

Multithreading in Java

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

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

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

- 1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
- 2) You can perform many operations together, so it saves time.
- 3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

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

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading **registers**, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.

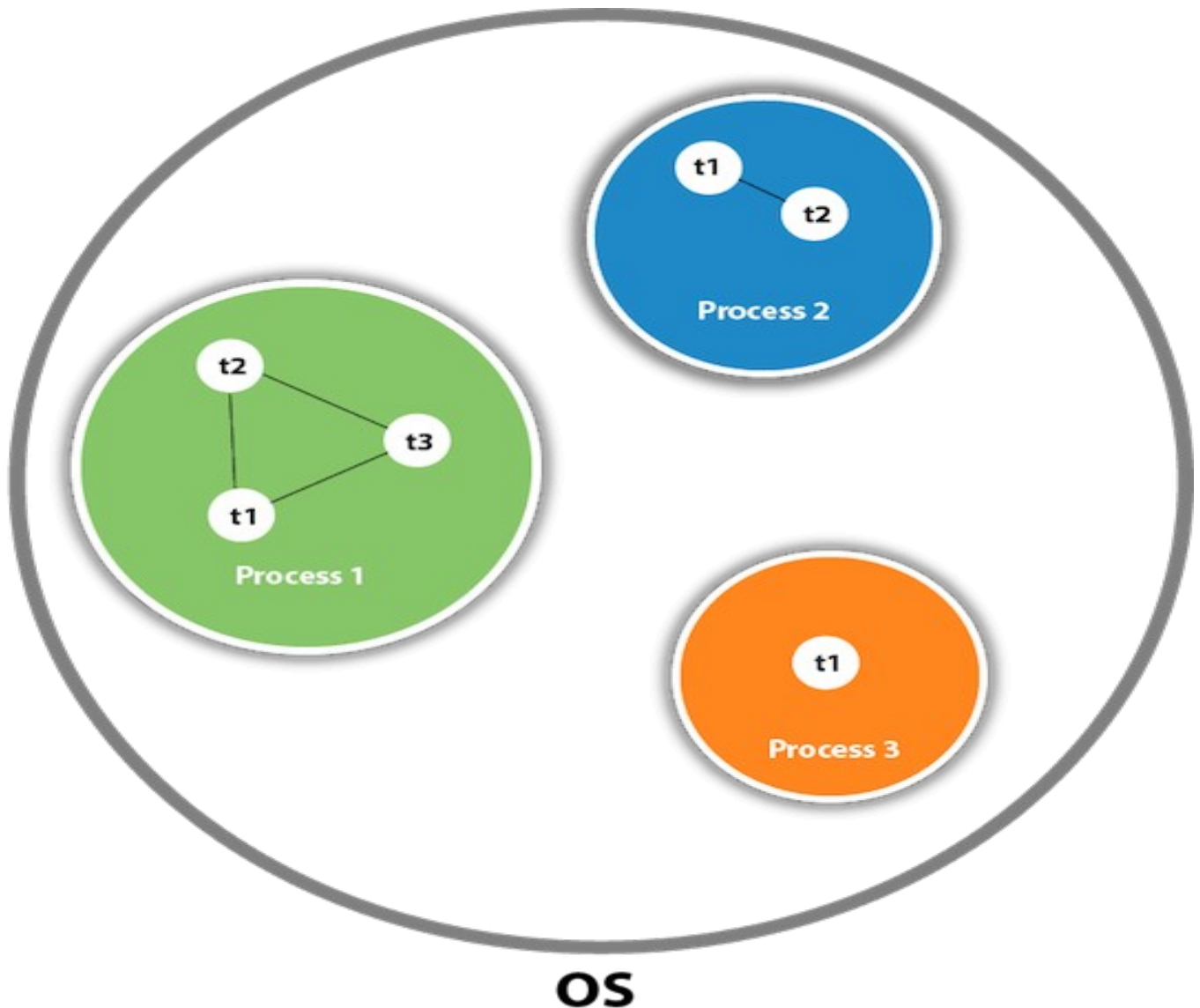
oCost of communication between the thread is low.

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

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.



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

Note: At a time one thread is executed only.

Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides **constructors** and methods to create and perform operations on a thread. Thread class extends **Object class** and implements Runnable interface.

Java Thread Methods

S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13	void	suspend()	It is used to suspend the thread.

14)	void	<code>resume()</code>	It is used to resume the suspended thread.
15)	void	<code>stop()</code>	It is used to stop the thread.
16)	void	<code>destroy()</code>	It is used to destroy the thread group and all of its subgroups.
17)	boolean	<code>isDaemon()</code>	It tests if the thread is a daemon thread.
18)	void	<code>setDaemon()</code>	It marks the thread as daemon or user thread.
19)	void	<code>interrupt()</code>	It interrupts the thread.
20)	boolean	<code>isinterrupted()</code>	It tests whether the thread has been interrupted.
21)	static boolean	<code>interrupted()</code>	It tests whether the current thread has been interrupted.
22)	static int	<code>activeCount()</code>	It returns the number of active threads in the current thread's thread group.
23)	void	<code>checkAccess()</code>	It determines if the currently running thread has permission to modify the thread.
24)	static boolean	<code>holdLock()</code>	It returns true if and only if the current thread holds the monitor lock on the specified object.
25)	static void	<code>dumpStack()</code>	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	<code>getStackTrace()</code>	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	<code>enumerate()</code>	It is used to copy every active thread's thread group and its subgroup into the specified array.

28)	Thread.State	<code>getState()</code>	It is used to return the state of the thread.
29)	ThreadGroup	<code>getThreadGroup()</code>	It is used to return the thread group to which this thread belongs
30)	String	<code>toString()</code>	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.
31)	void	<code>notify()</code>	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	<code>notifyAll()</code>	It is used to give the notification to all waiting threads of a particular object.
33)	void	<code>setContextClassLoader()</code>	It sets the context ClassLoader for the Thread.
34)	ClassLoader	<code>getContextClassLoader()</code>	It returns the context ClassLoader for the thread.
35)	static Thread.UncaughtExceptionHandler	<code>getDefaultUncaughtExceptionHandler()</code>	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	<code>setDefaultUncaughtExceptionHandler()</code>	It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception.

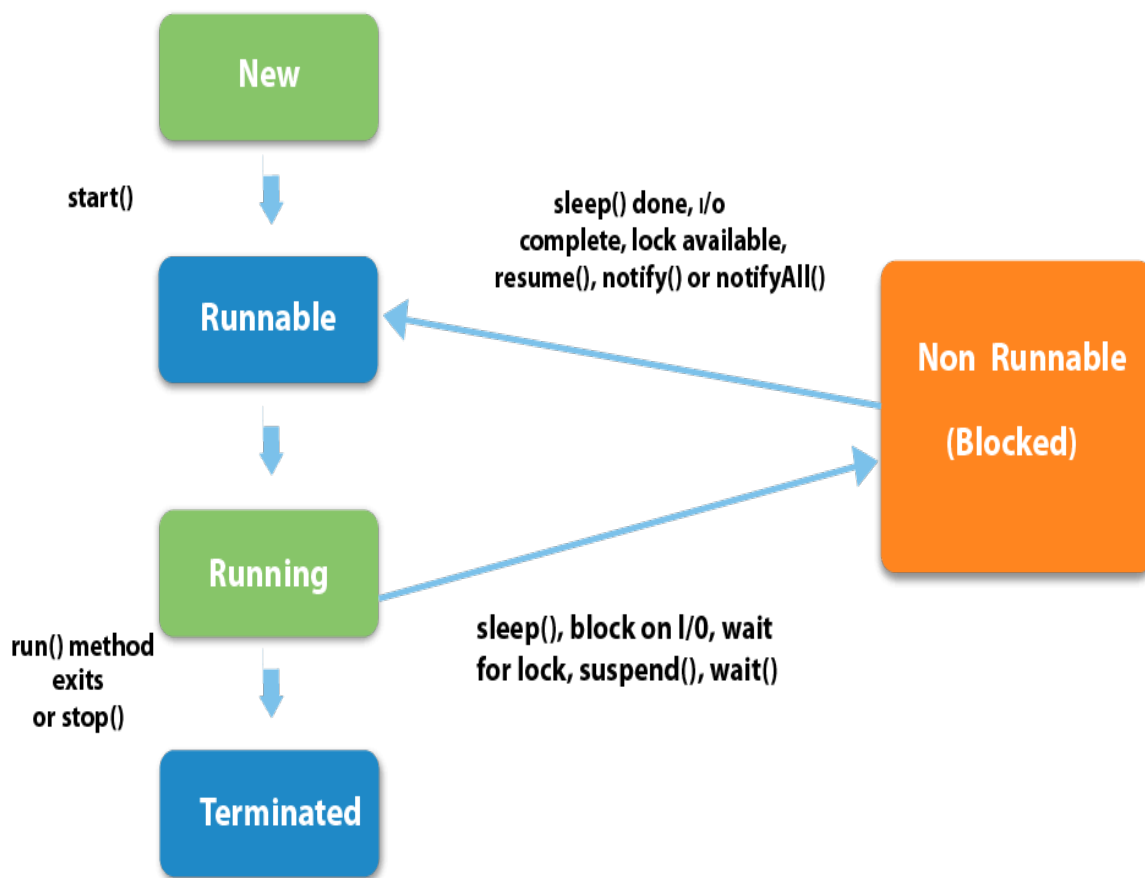
Life cycle of a Thread (Thread States)

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

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

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

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



1) New

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

2) Runnable

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

3) Running

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

4) Non-Runnable (Blocked)

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

5) Terminated

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

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 milliseconds): 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 milliseconds): waits for a thread to die for the specified milliseconds.

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(deprecated).
16. `public void resume():` is used to resume the suspended thread(deprecated).
17. `public void stop():` is used to stop the thread(deprecated).
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. }
5.
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 explicitly 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.

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 milliseconds) throws InterruptedException
- public static void sleep(long milliseconds, 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();
11.
12. t1.start();
13. t2.start();
14. }
15. }
```

Output:

1
1
2
2
3
3
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 scheduler 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 do 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

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.

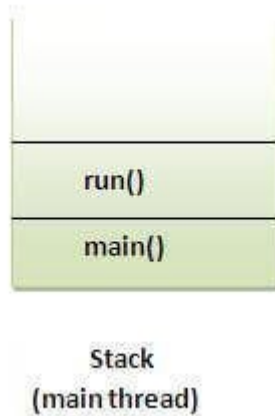
```
1. class TestCallRun1 extends Thread{
2. public void run(){
3. System.out.println("running...");
4. }
5. public static void main(String args[]){
6. TestCallRun1 t1=new TestCallRun1();
7. t1.run();//fine, but does not start a separate call stack
```

8. }

9. }

Test it Now

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();
- 11.
12. t1.run();
13. t2.run();
14. }
15. }

Test it Now

Output:1

2

3

4

5

1



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);}
18.
19.     t2.start();
20.     t3.start();
21. }
```

22.}

Test it Now

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 milliseconds) 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);}
18.
19. t2.start();
20. t3.start();
21. }
22.}

```

Test it Now

```

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.

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());
11.
12. t1.start();
13. t2.start();
14.
15. t1.setName("Sonoo Jaiswal");
16. System.out.println("After changing name of t1:"+t1.getName());
17. }
18.}

```

Test it Now

```

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();
9.
10.  t1.start();
11.  t2.start();
12. }
13.}

```

Test it Now

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 scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. `public static int MIN_PRIORITY`
2. `public static int NORM_PRIORITY`
3. `public static int MAX_PRIORITY`

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```

1. class TestMultiPriority1 extends Thread{
2.   public void run(){
3.     System.out.println("running thread name is:"+Thread.currentThread().getName());
4.     System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
5.
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();
14.
15. }
16. }
```

Test it Now

```
Output:running thread name is:Thread-0
```

```
        running thread priority is:10
```

```
        running thread name is:Thread-1
```

```
        running thread priority is:1
```

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.

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,	creates a thread group with given parent

	String name)	group and name.
--	--------------	-----------------

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.

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

```
object is garbage collected
```

```
object is garbage collected
```

Note: Neither finalization nor garbage collection is guaranteed.