

4

5

5

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

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

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

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

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

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changling name of t1:Sonoo Jaiswal

running...

### The currentThread() method:

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

### Syntax:

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

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

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

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

Output:Thread-0

Thread-1

# 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

1. class TestMultiNaming1 extends Thread{
2. public void run(){
3. System.out.println("running...");
4. }
5. public static void main(String args[]){
6. TestMultiNaming1 t1=new TestMultiNaming1();
7. TestMultiNaming1 t2=new TestMultiNaming1();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
11. t1.start();
12. t2.start();
14. t1.setName("Sonoo Jaiswal");
15. System.out.println("After changing name of t1:"+t1.getName());
16. }
17. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiNaming1)

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.

1. public static Thread currentThread()

### Example of currentThread() method

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

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiNaming2)

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 defiend in Thread class:

|  |
| --- |
| 1. public static int MIN\_PRIORITY 2. public static int NORM\_PRIORITY 3. public static int MAX\_PRIORITY |

|  |
| --- |
| Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10. |

### Example of priority of a Thread:

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

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

Output:running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1

# 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](http://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](http://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 reuse many times.

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

#### Advantage of Java Thread Pool

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

#### Real time usage

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

#### Example of Java Thread Pool

Let's see a simple example of 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: JavaThreadPoolExample.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. }

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

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

# ThreadGroup in Java

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

## 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 given parent group and name. |

## Important methods of ThreadGroup class

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

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | int activeCount() | returns no. of threads running in current group. |
| 2) | int activeGroupCount() | returns a no. of active group in this thread group. |
| 3) | void destroy() | destroys this thread group and all its sub groups. |
| 4) | String getName() | returns the name of this group. |
| 5) | ThreadGroup getParent() | returns the parent of this group. |
| 6) | void interrupt() | interrupts all threads of this group. |
| 7) | void list() | prints information of this group to standard console. |

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]

Thread[one,5,Parent ThreadGroup]

Thread[two,5,Parent ThreadGroup]

Thread[three,5,Parent ThreadGroup]

# Java Shutdown Hook

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

### When does the JVM shut down?

The JVM shuts down when:

* user presses ctrl+c on the command prompt
* System.exit(int) method is invoked
* user logoff
* user shutdown etc.

#### The addShutdownHook(Thread hook) method

The addShutdownHook() method of Runtime class is used to register the thread with the Virtual Machine. Syntax:

1. public void addShutdownHook(Thread hook){}

The object of Runtime class can be obtained by calling the static factory method getRuntime(). For example:

Runtime r = Runtime.getRuntime();

#### Factory method

The method that returns the instance of a class is known as factory method.

### Simple example of Shutdown Hook

1. class MyThread extends Thread{
2. public void run(){
3. System.out.println("shut down hook task completed..");
4. }
5. }
7. public class TestShutdown1{
8. public static void main(String[] args)throws Exception {
10. Runtime r=Runtime.getRuntime();
11. r.addShutdownHook(new MyThread());
13. System.out.println("Now main sleeping... press ctrl+c to exit");
14. try{Thread.sleep(3000);}catch (Exception e) {}
15. }
16. }

Output:Now main sleeping... press ctrl+c to exit

shut down hook task completed..

#### Note: The shutdown sequence can be stopped by invoking the halt(int) method of Runtime class.

### Same example of Shutdown Hook by annonymous class:

1. public class TestShutdown2{
2. public static void main(String[] args)throws Exception {
4. Runtime r=Runtime.getRuntime();
6. r.addShutdownHook(new Thread(){
7. public void run(){
8. System.out.println("shut down hook task completed..");
9. }
10. }
11. );
13. System.out.println("Now main sleeping... press ctrl+c to exit");
14. try{Thread.sleep(3000);}catch (Exception e) {}
15. }
16. }

Output:Now main sleeping... press ctrl+c to exit

shut down hook task completed..

# How to perform single task by multiple threads?

|  |
| --- |
| If you have to perform single task by many threads, have only one run() method.For example: |

***Program of performing single task by multiple threads***

1. class TestMultitasking1 extends Thread{
2. public void run(){
3. System.out.println("task one");
4. }
5. public static void main(String args[]){
6. TestMultitasking1 t1=new TestMultitasking1();
7. TestMultitasking1 t2=new TestMultitasking1();
8. TestMultitasking1 t3=new TestMultitasking1();
10. t1.start();
11. t2.start();
12. t3.start();
13. }
14. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestMultitasking1)

Output:task one

task one

task one

***Program of performing single task by multiple threads***

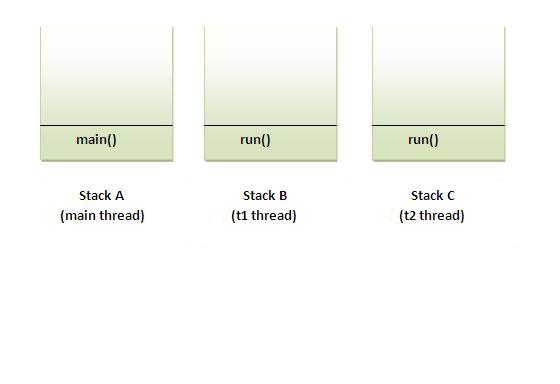
1. class TestMultitasking2 implements Runnable{
2. public void run(){
3. System.out.println("task one");
4. }
6. public static void main(String args[]){
7. Thread t1 =new Thread(new TestMultitasking2());//passing annonymous object of TestMultitasking2 class
8. Thread t2 =new Thread(new TestMultitasking2());
10. t1.start();
11. t2.start();
13. }
14. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestMultitasking2)

Output:task one

task one

#### Note: Each thread run in a separate callstack.



### How to perform multiple tasks by multiple threads (multitasking in multithreading)?

|  |
| --- |
| If you have to perform multiple tasks by multiple threads,have multiple run() methods.For example: |

***Program of performing two tasks by two threads***

1. class Simple1 extends Thread{
2. public void run(){
3. System.out.println("task one");
4. }
5. }
7. class Simple2 extends Thread{
8. public void run(){
9. System.out.println("task two");
10. }
11. }
13. class TestMultitasking3{
14. public static void main(String args[]){
15. Simple1 t1=new Simple1();
16. Simple2 t2=new Simple2();
18. t1.start();
19. t2.start();
20. }
21. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestMultitasking3)

Output:task one

task two

### Same example as above by annonymous class that extends Thread class:

***Program of performing two tasks by two threads***

1. class TestMultitasking4{
2. public static void main(String args[]){
3. Thread t1=new Thread(){
4. public void run(){
5. System.out.println("task one");
6. }
7. };
8. Thread t2=new Thread(){
9. public void run(){
10. System.out.println("task two");
11. }
12. };

15. t1.start();
16. t2.start();
17. }
18. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestMultitasking4)

Output:task one

task two

### Same example as above by annonymous class that implements Runnable interface:

***Program of performing two tasks by two threads***

1. class TestMultitasking5{
2. public static void main(String args[]){
3. Runnable r1=new Runnable(){
4. public void run(){
5. System.out.println("task one");
6. }
7. };
9. Runnable r2=new Runnable(){
10. public void run(){
11. System.out.println("task two");
12. }
13. };
15. Thread t1=new Thread(r1);
16. Thread t2=new Thread(r2);
18. t1.start();
19. t2.start();
20. }
21. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestMultitasking5)

Output:task one

task two

# 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 annonymous object etc.

### 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 annonymous 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](http://www.javatpoint.com/opr/test.jsp?filename=TestGarbage1)

object is garbage collected

object is garbage collected

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

# Java Runtime class

**Java Runtime** class is used to interact with java runtime environment. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of java.lang.Runtime class is available for one java application.

The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

## Important methods of Java Runtime class

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public static Runtime getRuntime() | returns the instance of Runtime class. |
| 2) | public void exit(int status) | terminates the current virtual machine. |
| 3) | public void addShutdownHook(Thread hook) | registers new hook thread. |
| 4) | public Process exec(String command)throws IOException | executes given command in a separate process. |
| 5) | public int availableProcessors() | returns no. of available processors. |
| 6) | public long freeMemory() | returns amount of free memory in JVM. |
| 7) | public long totalMemory() | returns amount of total memory in JVM. |

## Java Runtime exec() method

1. public class Runtime1{
2. public static void main(String args[])throws Exception{
3. Runtime.getRuntime().exec("notepad");//will open a new notepad
4. }
5. }

## How to shutdown system in Java

You can use shutdown -s command to shutdown system. For windows OS, you need to provide full path of shutdown command e.g. c:\\Windows\\System32\\shutdown.

Here you can use -s switch to shutdown system, -r switch to restart system and -t switch to specify time delay.

1. public class Runtime2{
2. public static void main(String args[])throws Exception{
3. Runtime.getRuntime().exec("shutdown -s -t 0");
4. }
5. }

## How to shutdown windows system in Java

1. public class Runtime2{
2. public static void main(String args[])throws Exception{
3. Runtime.getRuntime().exec("c:\\Windows\\System32\\shutdown -s -t 0");
4. }
5. }

## How to restart system in Java

1. public class Runtime3{
2. public static void main(String args[])throws Exception{
3. Runtime.getRuntime().exec("shutdown -r -t 0");
4. }
5. }

## Java Runtime availableProcessors()

1. public class Runtime4{
2. public static void main(String args[])throws Exception{
3. System.out.println(Runtime.getRuntime().availableProcessors());
4. }
5. }

## Java Runtime freeMemory() and totalMemory() method

In the given program, after creating 10000 instance, free memory will be less than the previous free memory. But after gc() call, you will get more free memory.

1. public class MemoryTest{
2. public static void main(String args[])throws Exception{
3. Runtime r=Runtime.getRuntime();
4. System.out.println("Total Memory: "+r.totalMemory());
5. System.out.println("Free Memory: "+r.freeMemory());
7. for(int i=0;i<10000;i++){
8. new MemoryTest();
9. }
10. System.out.println("After creating 10000 instance, Free Memory: "+r.freeMemory());
11. System.gc();
12. System.out.println("After gc(), Free Memory: "+r.freeMemory());
13. }
14. }

Total Memory: 100139008

Free Memory: 99474824

After creating 10000 instance, Free Memory: 99310552

After gc(), Free Memory: 100182832