Multithreading in JAVA

Dr. Mainak Adhikari

Assistant Professor,
Department of Computer Science,
IIIT Lucknow, Lucknow, India

Multithreading in JAVA

- Multithreading 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:

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

Multitasking in JAVA

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.

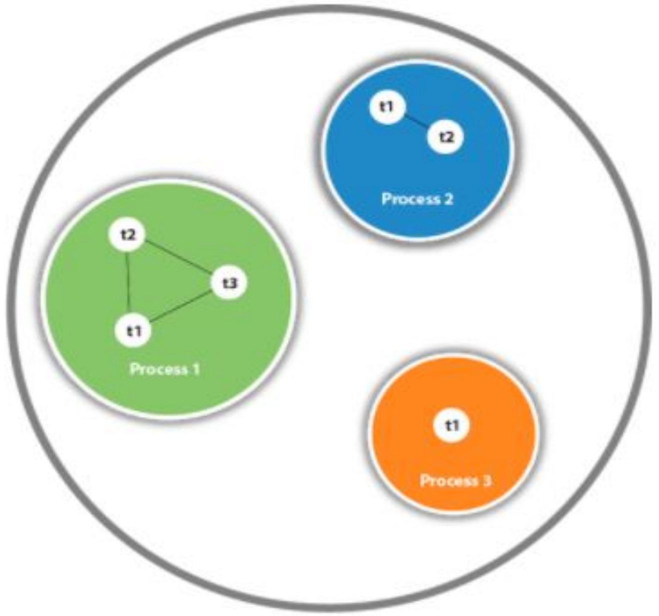
2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

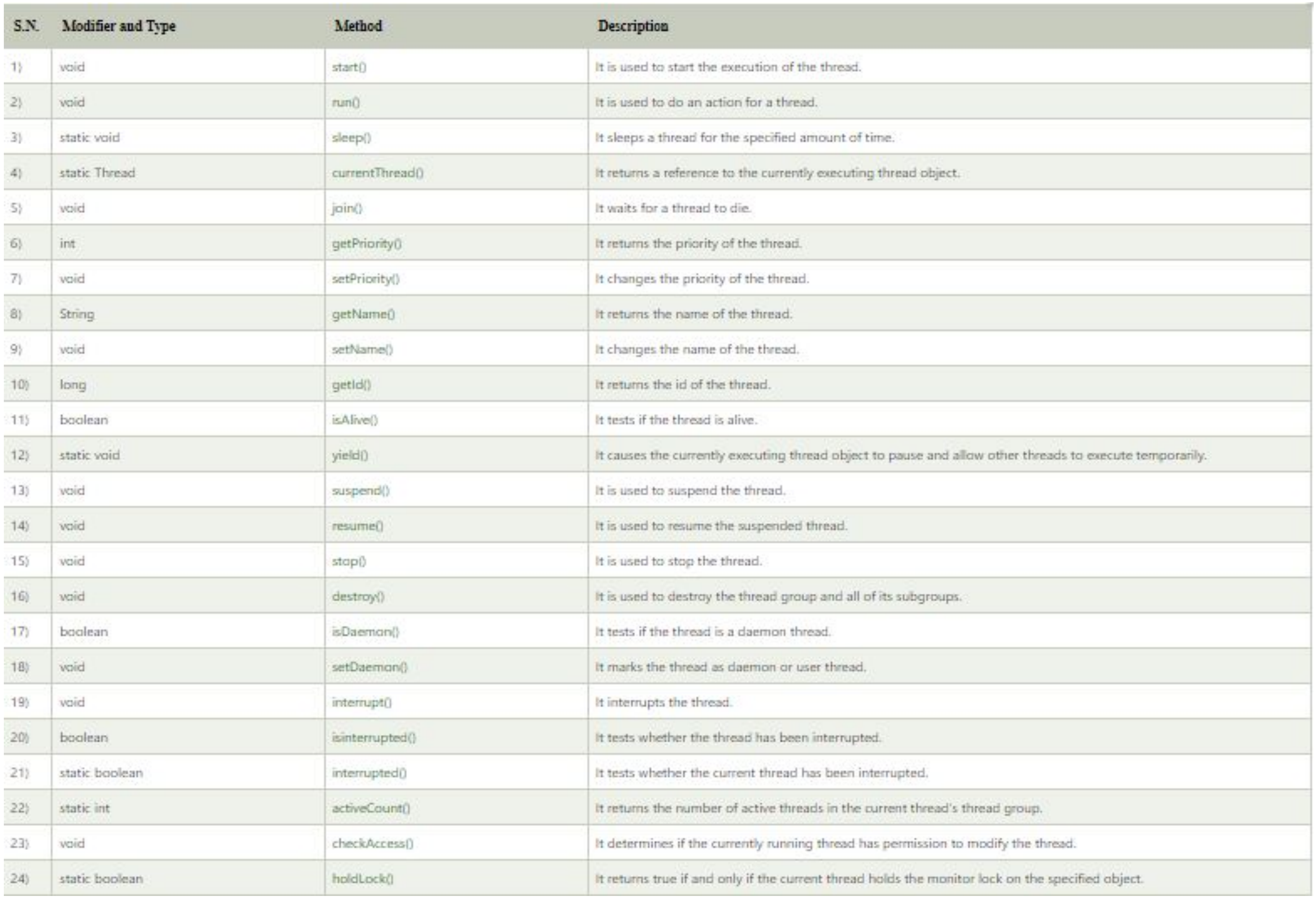
Thread in Java

- 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 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.
- **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

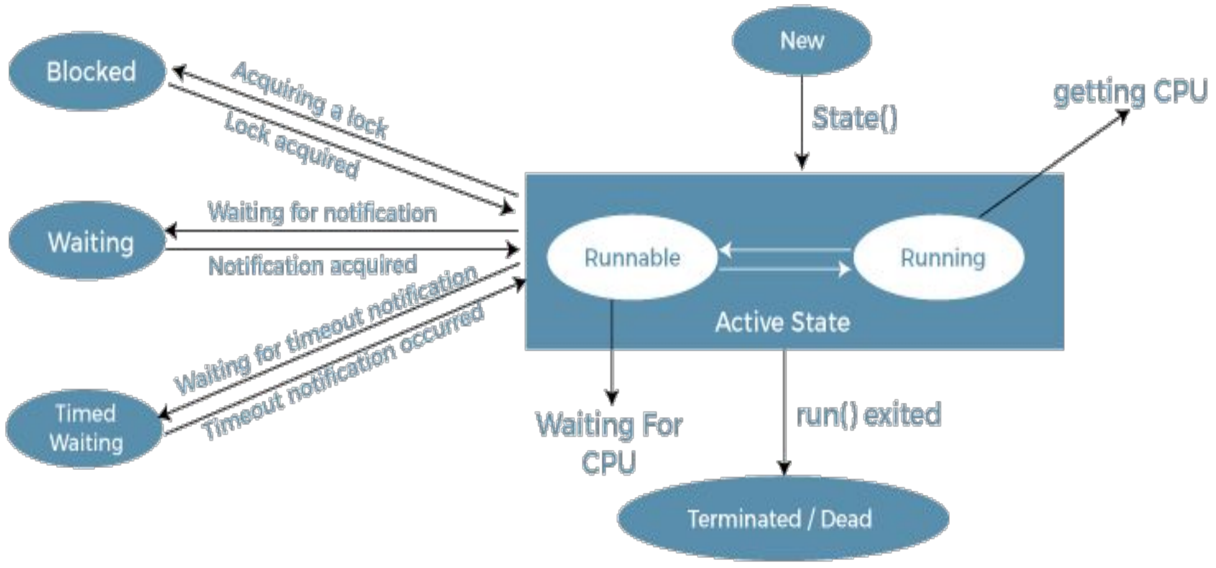


S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the 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	resume()	It is used to resume the suspended thread.
15)	void	stop()	It is used to stop the thread.
16)	void	destroy()	It is used to destroy the thread group and all of its subgroups.
17)	boolean	isDaemon()	It tests if the thread is a daemon thread.
18)	void	setDaemon()	It marks the thread as daemon or user thread.
19)	void	interrupt()	It interrupts the thread.
20)	boolean	isinterrupted()	It tests whether the thread has been interrupted.
21)	static boolean	interrupted()	It tests whether the current thread has been interrupted.
22)	static int	activeCount()	It returns the number of active threads in the current thread's thread group.
23)	void	checkAccess()	It determines if the currently running thread has permission to modify the thread.
24)	static boolean	holdLock()	It returns true if and only if the current thread holds the monitor lock on the specified object.

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states.

- **New**
- **Active**
- **Blocked / Waiting**
- **Timed Waiting**
- **Terminated**



Life Cycle of a Thread

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.

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.

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:

public static final Thread.State NEW
It represents the first state of a thread that is the NEW state.

public static final Thread.State RUNNABLE
It represents the runnable state.It means a thread is waiting in the queue to run.

public static final Thread.State BLOCKED
It represents the blocked state. In this state, the thread is waiting to acquire a lock.

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.

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.

Implementation of Thread States (Contd.)

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

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

```java
// ABC class implements the interface Runnable
class ABC implements Runnable
{
public void run()
{

// try-catch block
try
{
// moving thread t2 to the state timed waiting
Thread.sleep(100);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}


System.out.println("The state of thread t1 while it invoked the method join() on thread t2 -"+ ThreadState.t1.getState());

// try-catch block
try
{
Thread.sleep(200);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
}
}
```

```java
// ThreadState class implements the interface Runnable
public class ThreadState implements Runnable
{
public static Thread t1;
public static ThreadState obj;

// main method
public static void main(String argvs[])
{
// creating an object of the class ThreadState
obj = new ThreadState();
t1 = new Thread(obj);

// thread t1 is spawned
// The thread t1 is currently in the NEW state.
System.out.println("The state of thread t1 after spawning it - " + t1.getState());

// invoking the start() method on
// the thread t1
t1.start();

// thread t1 is moved to the Runnable state
System.out.println("The state of thread t1 after invoking the method start() on it - " + t1.getState());
}

public void run()
{
ABC myObj = new ABC();
Thread t2 = new Thread(myObj);

// thread t2 is created and is currently in the NEW state.
System.out.println("The state of thread t2 after spawning it - "+ t2.getState());
t2.start();
```

```java
// thread t2 is moved to the runnable state
System.out.println("the state of thread t2 after calling the method start() on it - " + t2.getState());

// try-catch block for the smooth flow of the  program
try
{
// moving the thread t1 to the state timed waiting
Thread.sleep(200);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}

System.out.println("The state of thread t2 after invoking the method sleep() on it - "+ t2.getState() );

// try-catch block for the smooth flow of the  program
try
{
// waiting for thread t2 to complete its execution
t2.join();
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
System.out.println("The state of thread t2 when it has completed it's execution - " + t2.getState());
}

}
```

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
```

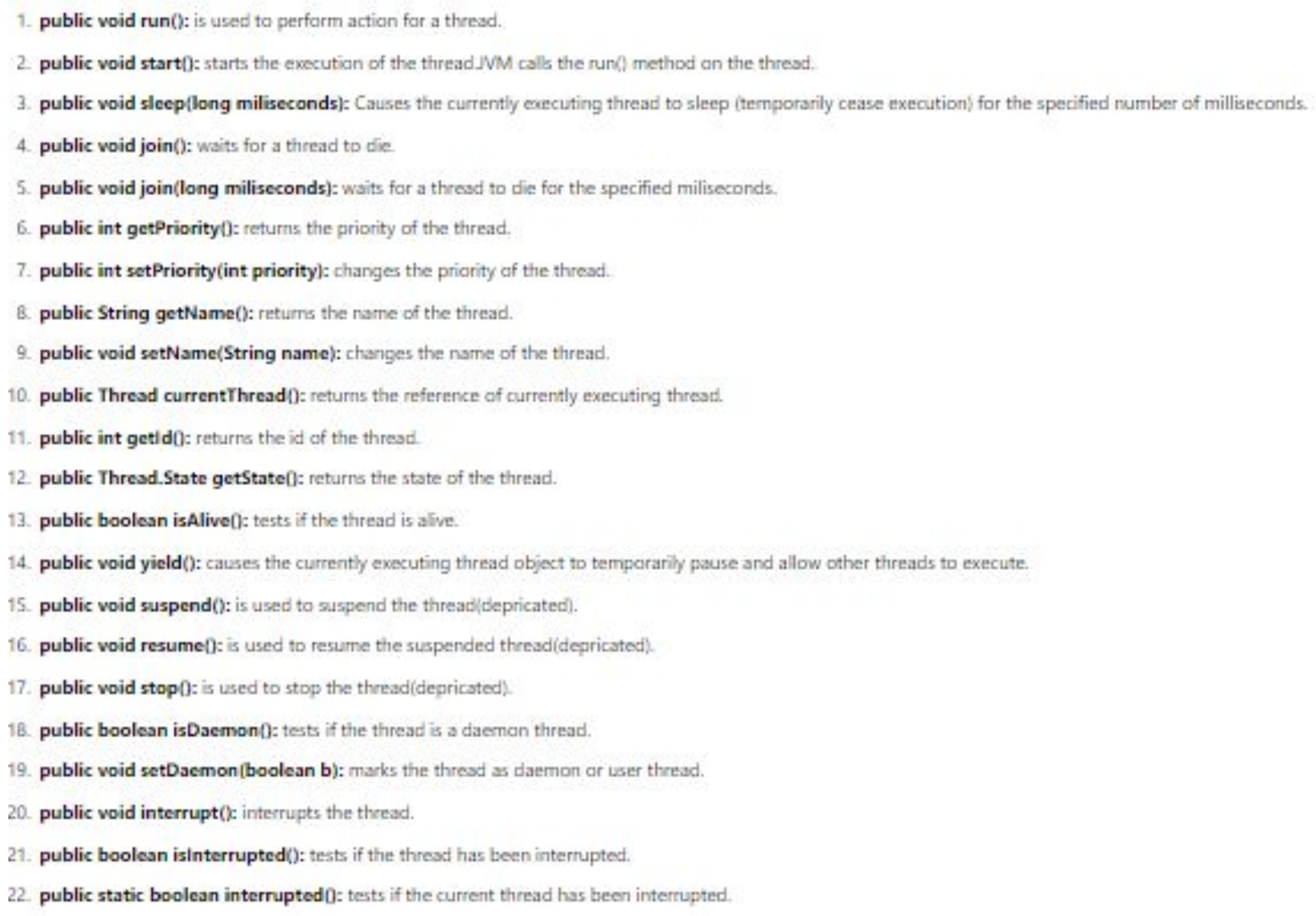
How to create a thread in Java

There are two ways to create a thread:
By extending Thread class
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.

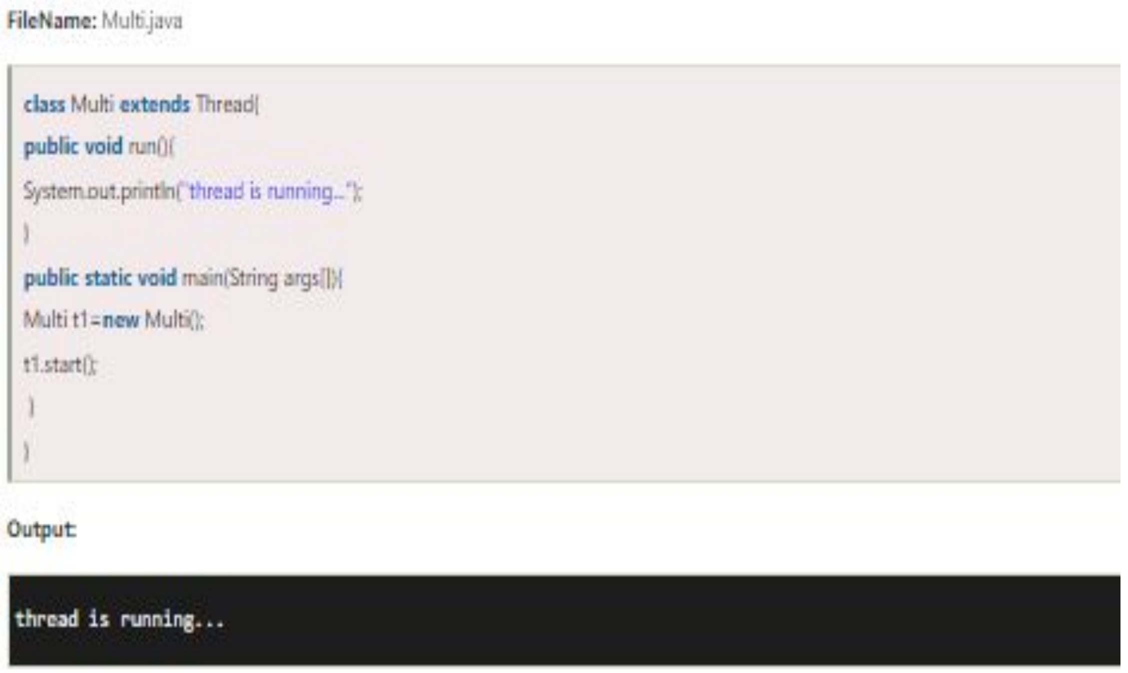
Commonly used Constructors of Thread class:
Thread()
Thread(String name)
Thread(Runnable r)
Thread(Runnable r,String name)

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.

Java Thread Example by extending Thread class



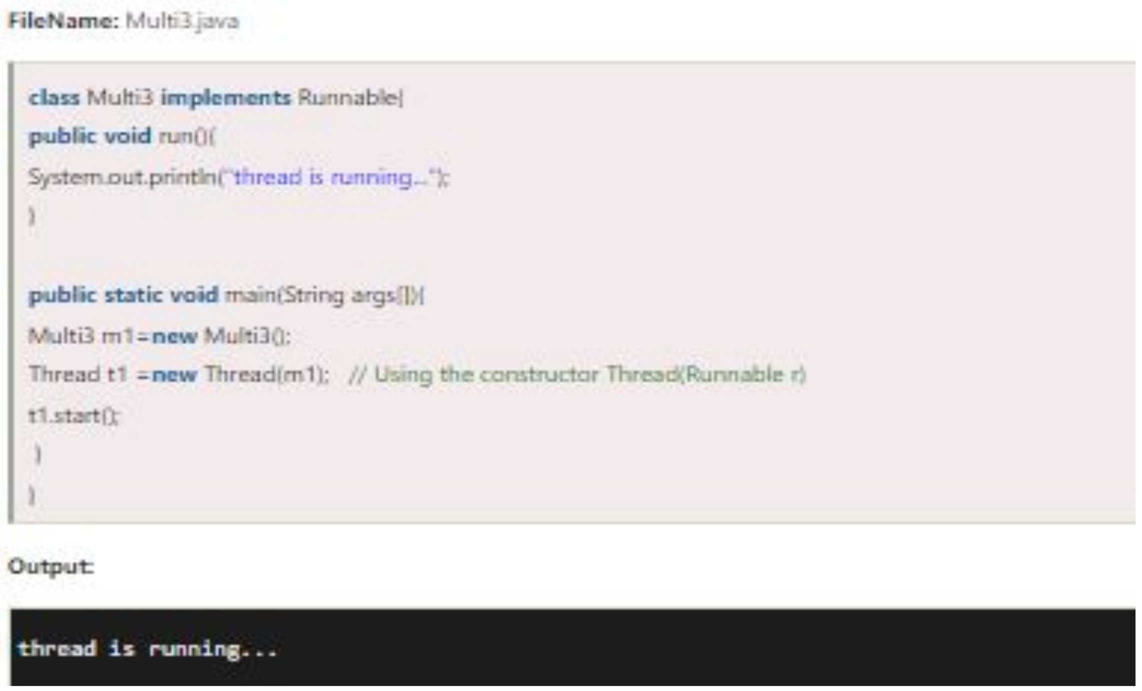
FileName: Multi.java

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output:

```
thread is running...
```

Java Thread Example by implementing Runnable interface



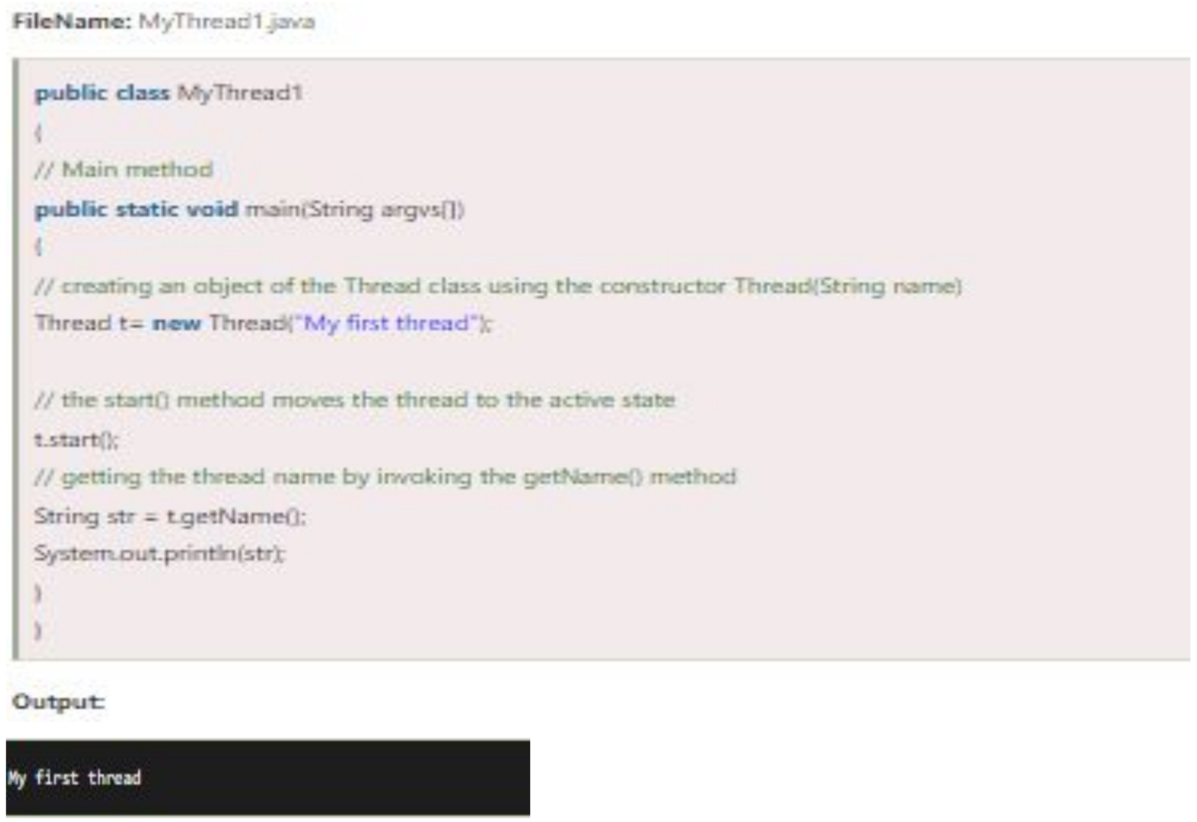
FileName: Multi3.java

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
t1.start();
 }
}
```

Output:

```
thread is running...
```

Using the Thread Class: Thread(String Name)



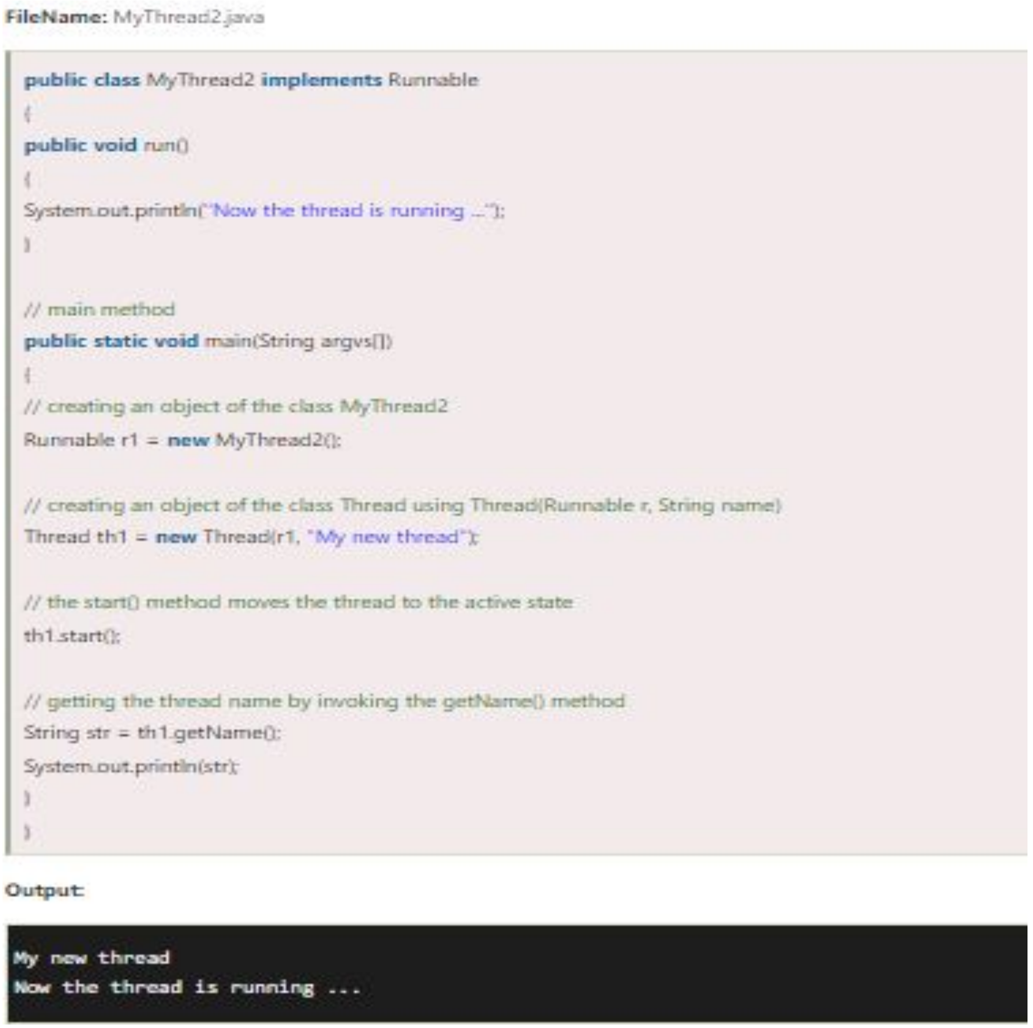
FileName: MyThread1.java

```
public class MyThread1
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
}
```

Output:

```
My first thread
```

Using the Thread Class: Thread(Runnable r, String name)



FileName: MyThread2.java

```java
public class MyThread2 implements Runnable
{
public void run()
{
System.out.println("Now the thread is running ...");
}

// main method
public static void main(String argvs[])
{
// creating an object of the class MyThread2
Runnable r1 = new MyThread2();

// creating an object of the class Thread using Thread(Runnable r, String name)
Thread th1 = new Thread(r1, "My new thread");

// the start() method moves the thread to the active state
th1.start();

// getting the thread name by invoking the getName() method
String str = th1.getName();
System.out.println(str);
}
}
```

Output:

```
My new thread
Now the thread is running ...
```

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

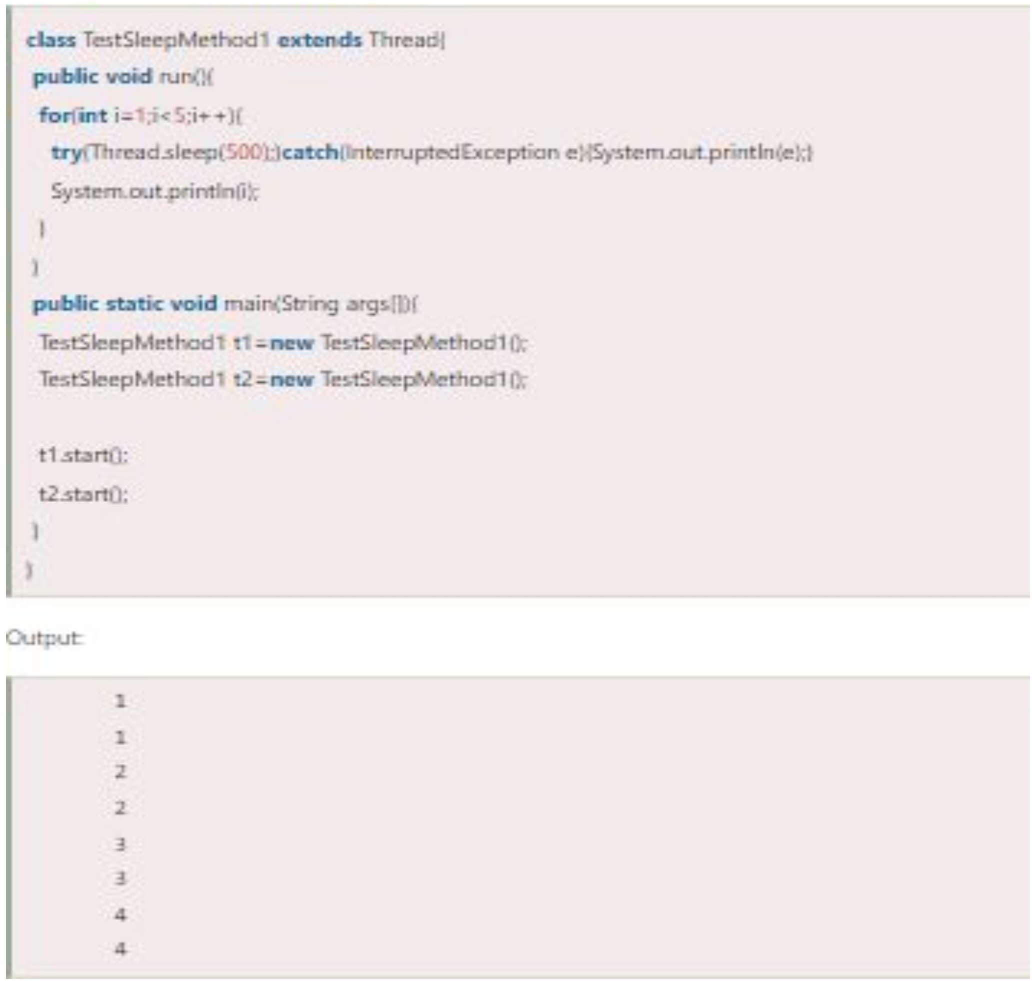
- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

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



```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

  t1.start();
  t2.start();
 }
}
```

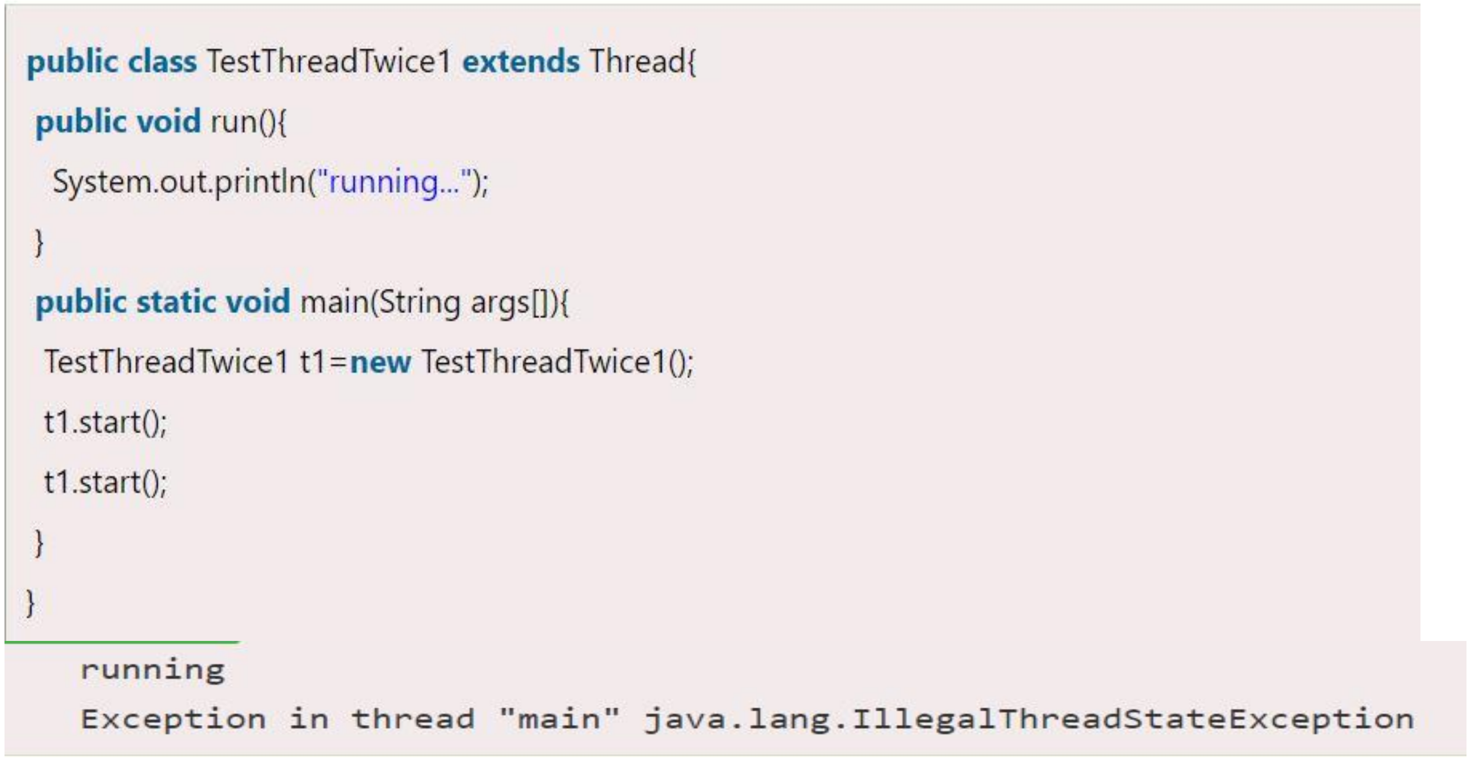
Output:

```
       1
       1
       2
       2
       3
       3
       4
       4
```

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.

Example of start a thread twice in java

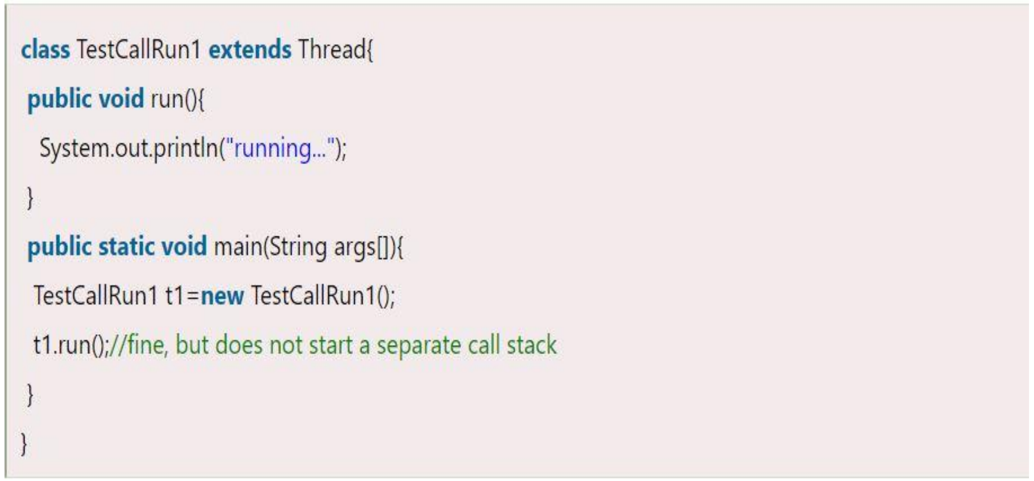


```java
public class TestThreadTwice1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

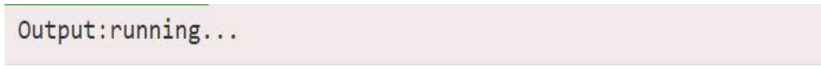
```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

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

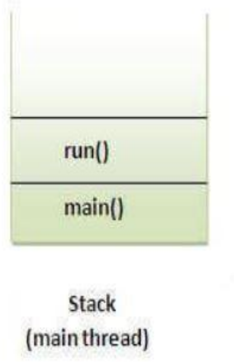
- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.



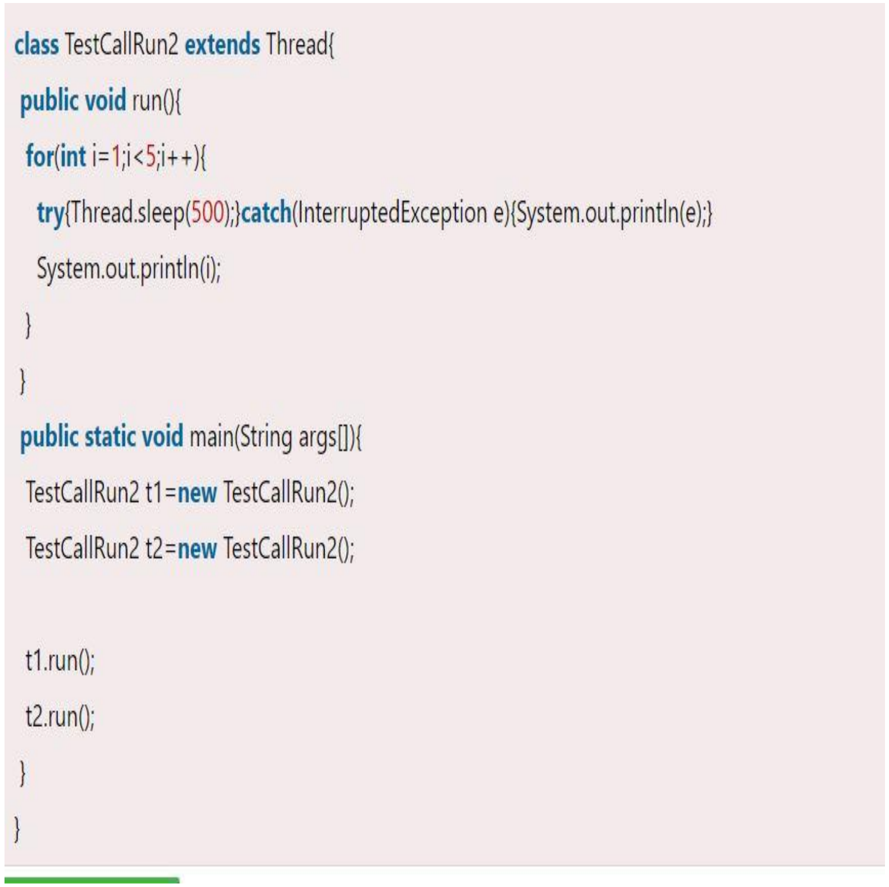
```
class TestCallRun1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestCallRun1 t1=new TestCallRun1();
  t1.run();//fine, but does not start a separate call stack
 }
}
```



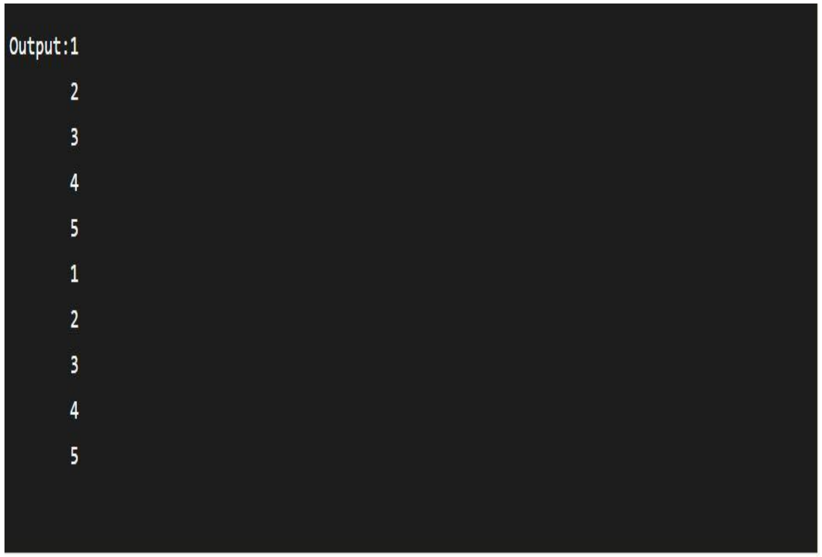
```
Output:running...
```



Problem if you direct call run() method



```
class TestCallRun2 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestCallRun2 t1=new TestCallRun2();
  TestCallRun2 t2=new TestCallRun2();

  t1.run();
  t2.run();
 }
}
```



```
Output:1
       2
       3
       4
       5
       1
       2
       3
       4
       5
```

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

The join() method

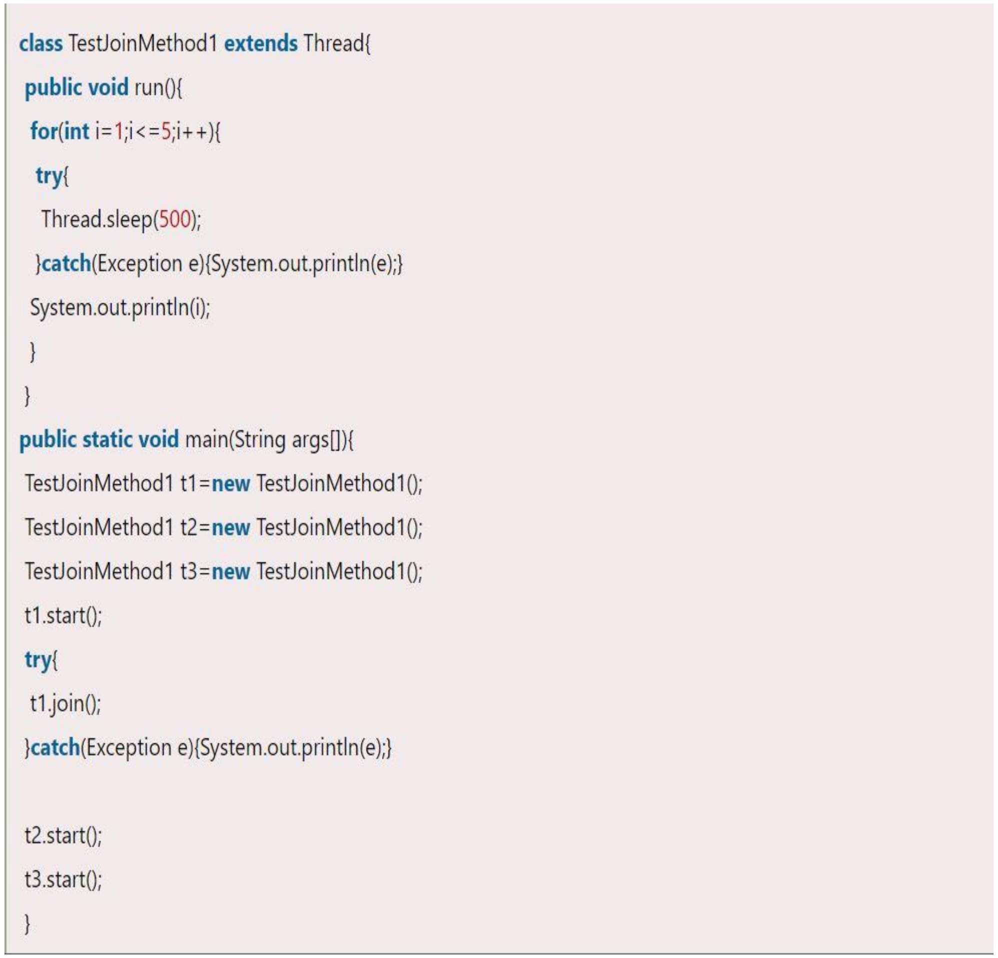
- The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

- **Syntax**

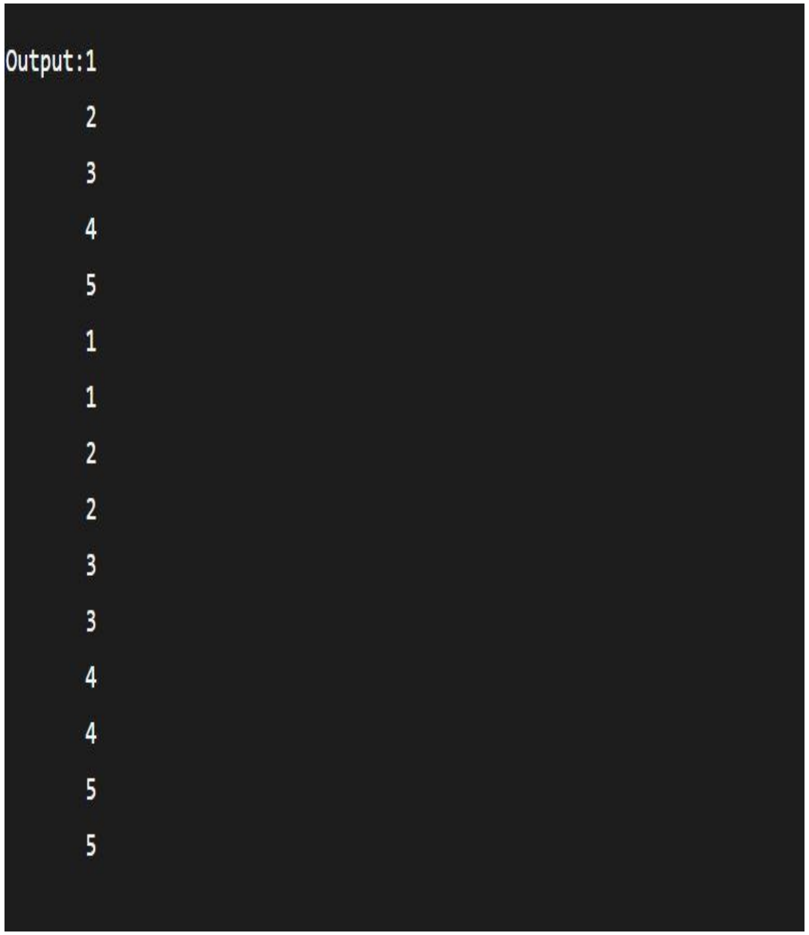
public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException

Example of join () method in java

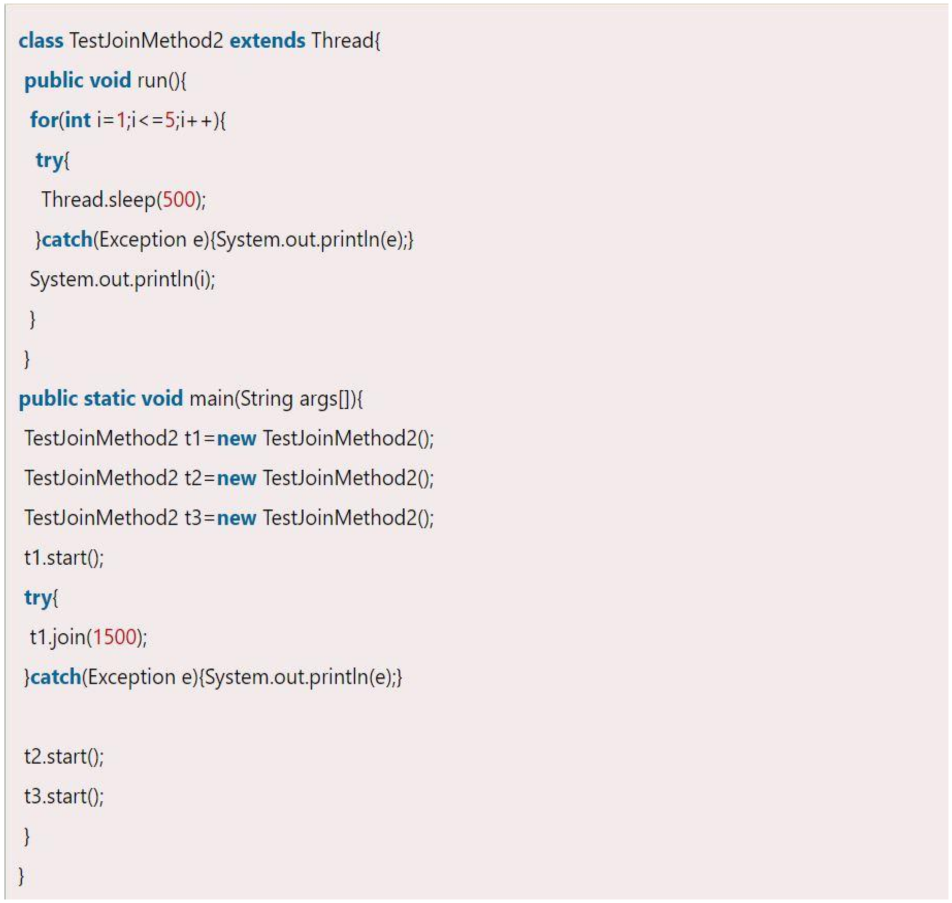


```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
```

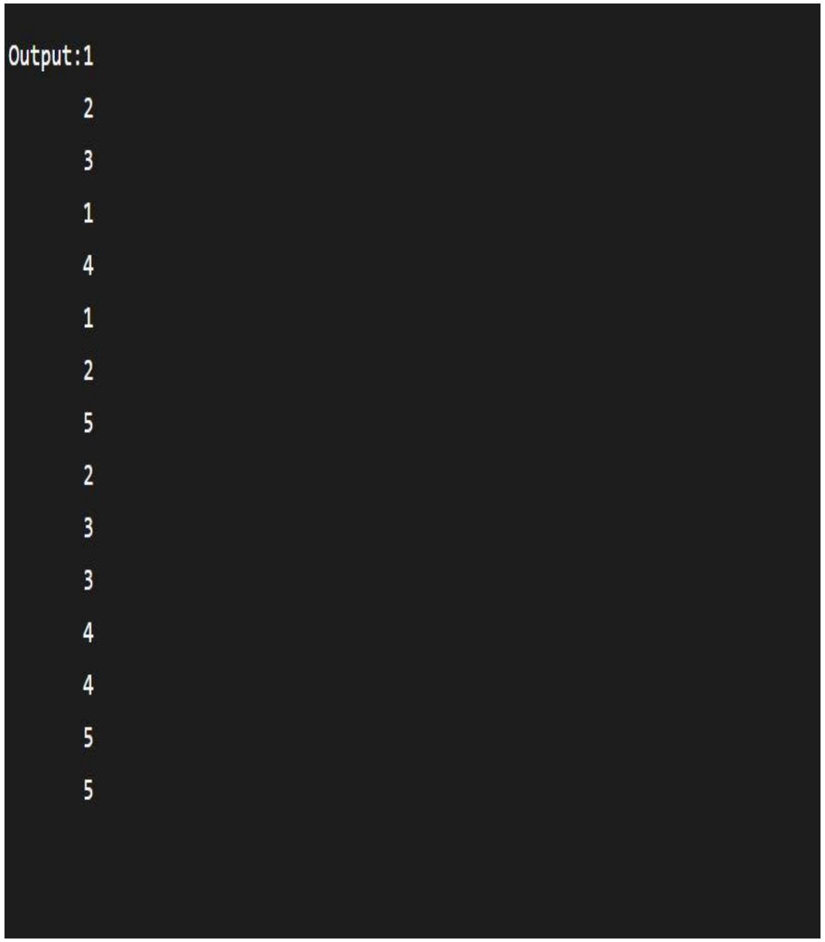


```
Output:1
       2
       3
       4
       5
       1
       1
       2
       2
       3
       3
       4
       4
       5
       5
```

Example of join(long miliseconds) method



```java
class TestJoinMethod2 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod2 t1=new TestJoinMethod2();
 TestJoinMethod2 t2=new TestJoinMethod2();
 TestJoinMethod2 t3=new TestJoinMethod2();
 t1.start();
 try{
  t1.join(1500);
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```



```
Output:1
       2
       3
       1
       4
       1
       2
       5
       2
       3
       3
       4
       4
       5
       5
```

Naming Thread and Current 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.

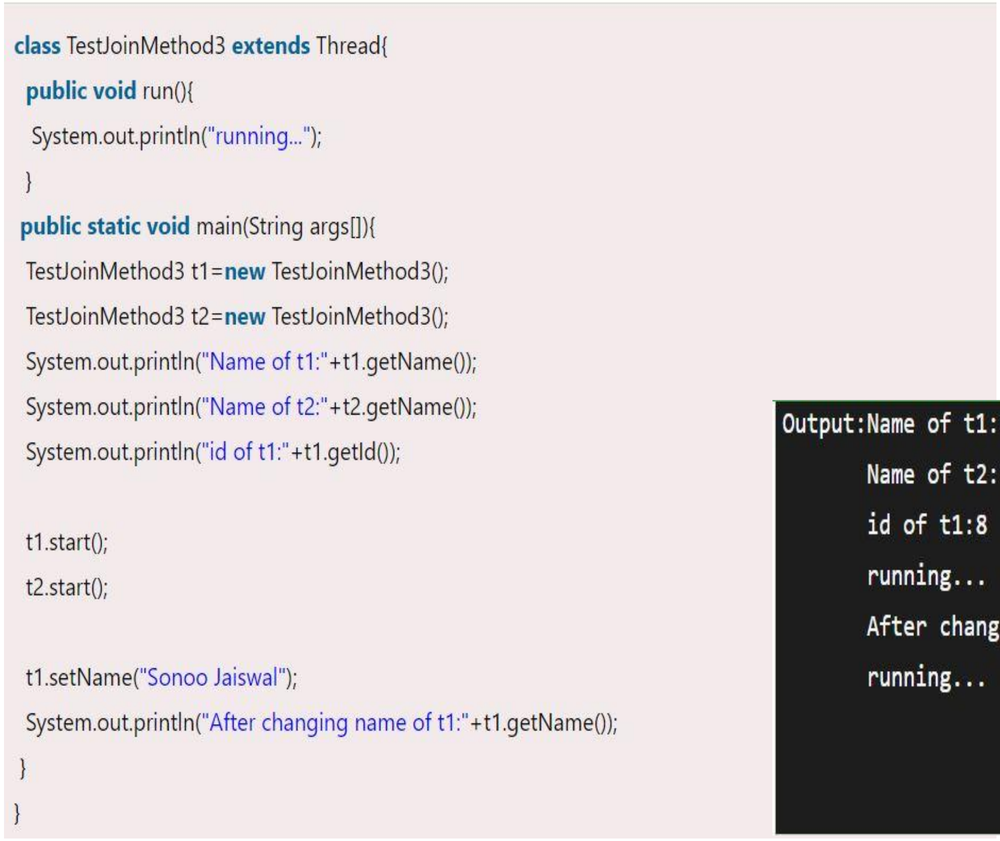
- The syntax of setName() and getName() methods are given below:

  public String getName(): is used to return the name of a thread.

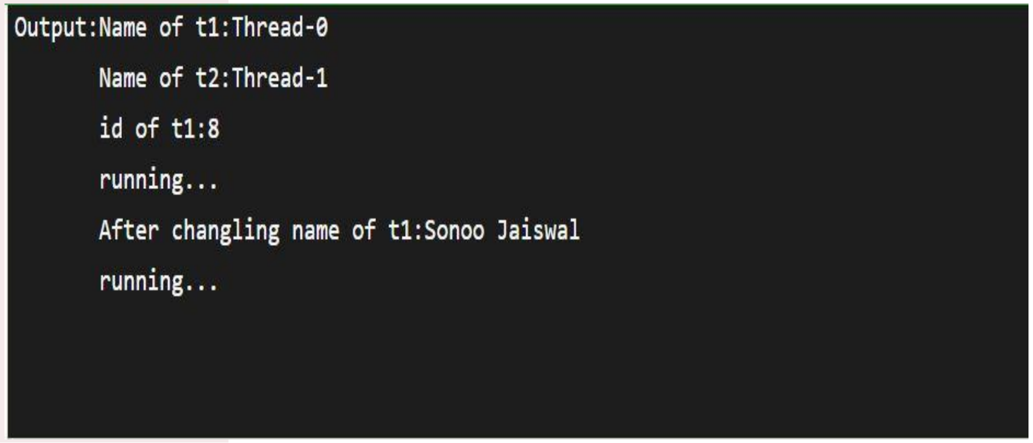
  public void setName(String name): is used to change the name of a thread.

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

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



```
class TestJoinMethod3 extends Thread{
 public void run(){
  System.out.println("running...");
 }
public static void main(String args[]){
 TestJoinMethod3 t1=new TestJoinMethod3();
 TestJoinMethod3 t2=new TestJoinMethod3();
 System.out.println("Name of t1:"+t1.getName());
 System.out.println("Name of t2:"+t2.getName());
 System.out.println("id of t1:"+t1.getId());

 t1.start();
 t2.start();

 t1.setName("Sonoo Jaiswal");
 System.out.println("After changing name of t1:"+t1.getName());
 }
}
```



```
Output:Name of t1:Thread-0
       Name of t2:Thread-1
       id of t1:8
       running...
       After changling name of t1:Sonoo Jaiswal
       running...
```

The currentThread() method

public static Thread currentThread()

```
class TestJoinMethod4 extends Thread{
 public void run(){
  System.out.println(Thread.currentThread().getName());
 }
 }
 public static void main(String args[]){
  TestJoinMethod4 t1=new TestJoinMethod4();
  TestJoinMethod4 t2=new TestJoinMethod4();

  t1.start();
  t2.start();
 }
}
```

```
Output:Thread-0
       Thread-1
```

Priority of a Thread (Thread Priority):

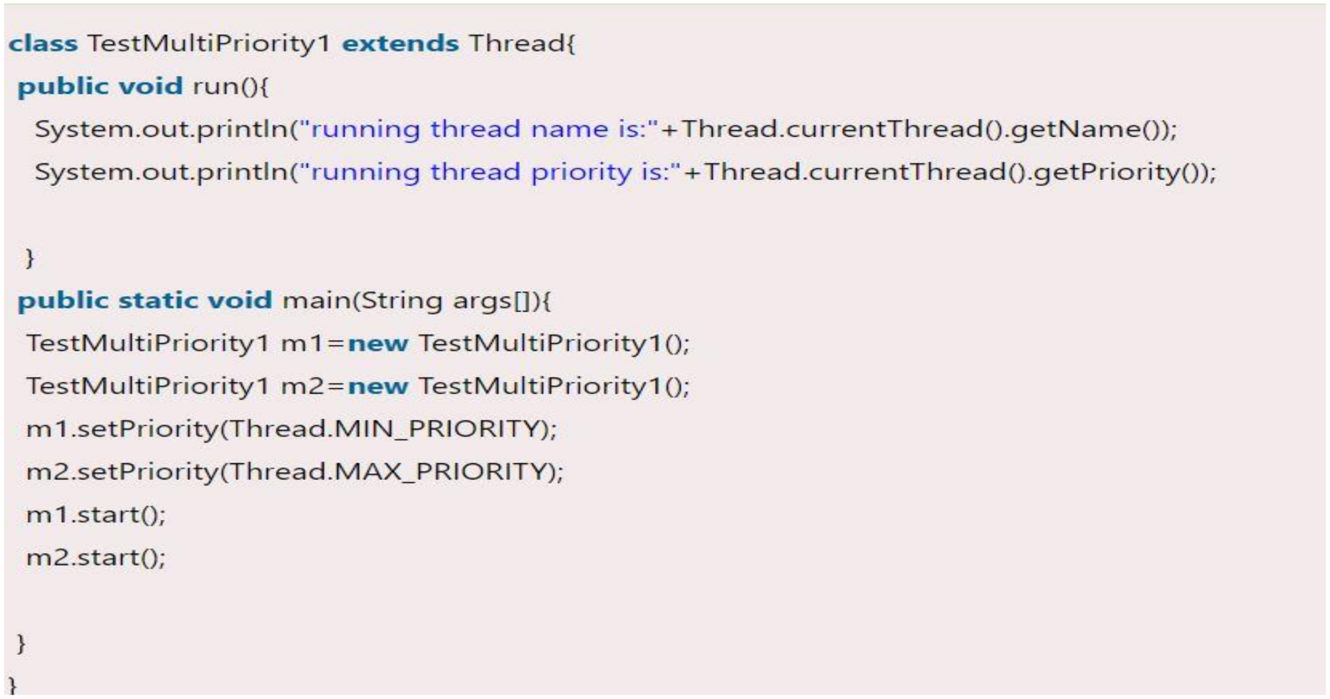
- Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

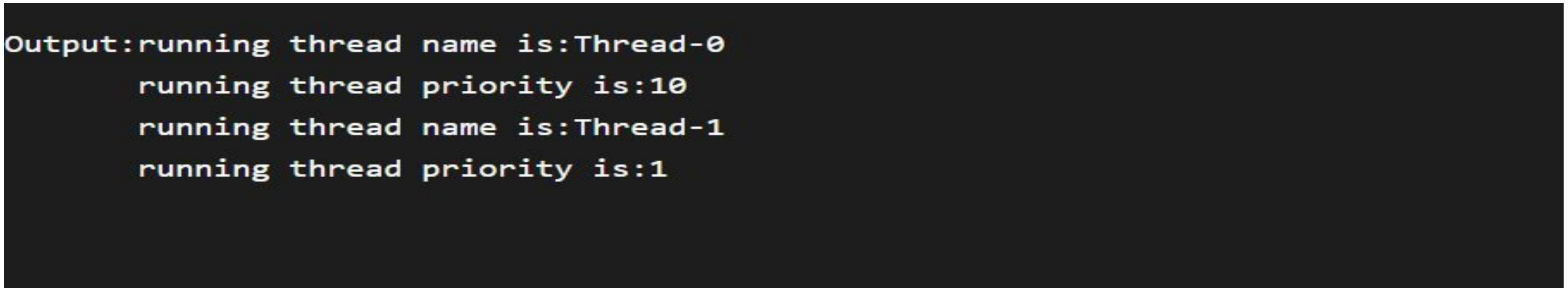
- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- public static int MAX_PRIORITY

- 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:



```java
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```



```
Output:running thread name is:Thread-0
       running thread priority is:10
       running thread name is:Thread-1
       running thread priority is:1
```

Daemon Thread in Java

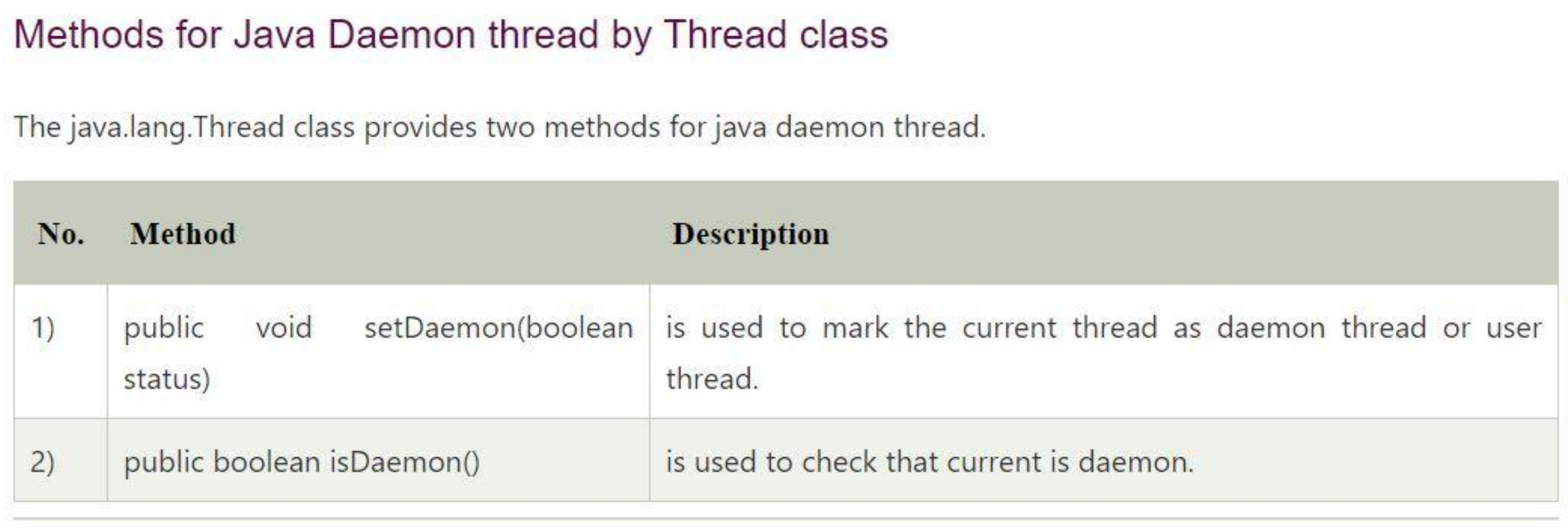
- **Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- There are many java daemon threads running automatically e.g. gc, finalizer, 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.

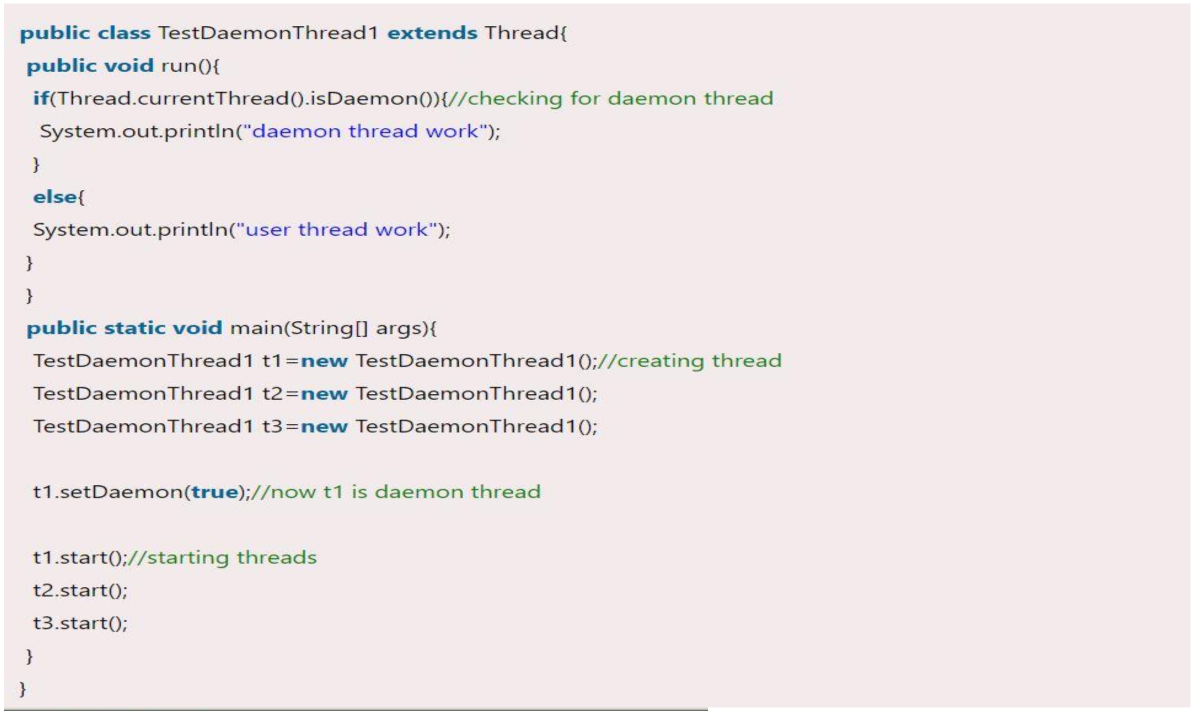


Methods for Java Daemon thread by Thread class

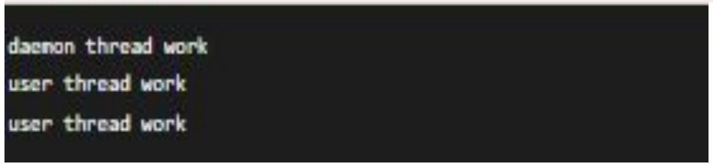
The java.lang.Thread class provides two methods for java daemon thread.

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

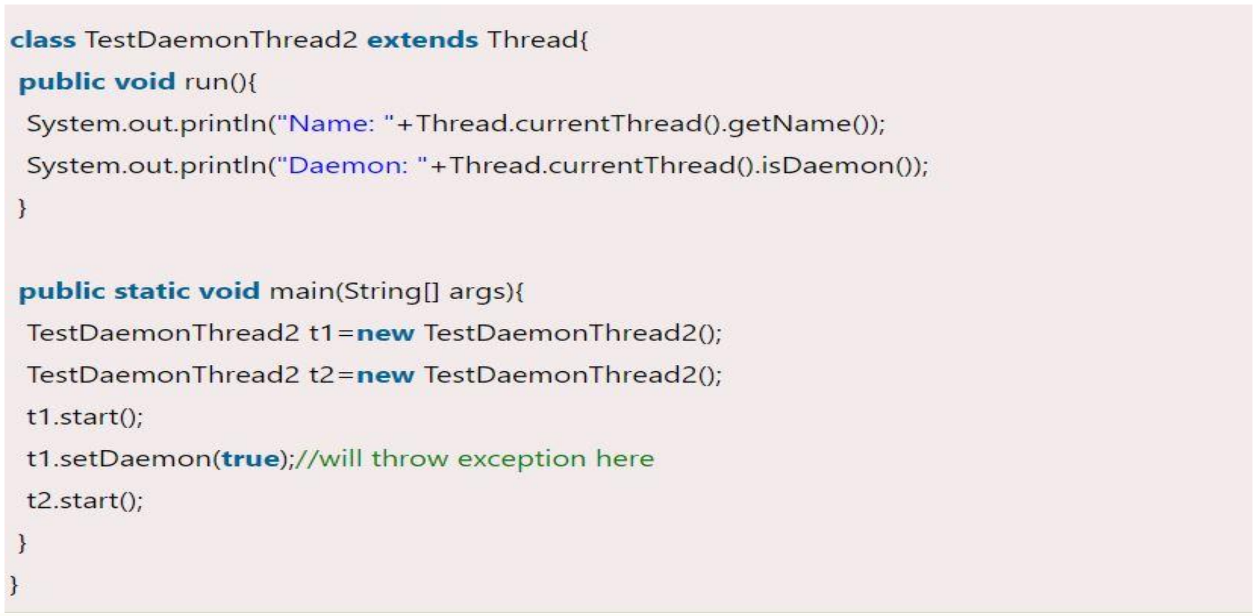


```
public class TestDaemonThread1 extends Thread{
 public void run(){
  if(Thread.currentThread().isDaemon()){//checking for daemon thread
   System.out.println("daemon thread work");
  }
  else{
  System.out.println("user thread work");
 }
 }
 public static void main(String[] args){
  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
  TestDaemonThread1 t2=new TestDaemonThread1();
  TestDaemonThread1 t3=new TestDaemonThread1();

  t1.setDaemon(true);//now t1 is daemon thread

  t1.start();//starting threads
  t2.start();
  t3.start();
 }
}
```

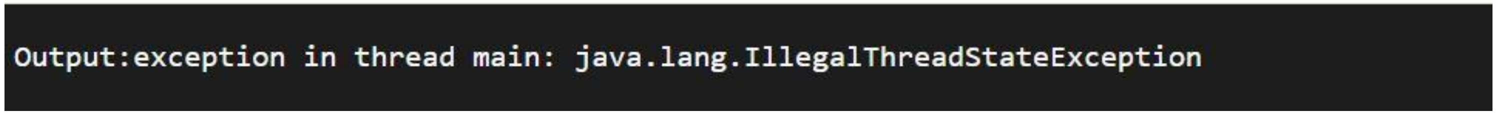


```
daemon thread work
user thread work
user thread work
```

Simple example of Daemon thread in java



```
class TestDaemonThread2 extends Thread{
 public void run(){
  System.out.println("Name: "+Thread.currentThread().getName());
  System.out.println("Daemon: "+Thread.currentThread().isDaemon());
 }

 public static void main(String[] args){
  TestDaemonThread2 t1=new TestDaemonThread2();
  TestDaemonThread2 t2=new TestDaemonThread2();
  t1.start();
  t1.setDaemon(true);//will throw exception here
  t2.start();
 }
}
```



```
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**

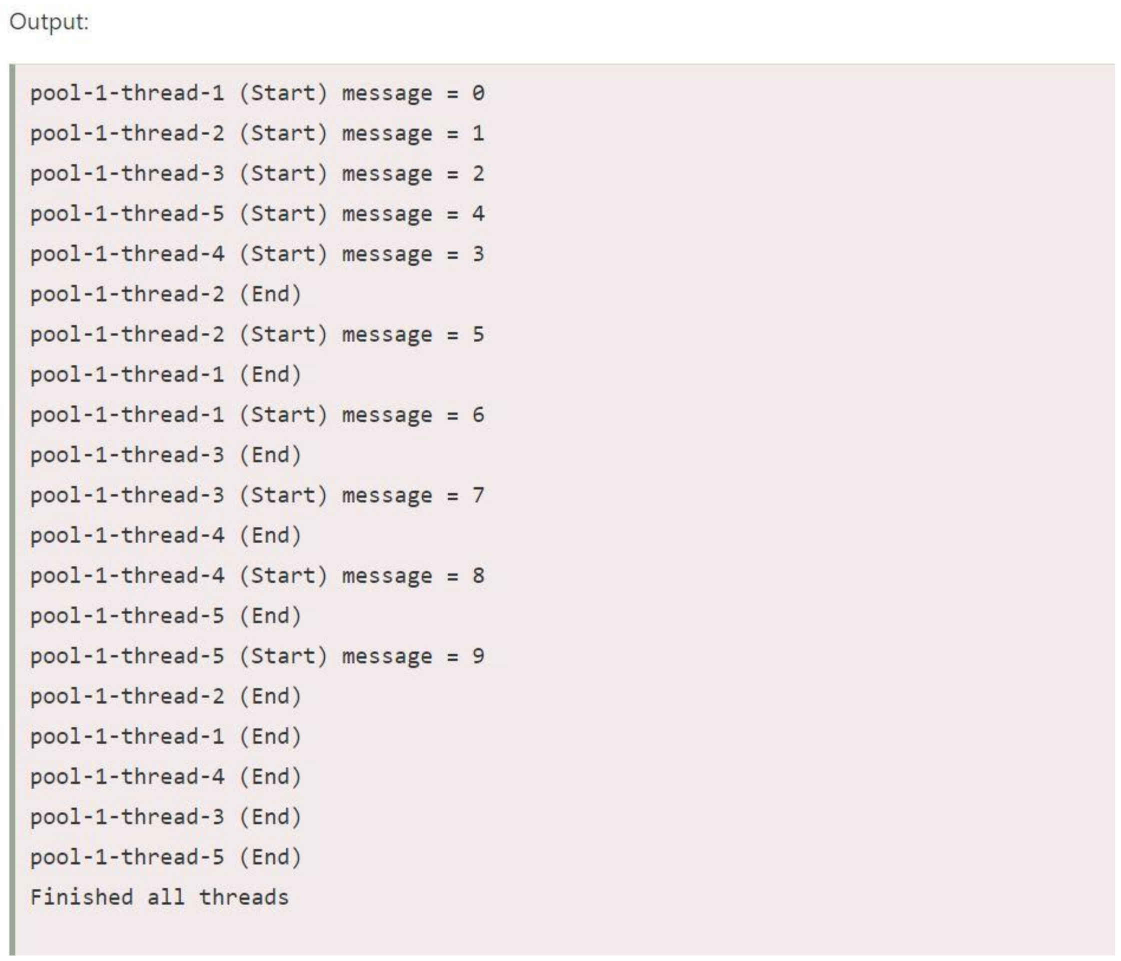
It is used in Servlet and JSP where container creates a thread pool to process the request.

Example of Java Thread Pool

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class WorkerThread implements Runnable {
   private String message;
   public WorkerThread(String s){
      this.message=s;
   }
    public void run() {
      System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);
      processmessage();//call processmessage method that sleeps the thread for 2 seconds
      System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name
   }
   private void processmessage() {
      try {  Thread.sleep(2000);  } catch (InterruptedException e) { e.printStackTrace(); }
   }
}
```

```java
public class TestThreadPool {
    public static void main(String[] args) {
       ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads
       for (int i = 0; i < 10; i++) {
          Runnable worker = new WorkerThread("" + i);
          executor.execute(worker);//calling execute method of ExecutorService
        }
       executor.shutdown();
       while (!executor.isTerminated()) {   }

       System.out.println("Finished all threads");
   }
}
```

Output of Java Thread Pool

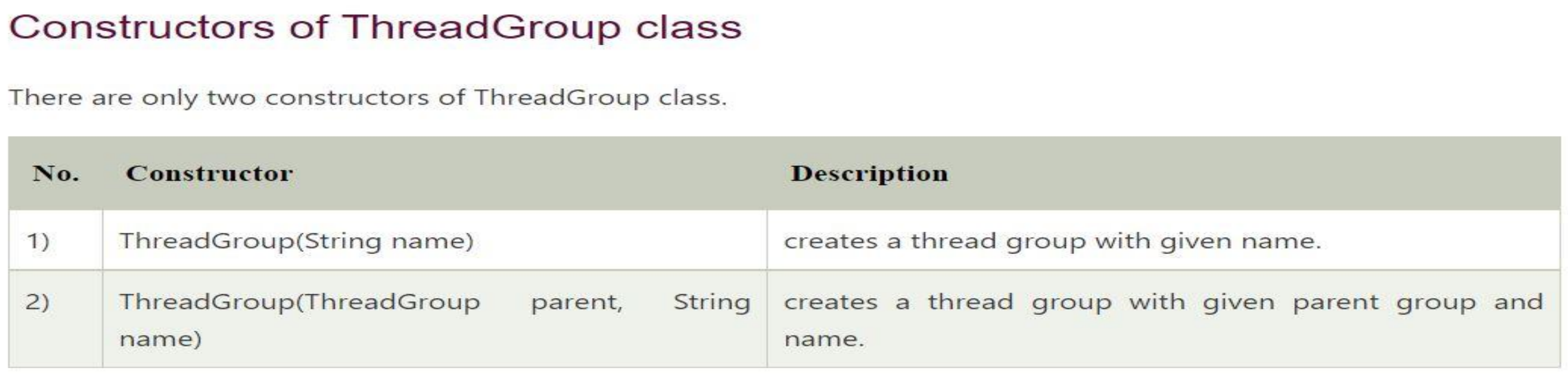


Output:

```
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
```

Thread Group 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.
- 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.

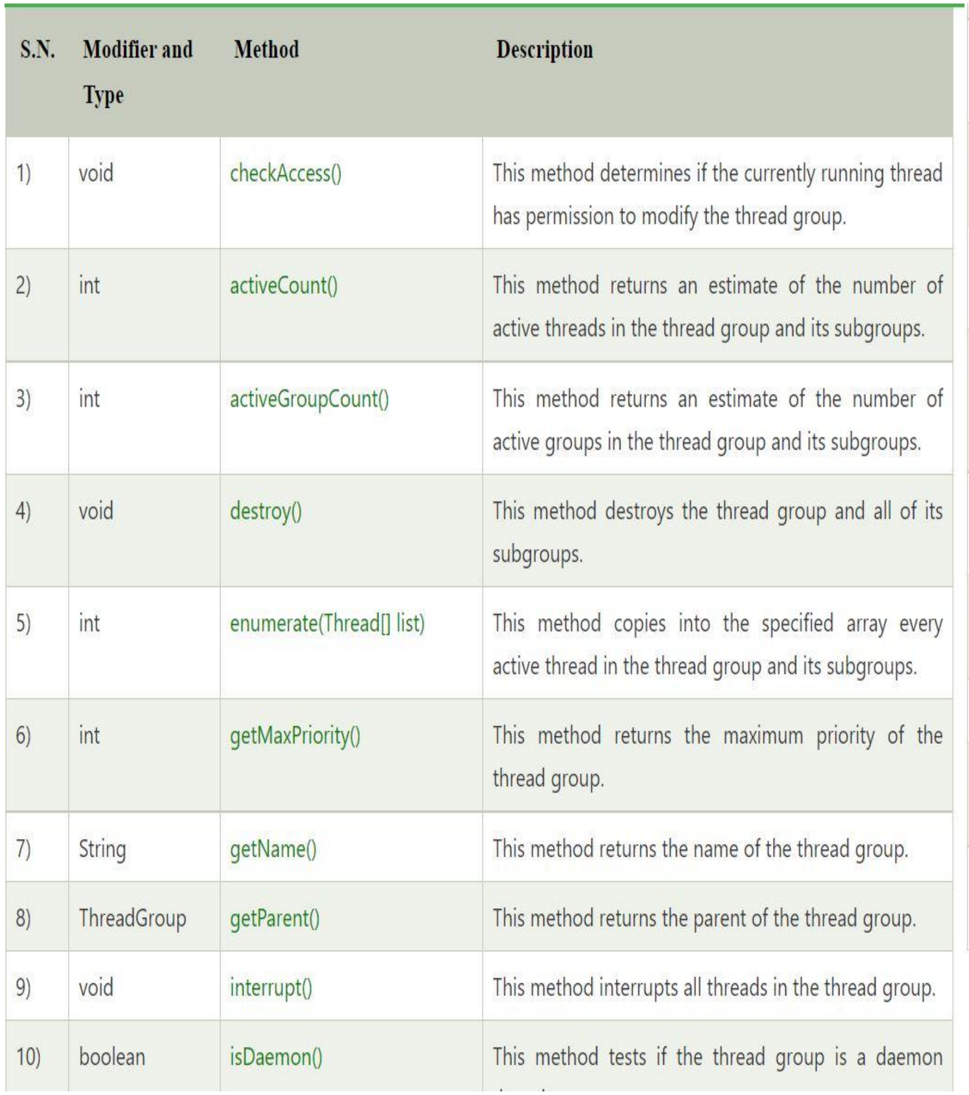
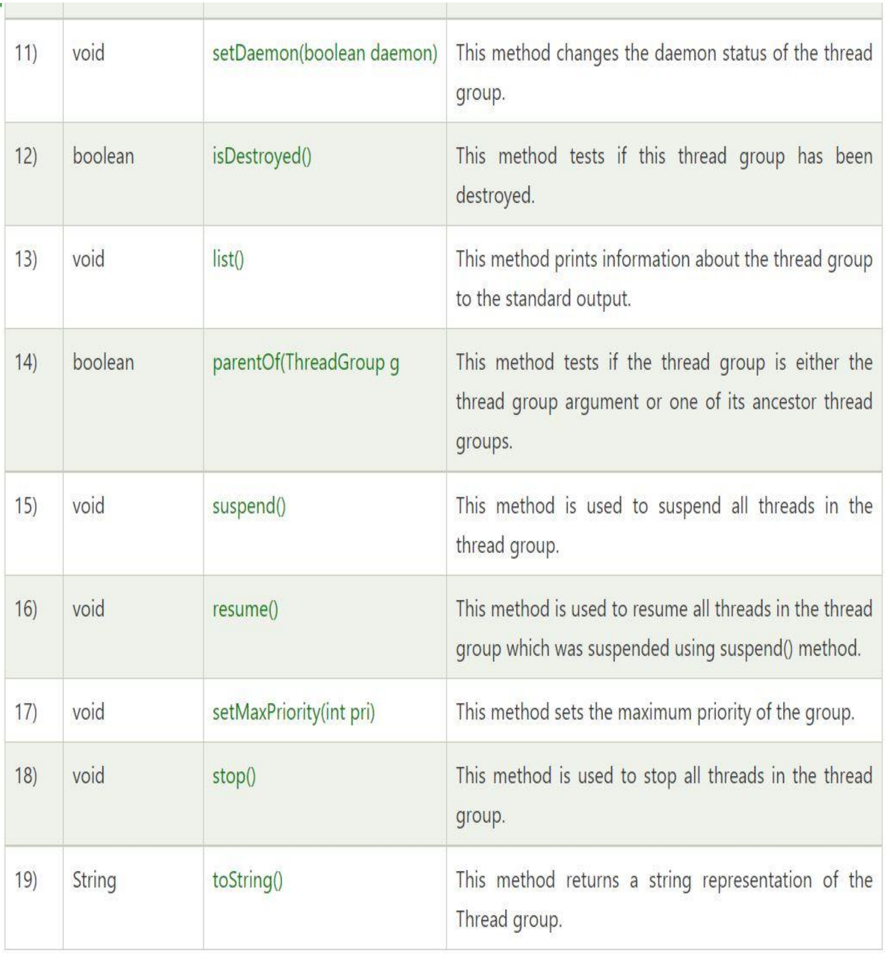


Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

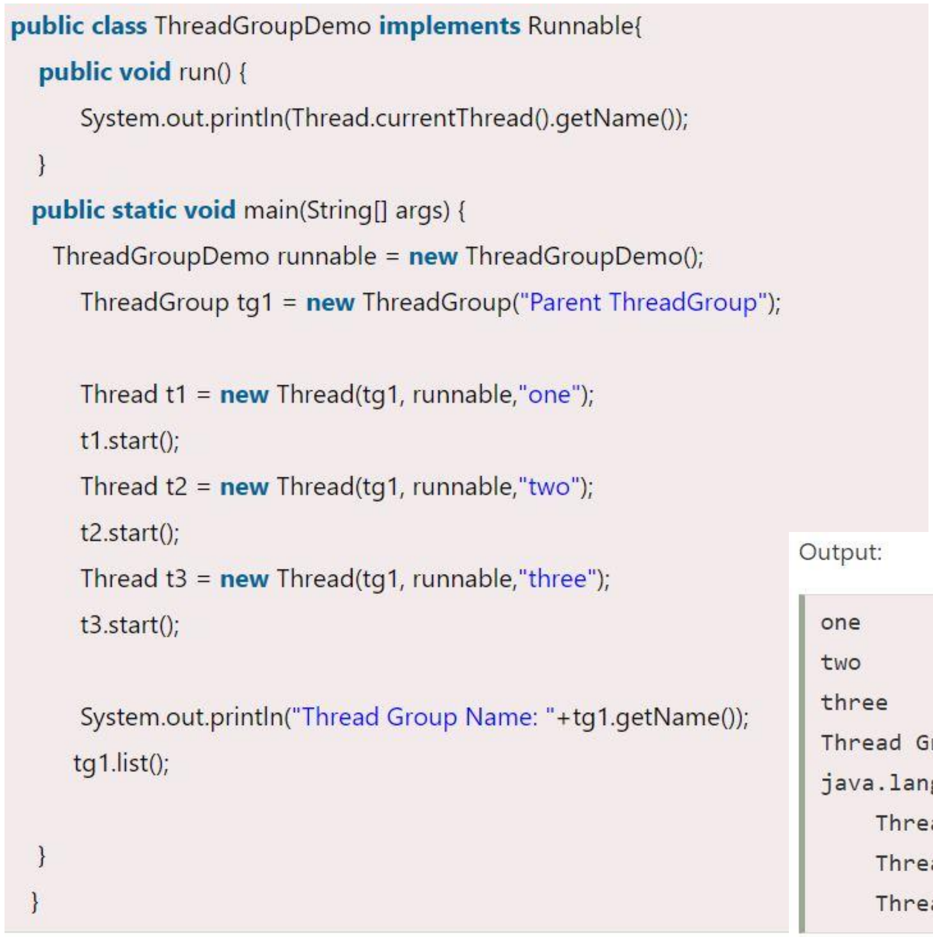
No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name.

Methods of Thread Group class

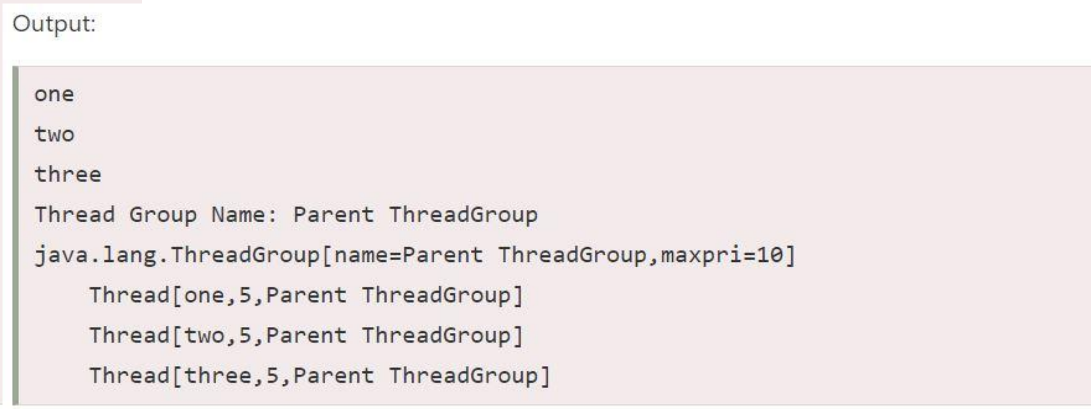



S.N.	Modifier and Type	Method	Description
1)	void	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.
2)	int	activeCount()	This method returns an estimate of the number of active threads in the thread group and its subgroups.
3)	int	activeGroupCount()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

Thread Group Example



```
public class ThreadGroupDemo implements Runnable{
   public void run() {
        System.out.println(Thread.currentThread().getName());
   }
  public static void main(String[] args) {
    ThreadGroupDemo runnable = new ThreadGroupDemo();
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

        Thread t1 = new Thread(tg1, runnable,"one");
        t1.start();
        Thread t2 = new Thread(tg1, runnable,"two");
        t2.start();
        Thread t3 = new Thread(tg1, runnable,"three");
        t3.start();

        System.out.println("Thread Group Name: "+tg1.getName());
       tg1.list();

   }
  }
```



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**

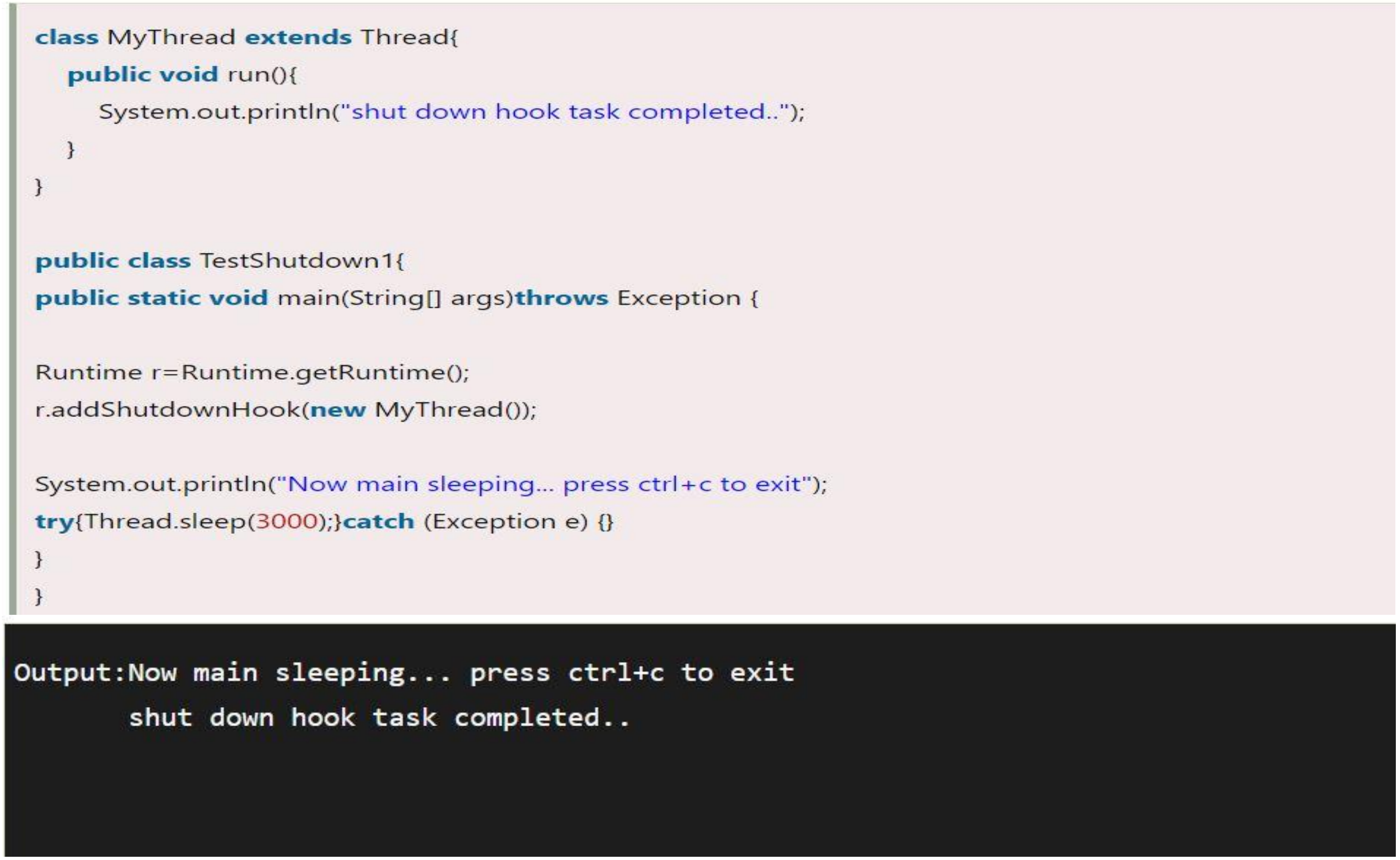
public void addShutdownHook(Thread hook){}

- 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



```
class MyThread extends Thread{
   public void run(){
      System.out.println("shut down hook task completed..");
   }
}

public class TestShutdown1{
public static void main(String[] args)throws Exception {

Runtime r=Runtime.getRuntime();
r.addShutdownHook(new MyThread());

System.out.println("Now main sleeping... press ctrl+c to exit");
try{Thread.sleep(3000);}catch (Exception e) {}
}
}
```

```
Output:Now main sleeping... press ctrl+c to exit
       shut down hook task completed..
```

Same example of Shutdown Hook by anonymous class

```
public class TestShutdown2{
public static void main(String[] args)throws Exception {

Runtime r=Runtime.getRuntime();

r.addShutdownHook(new Thread(){
public void run(){
    System.out.println("shut down hook task completed..");
    }
}
);

System.out.println("Now main sleeping... press ctrl+c to exit");
try{Thread.sleep(3000);}catch (Exception e) {}
}
}
```

```
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.

```java
class TestMultitasking1 extends Thread{
 public void run(){
   System.out.println("task one");
 }
 public static void main(String args[]){
  TestMultitasking1 t1=new TestMultitasking1();
  TestMultitasking1 t2=new TestMultitasking1();
  TestMultitasking1 t3=new TestMultitasking1();

  t1.start();
  t2.start();
  t3.start();
 }
}
```

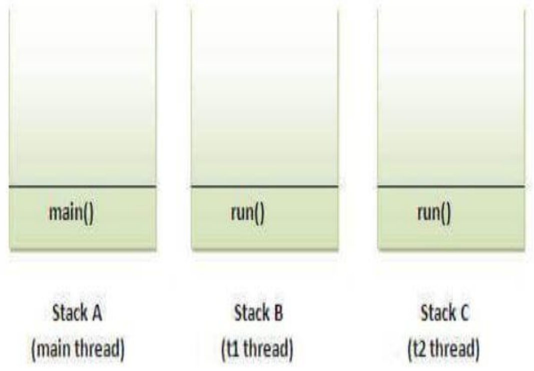
```
Output:task one
       task one
       task one
```

Program of performing single task by multiple threads

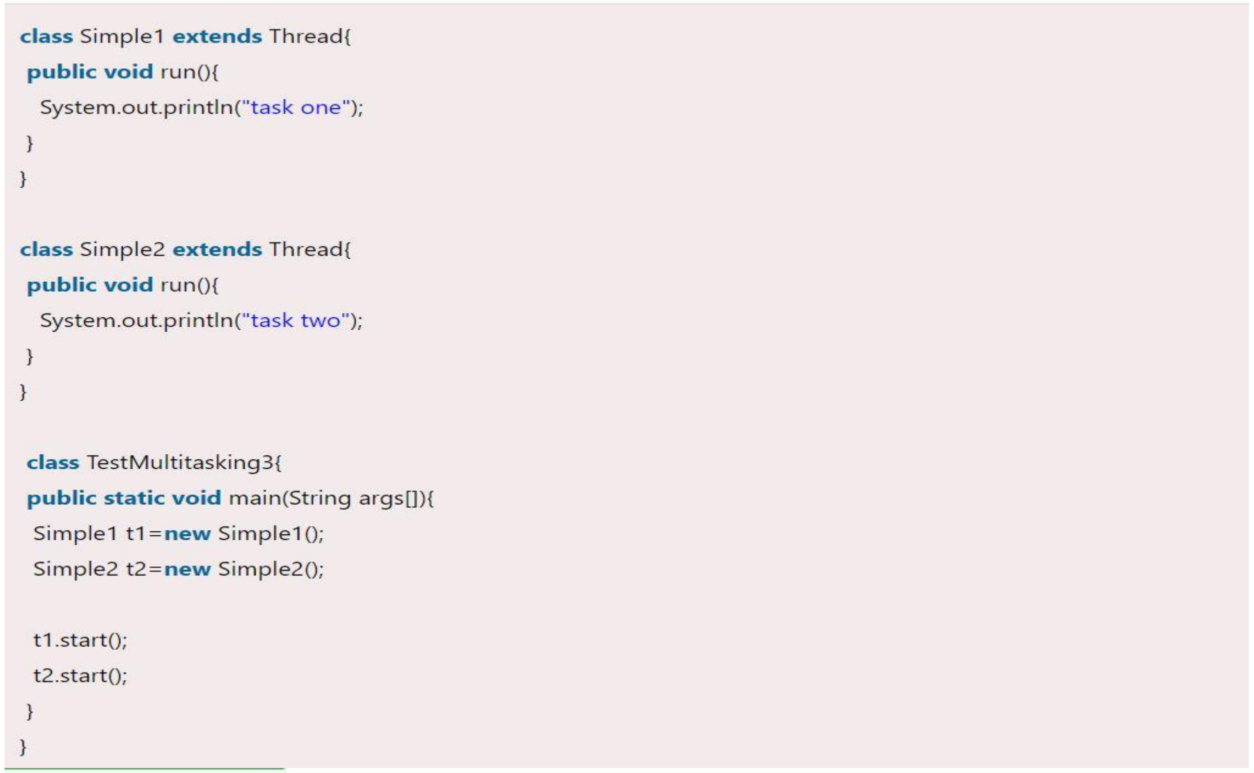
```
class TestMultitasking2 implements Runnable{
public void run(){
System.out.println("task one");
}

public static void main(String args[]){
Thread t1 =new Thread(new TestMultitasking2());//passing annonymous object of TestMultitasking2 class
Thread t2 =new Thread(new TestMultitasking2());

t1.start();
t2.start();

 }
}
```

```
Output:task one
       task one
```

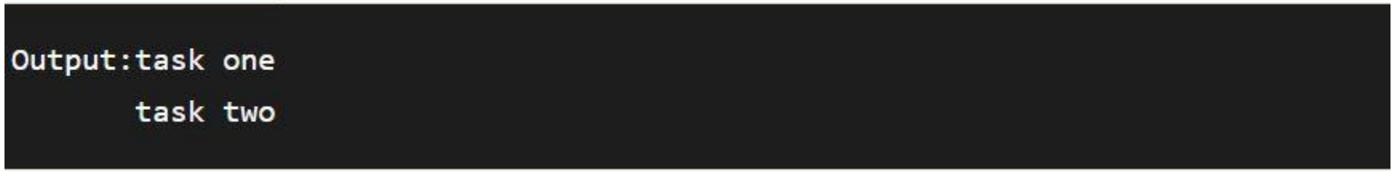
Note: Each thread run in a separate callstack.



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

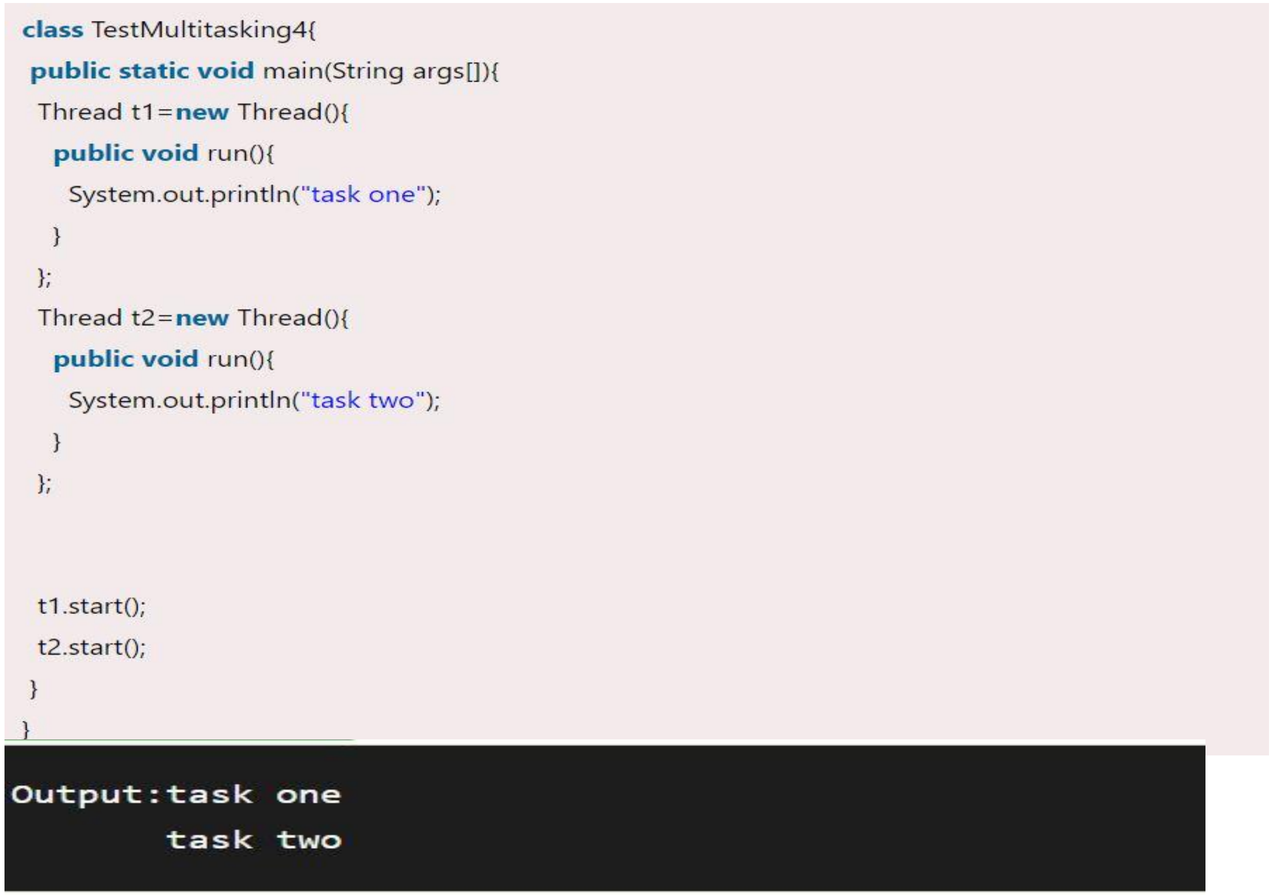


```
class Simple1 extends Thread{
 public void run(){
   System.out.println("task one");
 }
}

class Simple2 extends Thread{
 public void run(){
   System.out.println("task two");
 }
}

 class TestMultitasking3{
 public static void main(String args[]){
  Simple1 t1=new Simple1();
  Simple2 t2=new Simple2();

  t1.start();
  t2.start();
 }
}
```



```
Output:task one
       task two
```

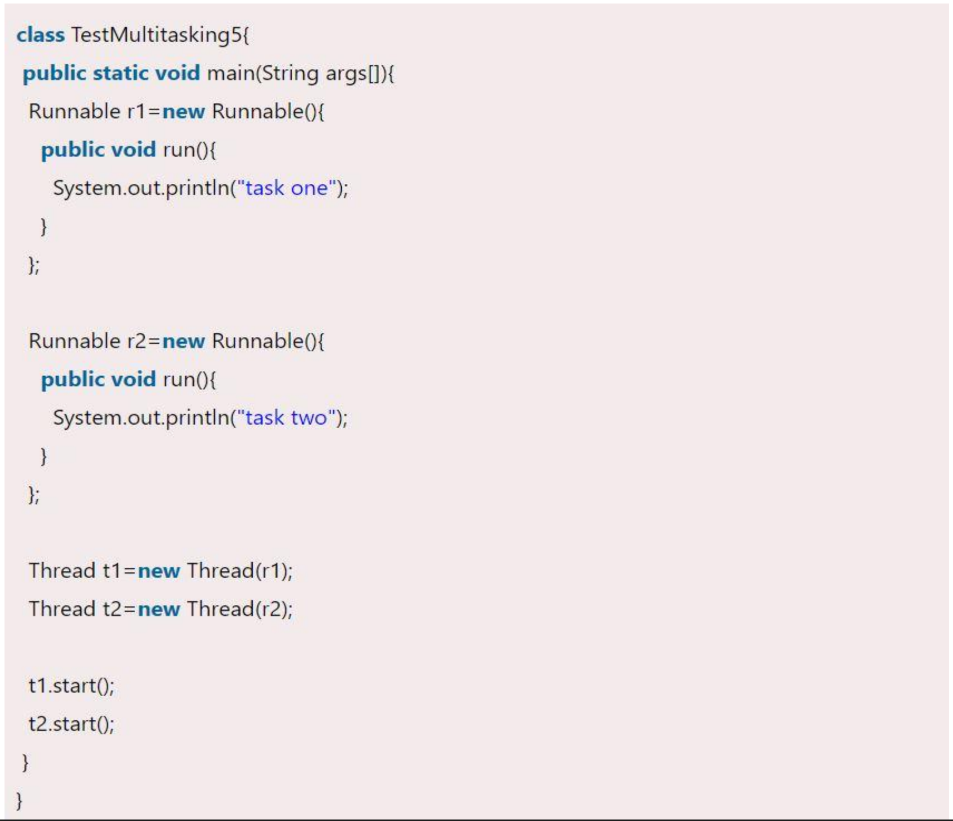
Same example as previous slide by anonymous class that extends Thread class



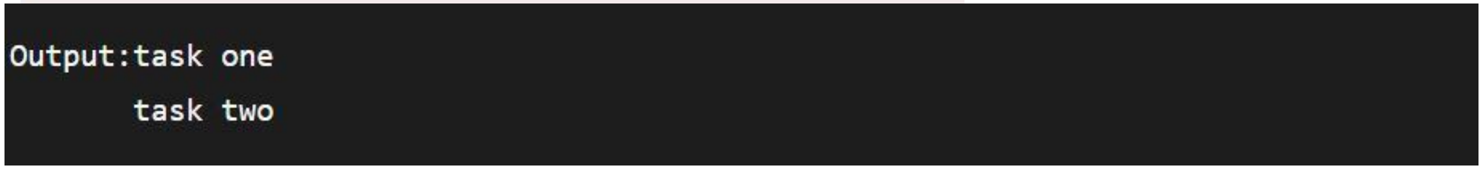
```java
class TestMultitasking4{
 public static void main(String args[]){
  Thread t1=new Thread(){
    public void run(){
      System.out.println("task one");
    }
  };
  Thread t2=new Thread(){
    public void run(){
      System.out.println("task two");
    }
  };


  t1.start();
  t2.start();
 }
}
```

```
Output:task one
       task two
```

Same example by anonymous class that implements Runnable interface



```java
class TestMultitasking5{
 public static void main(String args[]){
  Runnable r1=new Runnable(){
    public void run(){
      System.out.println("task one");
    }
  };

  Runnable r2=new Runnable(){
    public void run(){
      System.out.println("task two");
    }
  };

  Thread t1=new Thread(r1);
  Thread t2=new Thread(r2);

  t1.start();
  t2.start();
 }
}
```



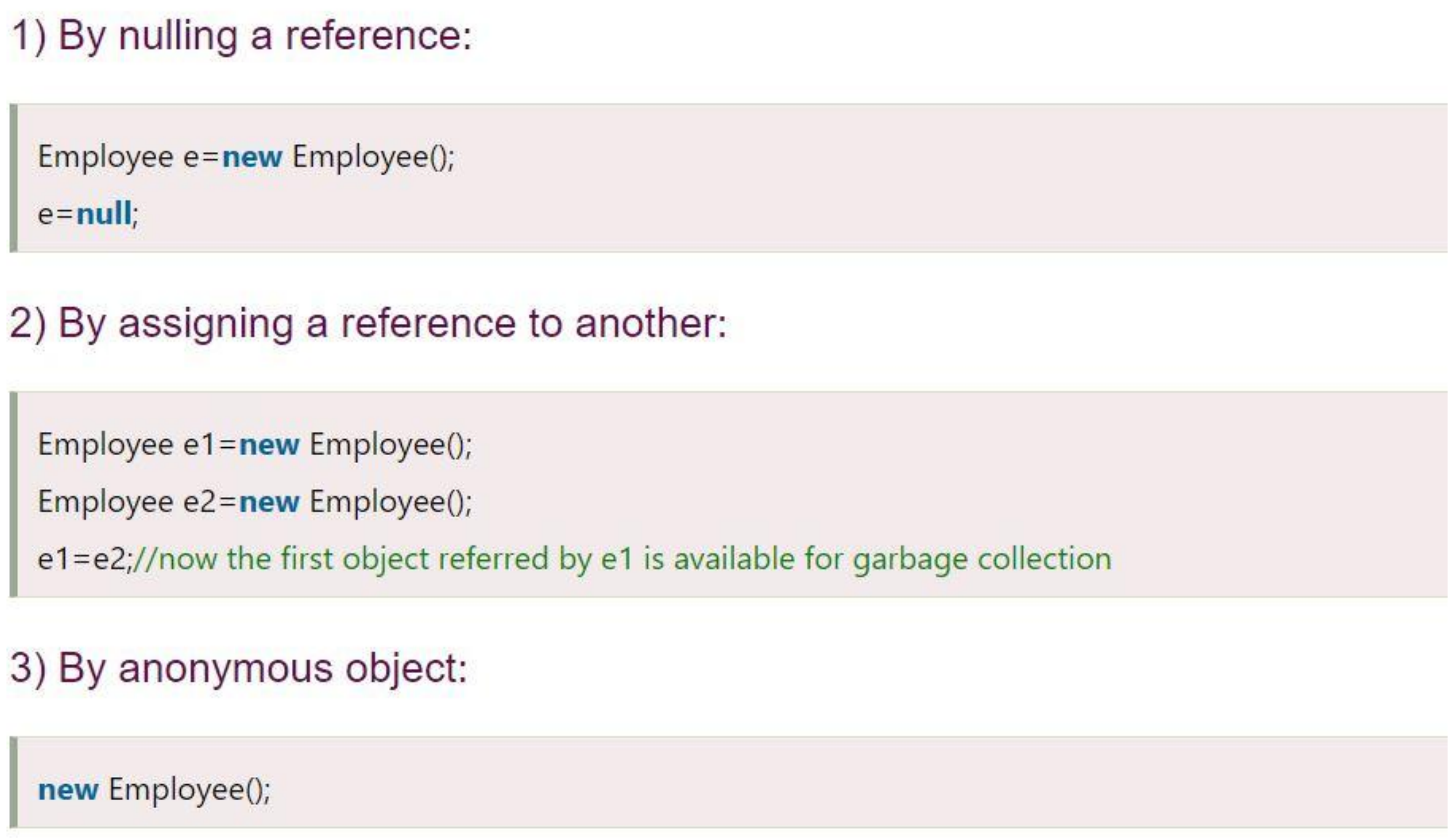
```
Output:task one
       task two
```

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?



1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

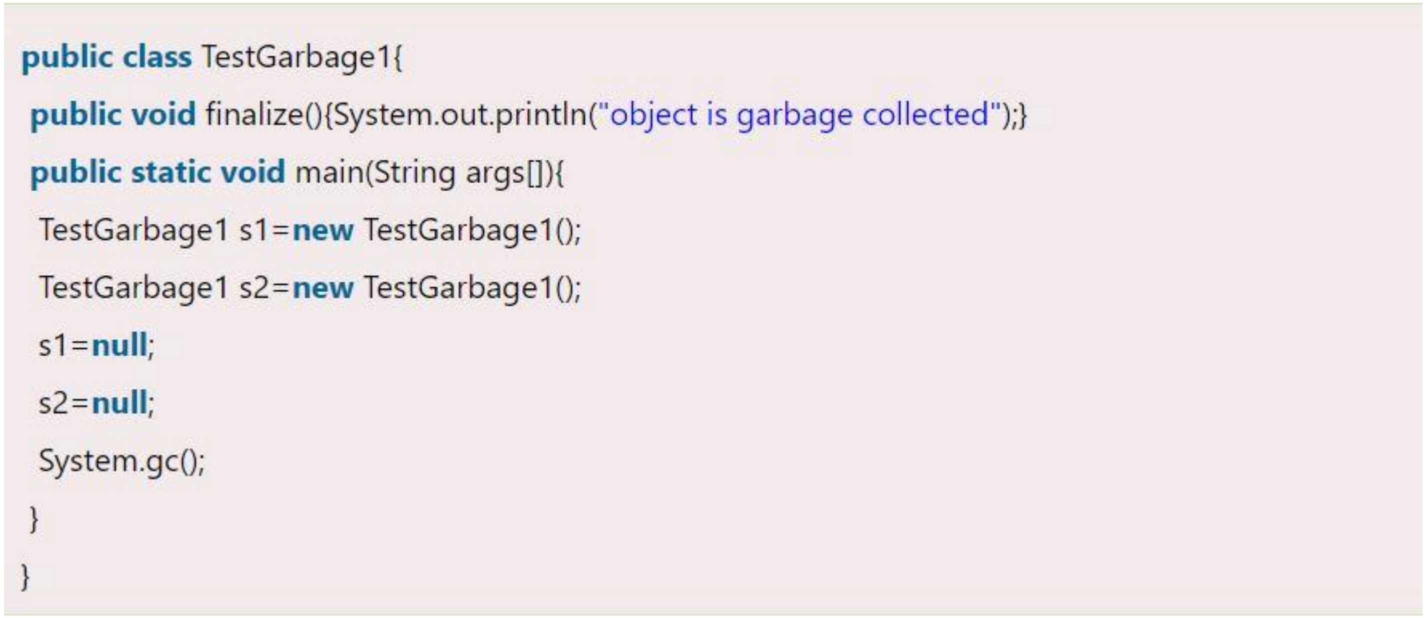
2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

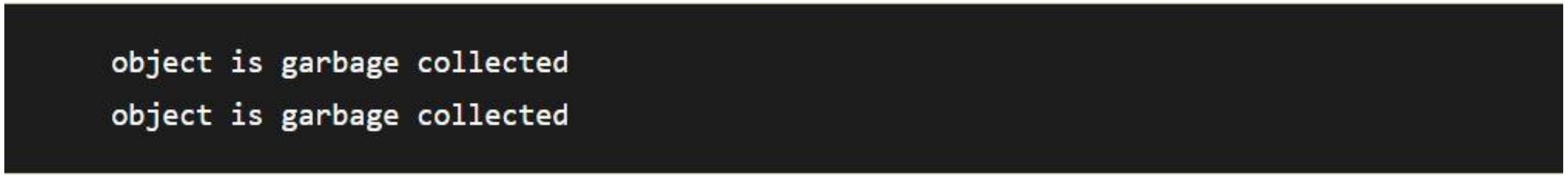
3) By anonymous object:

```
new Employee();
```

Simple Example of garbage collection in java



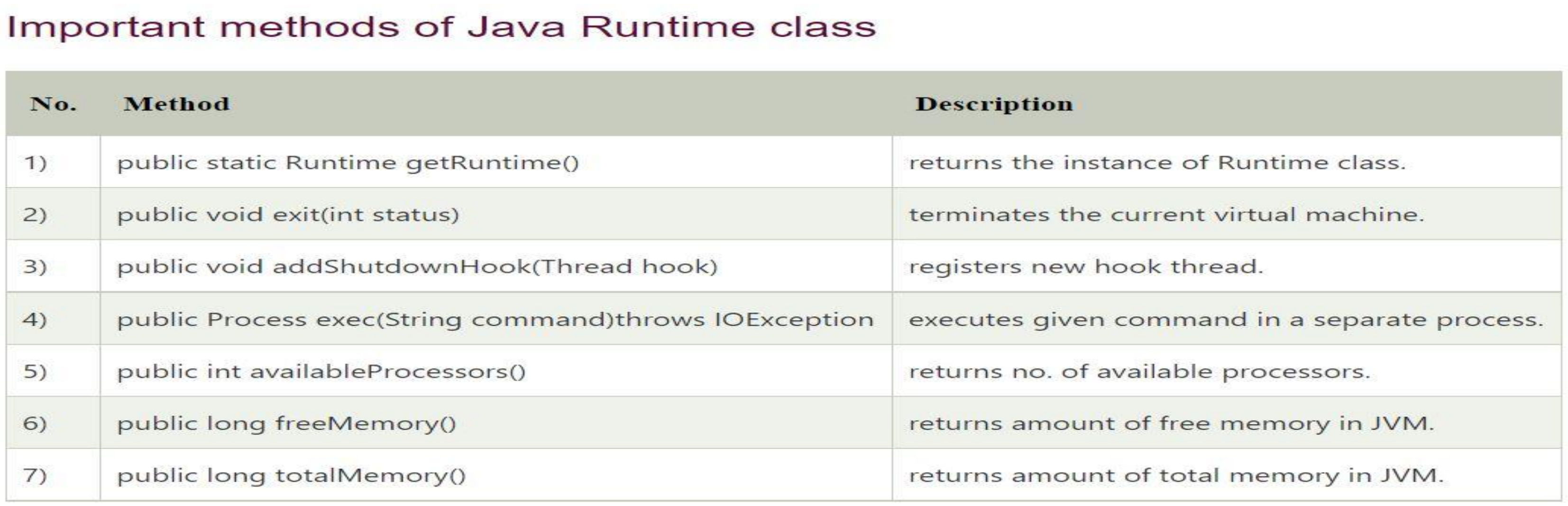
```java
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```



```
object is garbage collected
object is garbage collected
```

Java Runtime class

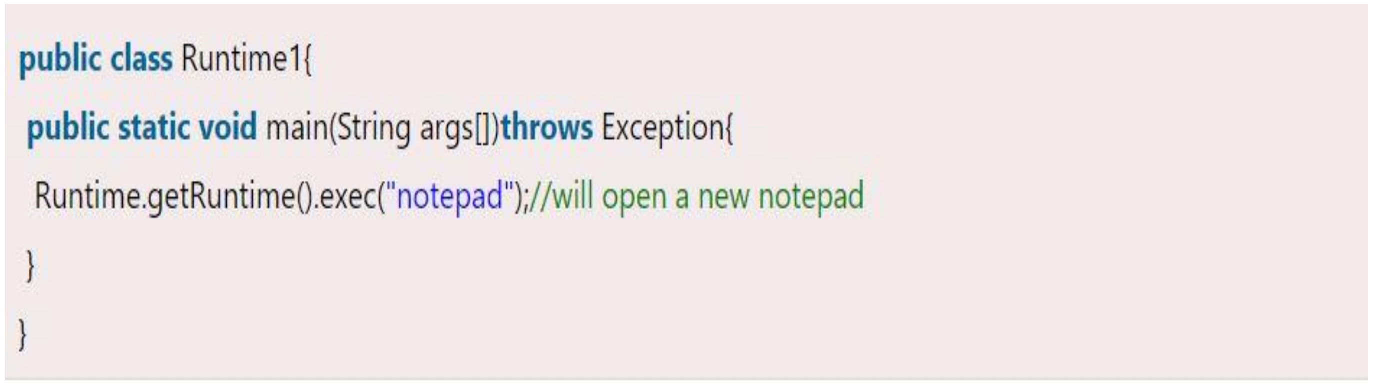
- **Java Runtime** class is used *to interact with java runtime environment*. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of java.lang.Runtime class is available for one java application.
- The **Runtime.getRuntime()**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

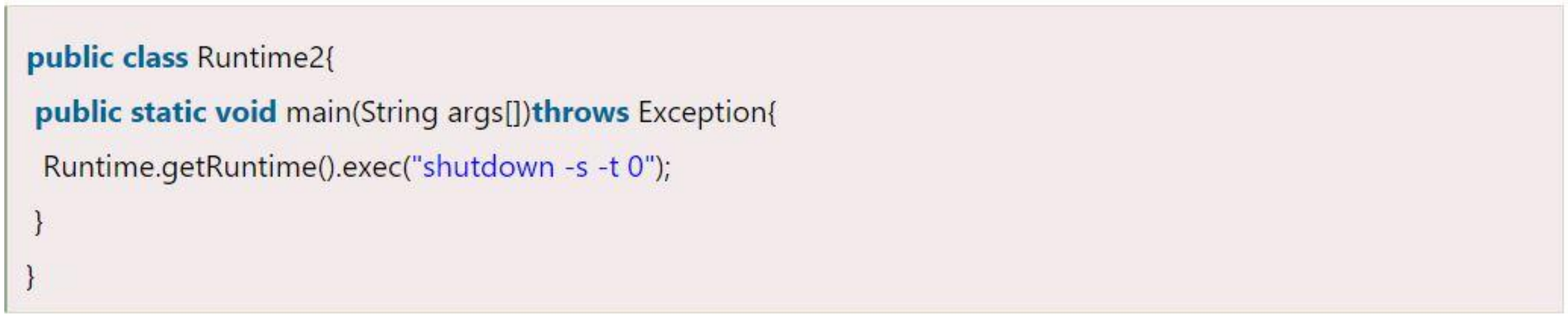
Java Runtime exec() method



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

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.

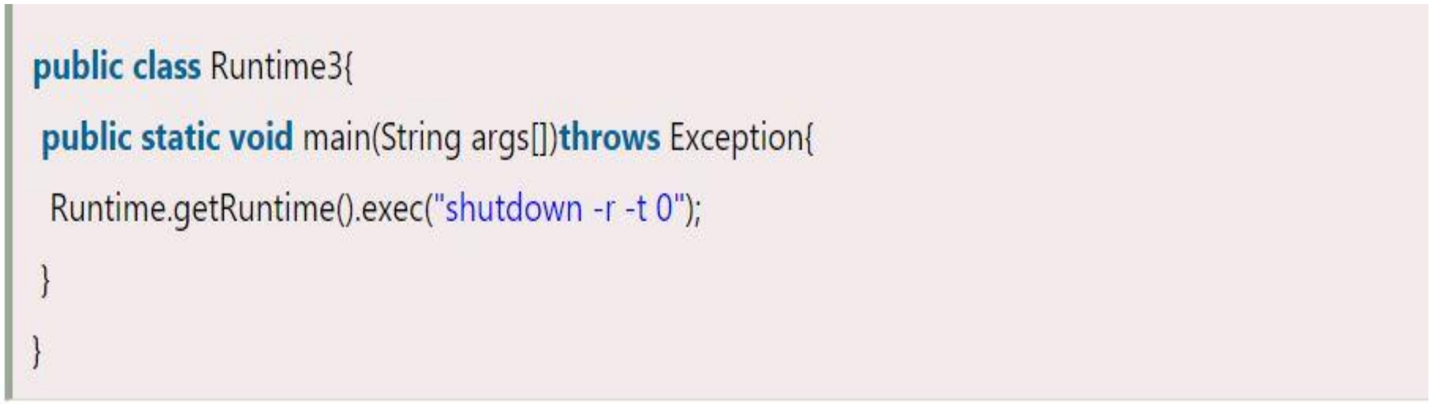


```
public class Runtime2{
 public static void main(String args[])throws Exception{
  Runtime.getRuntime().exec("shutdown -s -t 0");
 }
}
```

How to shutdown windows system in Java

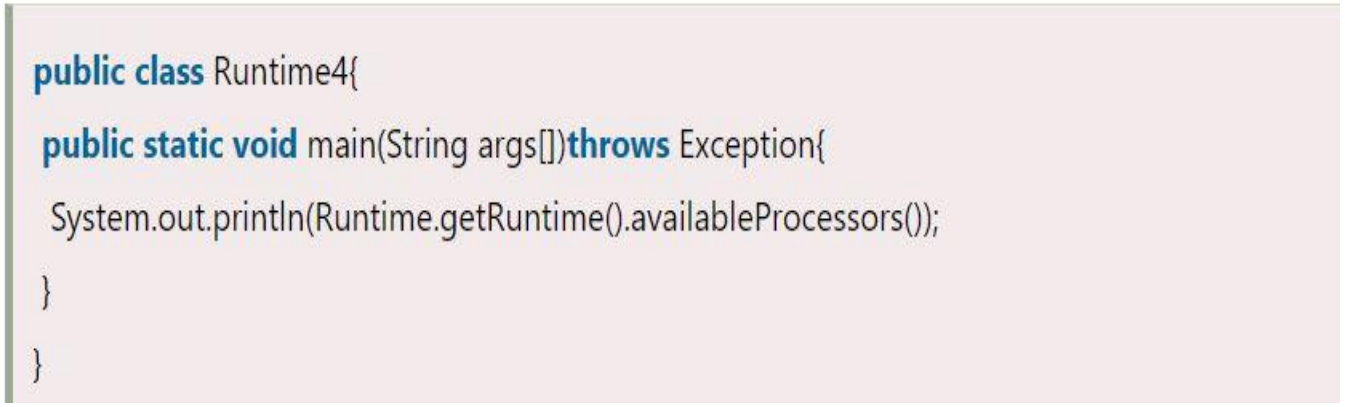
```
public class Runtime2{
 public static void main(String args[])throws Exception{
  Runtime.getRuntime().exec("c:\\Windows\\System32\\shutdown -s -t 0");
 }
}
```

How to restart system in Java



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

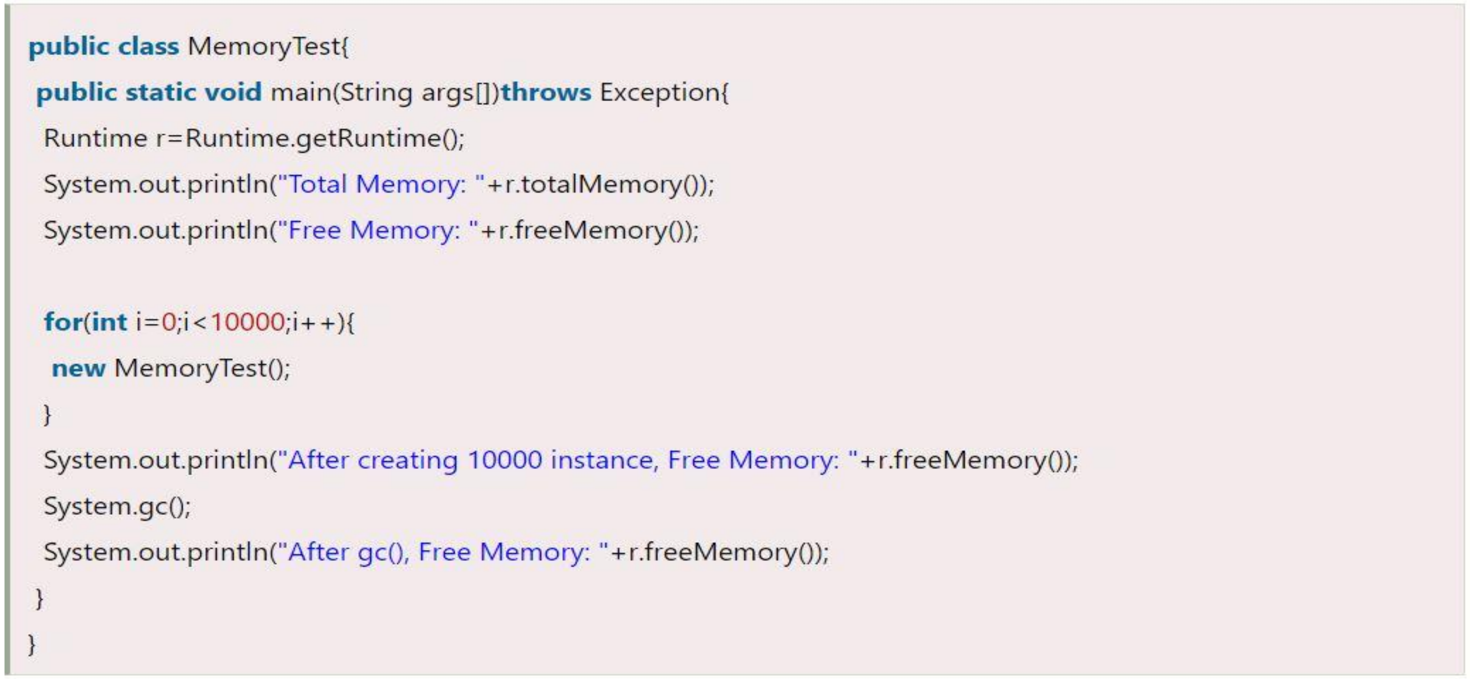
Java Runtime availableProcessors()



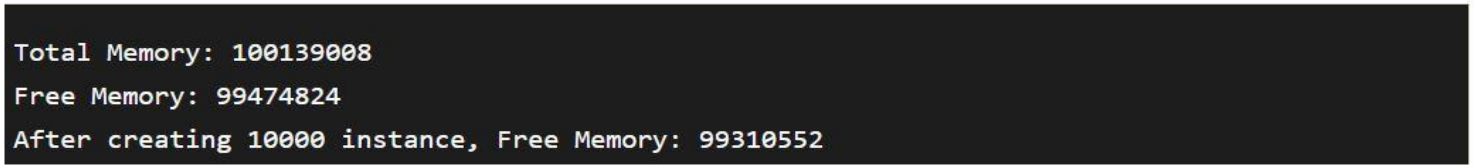
```
public class Runtime4{
 public static void main(String args[])throws Exception{
  System.out.println(Runtime.getRuntime().availableProcessors());
 }
}
```

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.



```
public class MemoryTest{
 public static void main(String args[])throws Exception{
  Runtime r=Runtime.getRuntime();
  System.out.println("Total Memory: "+r.totalMemory());
  System.out.println("Free Memory: "+r.freeMemory());

  for(int i=0;i<10000;i++){
   new MemoryTest();
  }
  System.out.println("After creating 10000 instance, Free Memory: "+r.freeMemory());
  System.gc();
  System.out.println("After gc(), Free Memory: "+r.freeMemory());
 }
}
```



```
Total Memory: 100139008
Free Memory: 99474824
After creating 10000 instance, Free Memory: 99310552
```

Thank You