```
Threasd is LW sub process,smallest unit of
processin..independent from other thread execution..

new->runnable(i.e t.start() invoked)-->running ->terminate..
                        -->Non-runnable(blocked) --> runnable



can we run thread twice: no

Thread creation:
anonymous
implemnt Runnable : public void run
class M implements Runnable{
public void run(){}  }
public staic void main(){
Thread t=new Thread(new M())

extends Thread class : public void run() :
class M extends Thread{}: its object ids direct thread.

Thread(),Thread(String name),Thread(Runnable r),Thread
(Runnable r,"Name")
```

```
MEthds:
Thread.currentThread()
t.suspend(),t.resume("from suspende state"),t.stop
(),isDaemon,setDaemon("true"),isInterupted(),isAlive(),join
(),setPriority(),getName(),setName(),sleep(),run()..


Joining a thread:
t.join() : wait till this gets over ..no other will start
even if Thread.sleep() is invoked on him..

t.isALive() : test whether its running or not..

t.join(1500): this thread will run for initial 1.5 sec..

Thread t=Thread.currentThread();/return runing instance
t.getName();
```

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

```
Sleeping a thread:
Thread.sleep(100) : will let sleep thread for that time..and
meanwhile invoke other waiting thread..

Priority of thread:
Thread.MIN_PRIORITY i.e 0
Thread.MAX_PRIORITY i.e 10
Thread.NORM_PRIORITY i.e 5

t.setPriority(Thread.MAX_PRIORITY) :
t.start();

t.getName(),t.setName("name"),t.getId()...

shutDownHook() : can be used to do cleanup task before JVM
terminate normaly or abruptly..
Runtime r=Runtime.getInstance();
r.addShutdownHook( new MyThread());
r.addShutdownHook(new Runnable(){  void run()...});

What if we call run of thread class directly :
t.run()...it will be in same stack as normal function and
won't create a new stack for thread..

Daemon Thread: only to server user thread..has no other
use..
t.setDaemon("true");   //do this before thread start..
t.start();
t.isDaemon(); : true

How to do single task in multiple thread:
make two class extend thread..have diff run() in it..
create class to create those class object and start() it..

both will do diff task at same time..

class s1 extens Thread{}
class s2 extends Thread{}
class Test { s1 a=new s1();s2 b=new s2();a,b.start().
}
```

```
Thread pool:
ExecutotService and Executors is used for this..
ExecutotService ex=Executors.newFixedThreadPool(5);pool f
5..
ex.execute(thread instance);
ex.shutDown();
ex.awaitTernination();
or (while!ex.isTerminated())
```

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.

Java Multithreading is mostly used in games, animation etc.

Advantage 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)

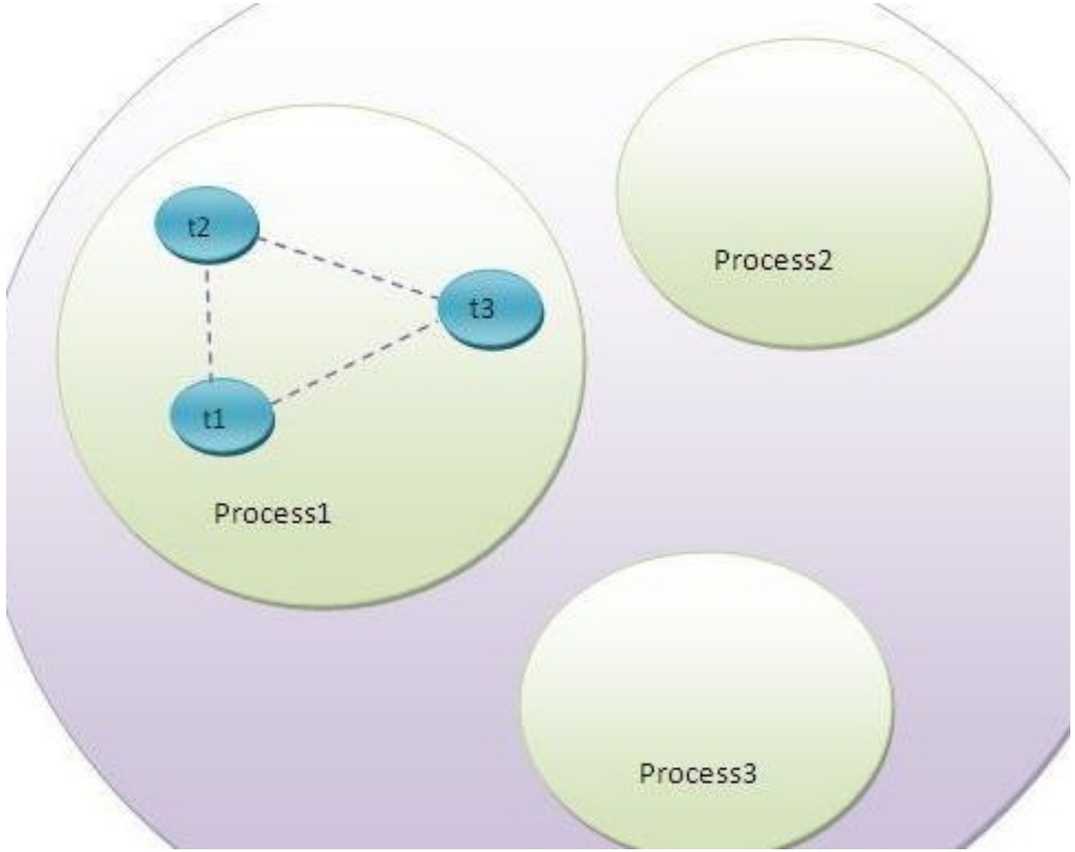
- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

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.



Note: At a time one thread is executed only.

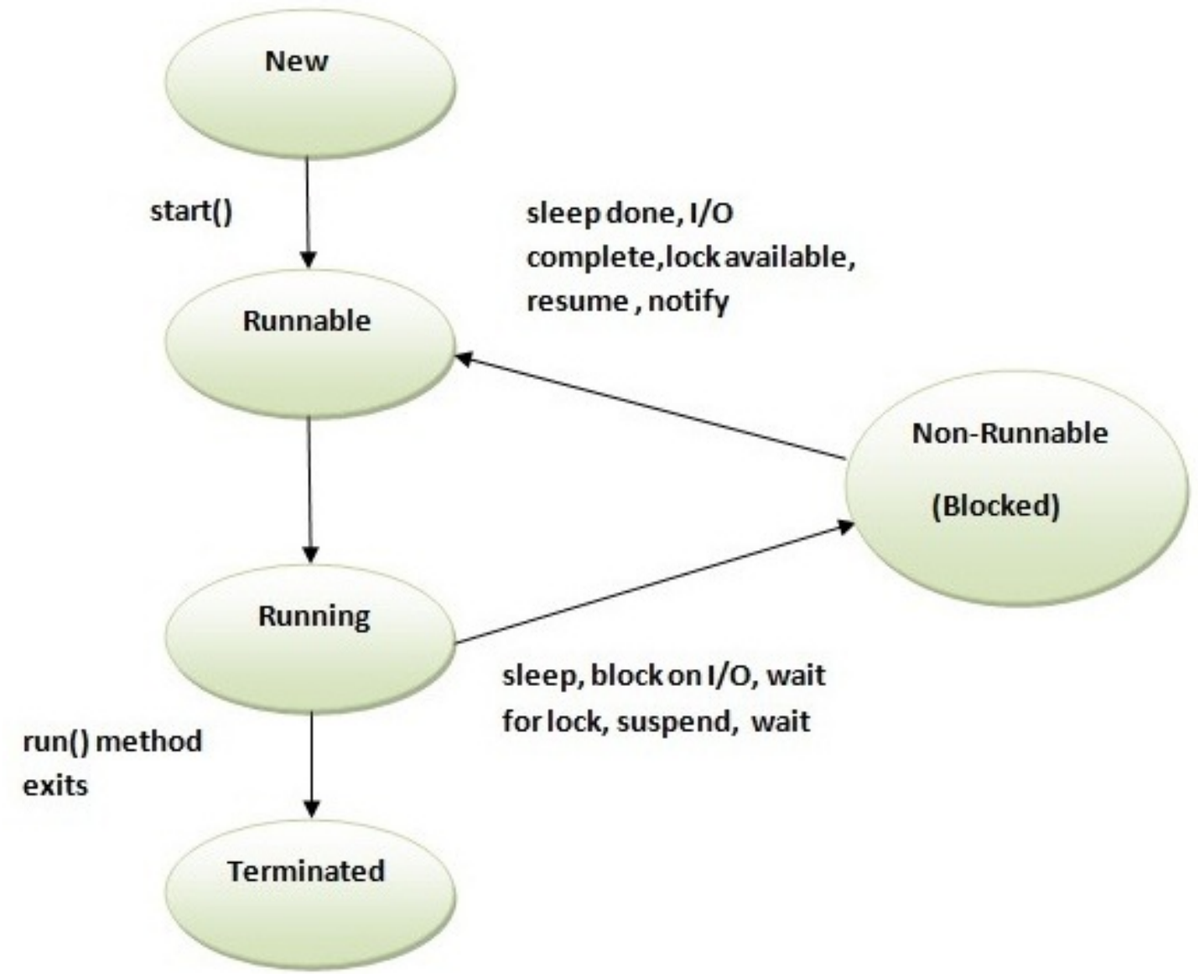
Life cycle of a Thread (Thread States)

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.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

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.

Thread class constructor allocates a new thread object.When you create object of Multi class,your class constructor is invoked(provided by Compiler) fromwhere Thread class constructor is invoked(by super() as first statement).So your Multi class object is thread object now.

1)By extending Thread class:

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

```
Output:thread is running...
```

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

```
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 java sleep() method of Thread class is used to sleep a thread for the specified milliseconds 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 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
       5
       5
```

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:

```
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();
 }
}
```

Test it Now

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

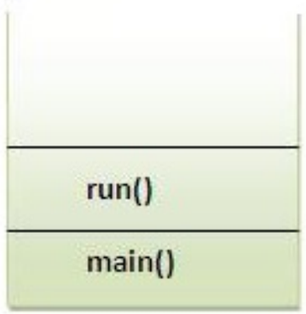
What if we call run() method directly instead start() method?

- 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.

```
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
 }
}
```

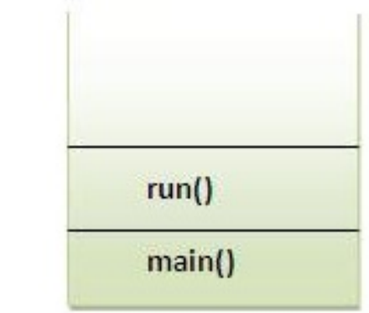
Test it Now

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



Stack
(main thread)

```
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();
 }
}
```



Stack
(main thread)

```
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.

Joining threads

Sometimes one thread needs to know when another thread is ending. In java, **isAlive()** and **join()** are two different methods to check whether a thread has finished its execution.

The **isAlive()** methods return **true** if the thread upon which it is called is still running otherwise it return **false**.

```
final boolean isAlive()
```

But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

```
final void join() throws InterruptedException
```

Using **join()** method, we tell our thread to wait until the specifid thread completes its execution. There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

```
final void join(long milliseconds) throws InterruptedException
```

The following example shows the usage of java.lang.Thread.join() method.

```
package com.tutorialspoint;

import java.lang.*;

public class ThreadDemo implements Runnable {

   public void run() {

      Thread t = Thread.currentThread();
      System.out.print(t.getName());
      //checks if this thread is alive
      System.out.println(", status = " + t.isAlive());
   }

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

      Thread t = new Thread(new ThreadDemo());
      // this will call run() function
      t.start();
      // waits for this thread to die
      t.join();
      System.out.print(t.getName());
      //checks if this thread is alive
      System.out.println(", status = " + t.isAlive());
   }
}
```

Let us compile and run the above program, this will produce the following result:

```
Thread-0, status = true
Thread-0, status = false
```

Naming a thread:

<<prev

The Thread class provides methods to change and get the name of a thread.

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:

```
class TestMultiNaming1 extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  TestMultiNaming1 t1=new TestMultiNaming1();
  TestMultiNaming1 t2=new TestMultiNaming1();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());

  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 changeling name of t1:Sonoo Jaiswal
       running...
```

The currentThread() method:

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

Syntax of currentThread() method:

- **public static Thread currentThread():** returns the reference of currently running thread.
-

Example of currentThread() method:

```
class TestMultiNaming2 extends Thread{
 public void run(){
  System.out.println(Thread.currentThread().getName());
 }
 public static void main(String args[]){
  TestMultiNaming2 t1=new TestMultiNaming2();
  TestMultiNaming2 t2=new TestMultiNaming2();

  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 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:

```
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

File: MyThread.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
```

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

```
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
```

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

```
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
```

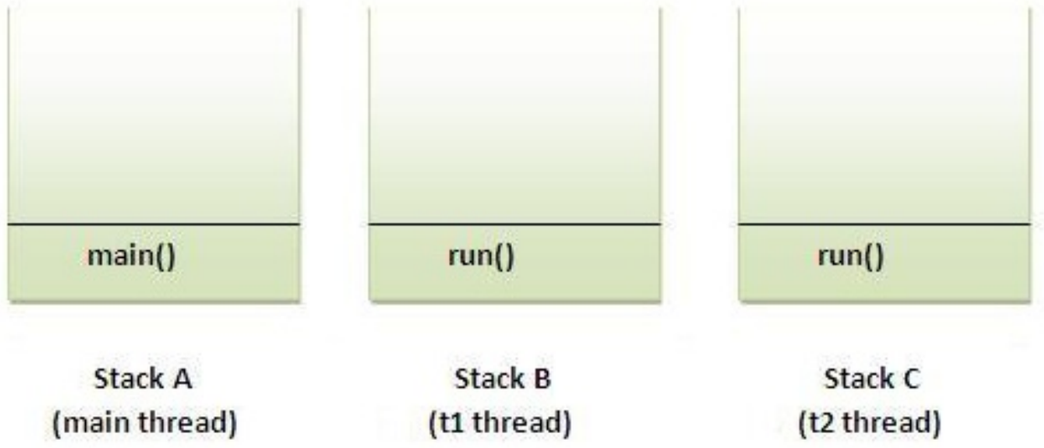
Program of performing single task by multiple threads

```
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();

 }
}
```

Test it Now

```
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

```
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
```

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

```
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();
 }
}
```

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

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

Program of performing two tasks by two threads

```
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 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.

In above program, we are creating fixed size thread pool of 5 worker threads. Then we are submitting 10 jobs to this pool, since the pool size is 5, it will start working on 5 jobs and other jobs will be in wait state, as soon as one of the job is finished, another job from the wait queue will be picked up by worker thread and get's executed.

The output confirms that there are five threads in the pool named from "pool-1-thread-1" to "pool-1-thread-5" and they are responsible to execute the submitted tasks to the pool.

Executors class provide simple implementation of **ExecutorService** using **ThreadPoolExecutor** but ThreadPoolExecutor provides much more feature than that. We can specify the number of threads that will be alive when we create ThreadPoolExecutor instance and we can limit the size of thread pool and create our own **RejectedExecutionHandler** implementation to handle the jobs that can't fit in the worker queue.

Example of Java Thread Pool

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

File: WorkerThrad.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(); }
    }
}
```

File: JavaThreadPoolExample.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");
    }
}
```

```
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
```

In above program, we are creating fixed size thread pool of 5 worker threads. Then we are submitting 10 jobs to this pool, since the pool size is 5, it will start working on 5 jobs and other jobs will be in wait state, as soon as one of the job is finished, another job from the wait queue will be picked up by worker thread and get's executed.

```
package de.vogella.concurrency.threadpools;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
  private static final int NTHREDS = 10;

  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    for (int i = 0; i < 500; i++) {
      Runnable worker = new MyRunnable(10000000L + i);
      executor.execute(worker);
    }
    // This will make the executor accept no new threads
    // and finish all existing threads in the queue
    executor.shutdown();
    // Wait until all threads are finish
    executor.awaitTermination();
    System.out.println("Finished all threads");
  }
}
```

8. Futures and Callables

The code examples for this section are created in a Java project called *de.vogella.concurrency.callables*.

The executor framework presented in the last chapter works with `Runnables`. Runnable do not return result.

In case you expect your threads to return a computed result you can use `java.util.concurrent.Callable`. The `Callable` object allows to return values after completion.

The `Callable` object uses generics to define the type of object which is returned.

If you submit a `Callable` object to an `Executor` the framework returns an object of type `java.util.concurrent.Future`. This `Future` object can be used to check the status of a `Callable` and to retrieve the result from the `Callable`.

On the `Executor` you can use the method submit to submit a `Callable` and to get a future. To retrieve the result of the future use the `get()` method.

```
package de.vogella.concurrency.callables;

import java.util.concurrent.Callable;

public class MyCallable implements Callable<Long> {
  @Override
  public Long call() throws Exception {
    long sum = 0;
    for (long i = 0; i <= 100; i++) {
      sum += i;
    }
    return sum;
  }

}
```

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(Runnable r) method

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

```
public void addShutdownHook(Runnable r){}
```

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..
```

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

Same example of Shutdown Hook by annonymous class:

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

Runtime r=Runtime.getRuntime();

r.addShutdownHook(new Runnable(){
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) {}
}
}
```

Test it Now

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

Example of `isAlive` method

```
public class MyThread extends Thread
{
public void run()
   {
       System.out.println("r1 ");
       try{
        Thread.sleep(500);
    }catch(InterruptedException ie){}

       System.out.println("r2 ");
  }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();
t2.start();
System.out.println(t1.isAlive());
System.out.println(t2.isAlive());
}
}
```

Output

```
r1
true
true
r1
r2
r2
```

Output

```
r1
r1
r2
r2
```

Example of thread without `join()` method

```
public class MyThread extends Thread
{
public void run()
   {
       System.out.println("r1 ");
       try{
        Thread.sleep(500);
    }catch(InterruptedException ie){}

       System.out.println("r2 ");
  }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();
t2.start();
}
}
```

Example of thread with `join()` method

```
public class MyThread extends Thread
{
public void run()
   {
       System.out.println("r1 ");
       try{
        Thread.sleep(500);
    }catch(InterruptedException ie){}

       System.out.println("r2 ");
  }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();

try{
  t1.join();            //Waiting for t1 to finish
}catch(InterruptedException ie){}

t2.start();
}
}
```

```
r1
r2
r1
r2
```

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

Example of join(long miliseconds) method

```
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();
 }
}
```

Specifying time with join()

If in the above program, we specify time while using **join()** with **m1**, then **m1** will execute for that time, and then **m2** and **m3** will join it.

```
m1.join(1500);
```

Doing so, initially m1 will execute for 1.5 seconds, after which m2 and m3 will join it.