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.

MultiTHreading

Thread

Thread can be called lightweight process. Thread requires less resources to create and exists in the process, thread shares the process resources.

Every java application has at least one thread – main thread. Although there are so many other java threads running in background like memory management, system management, signal processing etc. But from application point of view – main is the first java thread and we can create multiple threads from it.

Multithreading refers to two or more threads executing concurrently in a single program. A computer single core processor can execute only one thread at a time and time slicing is the OS feature to share processor time between different processes and threads.

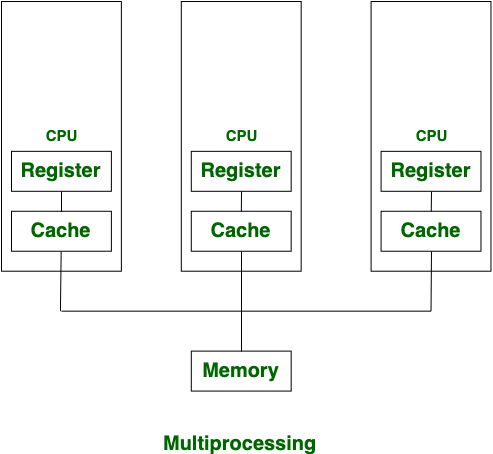
The typical difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space.

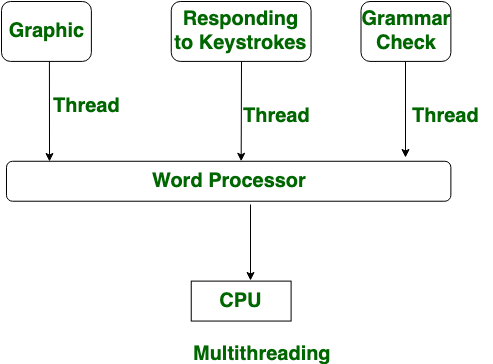
## Difference between Multiprocessing and Multithreading

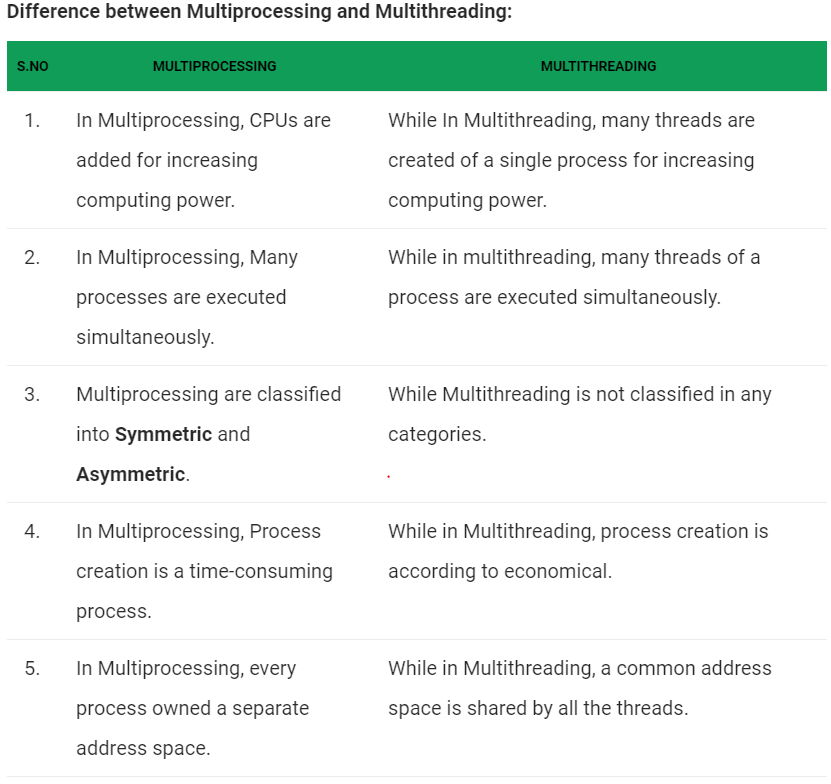
Both **Multiprocessing** and **Multithreading** are used to increase the computing power of a system.

**Multiprocessing:**  
Multiprocessing is a system that has more than one or two processors. In Multiprocessing, CPUs are added for increasing computing speed of the system. Because of Multiprocessing, There are many processes are executed simultaneously. Multiprocessing are classified into two categories:



**Multithreading:**  
Multithreading is a system in which multiple threads are created of a process for increasing the computing speed of the system. In multithreading, many threads of a process are executed simultaneously and process creation in multithreading is done according to economical.



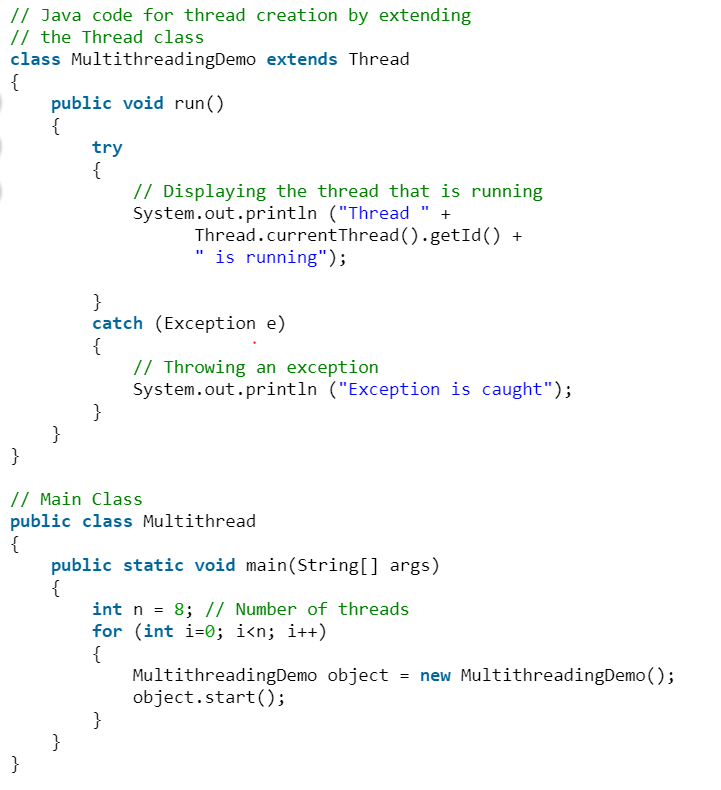


## Multithreading in Java

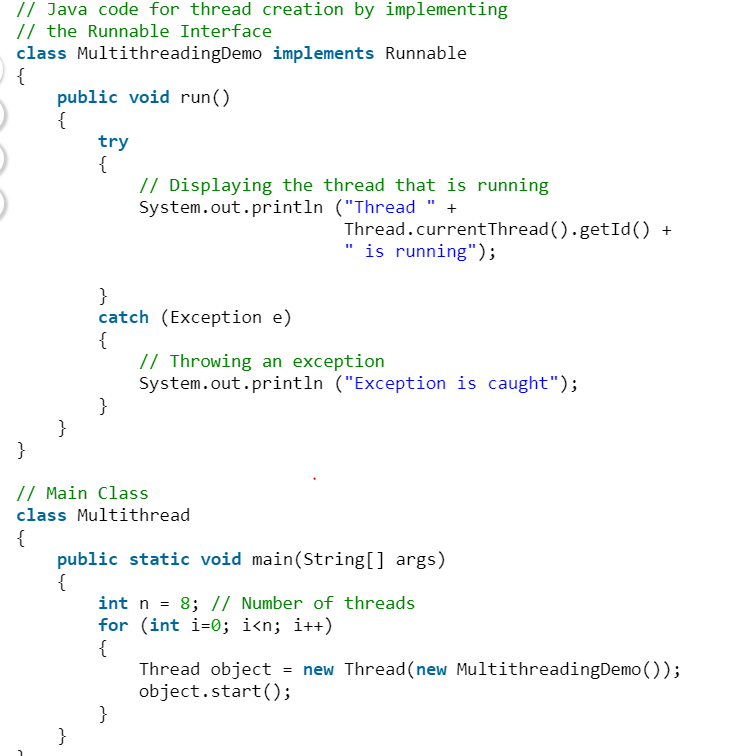
Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.  
a process in divided into smaller process.

Threads can be created by using two mechanisms :  
1. Extending the Thread class  
2. Implementing the Runnable Interface

**Thread creation by extending the Thread class**  
  
We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.



**Thread creation by implementing the Runnable Interface**  
  
We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.



**Thread Class vs Runnable Interface**  
1. If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.  
  
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

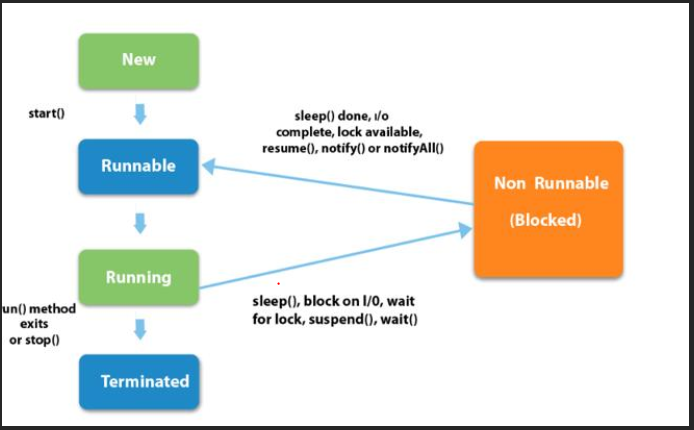
**Can we start a thread twice**

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.

## **Lifecycle and States of a Thread in Java**

A [thread](http://www.geeksforgeeks.org/multithreading-in-java/) in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant: (New ,Runnable,Running,blocked/waiting , terminated)

1. New
2. Runnable
3. running
4. Blocked/Waiting
5. Terminated



1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it’s code is yet to be run and hasn’t started to execute.
2. **Runnable State:** A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.  
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.
3. **Blocked/Waiting state:**When a thread is temporarily inactive, then it’s in one of the following states:

* Blocked
* Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It’s the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states does not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run

1. **Timed Waiting:** A thread lies in timed waiting state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
2. **Terminated State:** A thread terminates because of either of the following reasons:

* Because it exists normally. This happens when the code of thread has entirely executed by the program.
* Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

A thread that lies in a terminated state does no longer consumes any cycles of CPU.

**Implementing Thread States in Java**

In Java, to get the current state of the thread, use **Thread.getState()** method to get the current state of the thread. Java provides **java.lang.Thread.State** class that defines the ENUM constants for the state of a thread

------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.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.(In computing, preemption is the act of temporarily interrupting a task being carried out by a computer system)

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.

## 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 defined in Thread class:

public static int MIN\_PRIORITY

public static int NORM\_PRIORITY

public static int MAX\_PRIORITY (high runs first)

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

TestMultiPriority1 m2=new TestMultiPriority1();

m1.setPriority(Thread.MIN\_PRIORITY);

## 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.

**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.

Methods for Java Daemon thread by Thread class

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

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.

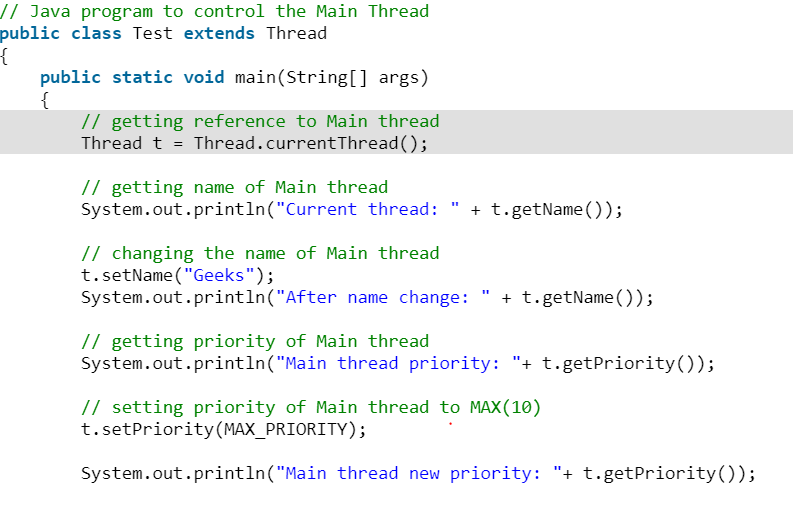
Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException

**Main Thread**

When a Java program starts up, one thread begins running immediately. This is usually called the *main* thread of our program, because it is the one that is executed when our program begins.

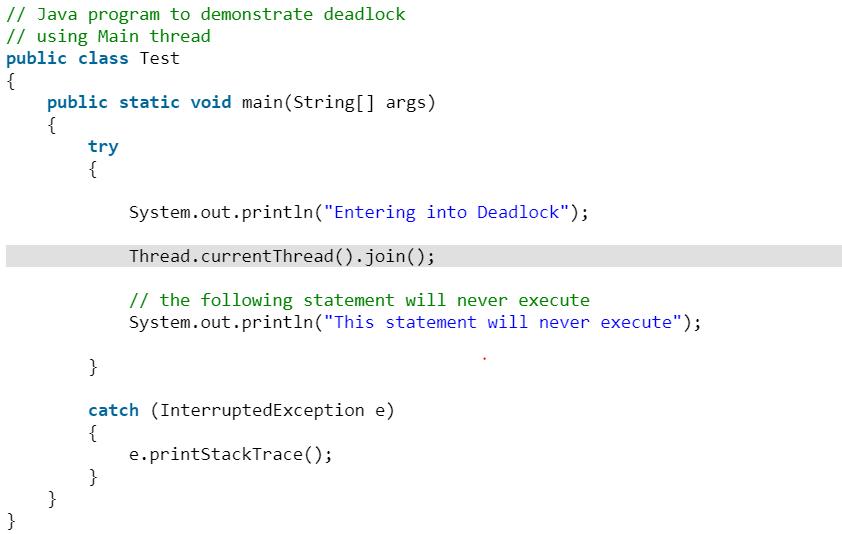
**Properties :**

* It is the thread from which other “child” threads will be spawned.
* Often, it must be the last thread to finish execution because it performs various shutdown actions
* **How to control Main thread**
* The main thread is created automatically when our program is started. To control it we must obtain a reference to it. This can be done by calling the method currentThread( ) which is present in Thread class. This method returns a reference to the thread on which it is called. The default priority of Main thread is 5 and for all remaining user threads priority will be inherited from parent to child.



**Deadlocking with use of Main Thread(only single thread)**

We can create a deadlock by just using Main thread, i.e. by just using a single thread. The following java program demonstrate this.



**Explanation :**  
The statement “Thread.currentThread().join()”, will tell Main thread to wait for this thread(i.e. wait for itself) to die. Thus Main thread wait for itself to die, which is nothing but a deadlock.

## What does start() function do in multithreading in Java?

In both the approaches, we override the run() function, but we start a thread by calling the start() function. So why don’t we directly call the oveerridden run() function? Why always the start function is called to execute a thread?

**What happens when a function is called?**  
When a function is called the following operations take place:

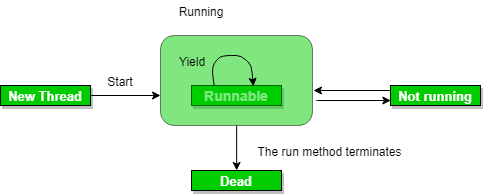
1. The arguments are evaluated.
2. A new stack frame is pushed into the call stack.
3. Parameters are initialized.
4. Method body is executed.
5. Value is retured and current stack frame is popped from the call stack.

**The purpose of start() is to create a separate call stack for the thread. A separate call stack is created by it, and then run() is called by JVM.**

## Java Concurrency – yield(), sleep() and join() methods

We can prevent the execution of a thread by using one of the following methods of Thread class.

1. **yield():** Suppose there are three threads t1, t2, and t3. Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state. Completion time for thread t1 is 5 hour and completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job. In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent execution of a thread in between if something important is pending. yeild() helps us in doing so.  
   **yield()**basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.



**Use of yield method:**

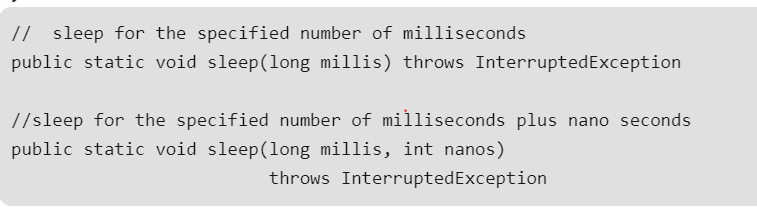
* Whenever a thread calls java.lang.Thread.yield method, it gives hint to the thread scheduler that it is ready to pause its execution. Thread scheduler is free to ignore this hint.
* If any thread executes yield method , thread scheduler checks if there is any thread with same or high priority than this thread. If processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give processor to other thread and if not – current thread will keep executing.



**Note:**

* Once a thread has executed yield method and there are many threads with same priority is waiting for processor, then we can't specify which thread will get execution chance first.
* The thread which executes the yield method will enter in the Runnable state from Running state.
* Once a thread pauses its execution, we can't specify when it will get chance again it depends on thread scheduler.

**sleep():** This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.





Note:

* Based on the requirement we can make a thread to be in sleeping state for a specified period of time
* Sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).

**yield() vs sleep()**

**yield:()** indicates that the thread is not doing anything particularly important and if any other threads or processes need to be run, they can. Otherwise, the **current thread will continue to run.**

**sleep()**: causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, **the CPU will be idle** (and probably enter a power saving mode).

[**join():**](https://www.geeksforgeeks.org/joining-threads-in-java/) The join() method of a Thread instance is used to join the start of a thread’s execution to end of other thread’s execution such that a thread does not start running until another thread ends. If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing.  
The join() method waits at most this much milliseconds for this thread to die. A timeout of 0 means to wait forever  
Syntax:

// waits for this thread to die.

public final void join() throws InterruptedException

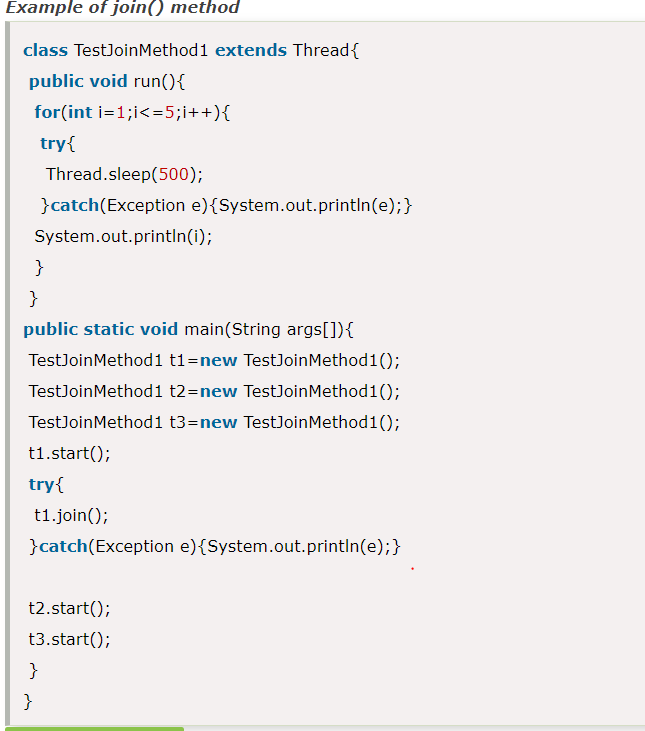
// waits at most this much milliseconds for this thread to die

public final void join(long millis)

throws InterruptedException

// waits at most milliseconds plus nanoseconds for this thread to die.

The java.lang.Thread.join(long millis, int nanos)



Note:

* If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.
* Giving a timeout within join(), will make the join() effect to be nullified after the specific timeout.

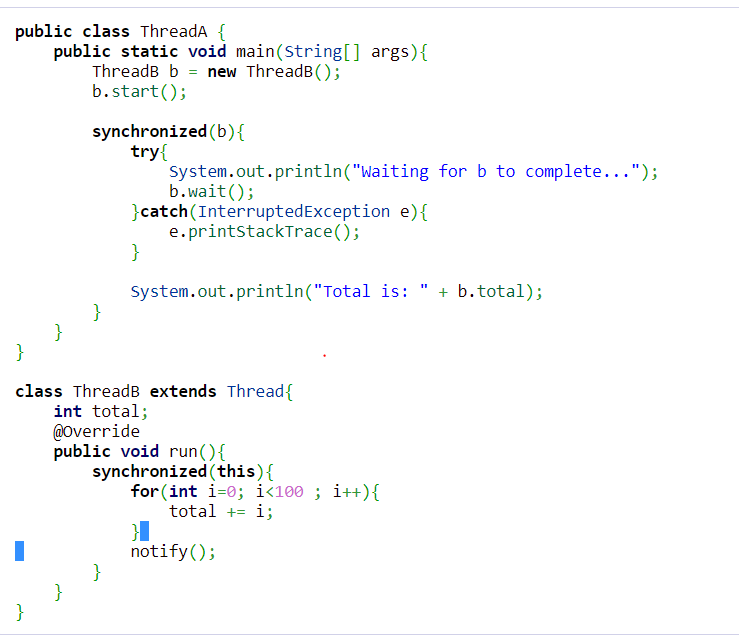
## Inter-thread Communication in Java

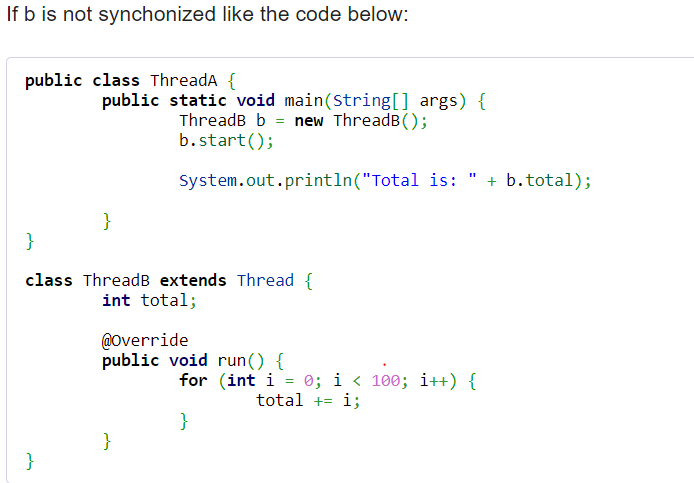
**What is Polling and what are problems with it?**  
The process of testing a condition repeatedly till it becomes true is known as polling.

Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, certain action is taken. This waste many CPU cycles and makes the implementation inefficient.  
For example, in a classic queuing problem where one thread is producing data and other is consuming it.

**How Java multi threading tackles this problem?**  
To avoid polling, Java uses three methods, namely, **wait(), notify() and notifyAll().**  
All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.

* **wait()-**It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
* **notify()-**It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource.
* **notifyAll()-**It wakes up all the threads that called wait() on the same object.





## Thread Pools in Java

**Background**

Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks. An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread. While this approach seems simple to implement, it has significant disadvantages. A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.

Since active threads consume system resources, a [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.

**What is ThreadPool in Java?**

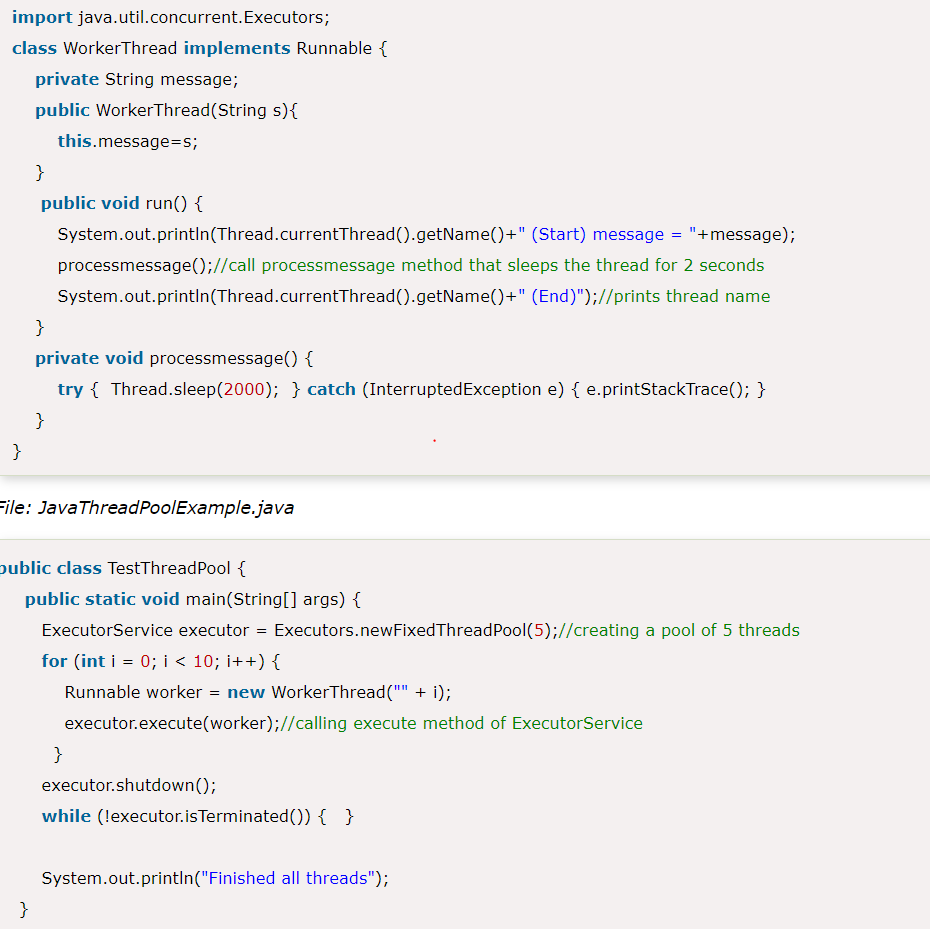
**A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.** Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

**Risks in using Thread Pools**

* + 1. [**Deadlock**](https://www.geeksforgeeks.org/deadlock-in-java-multithreading/)**:**While deadlock can occur in any multi-threaded program, thread pools introduce another case of deadlock, one in which all the executing threads are waiting for the results from the blocked threads waiting in the queue due to the unavailability of threads for execution.
    2. **Thread Leakage :**Thread Leakage occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. As an example, if the thread throws an exception and pool class does not catch this exception, then the thread will simply exit, reducing the size of the thread pool by one. If this repeats many times, then the pool would eventually become empty and no threads would be available to execute other requests.
    3. **Resource Thrashing :**If the thread pool size is very large then time is wasted in context switching between threads. Having more threads than the optimal number may cause starvation problem leading to resource thrashing as explained.

**Important Points**

* + 1. Don’t queue tasks that concurrently wait for results from other tasks. This can lead to a situation of deadlock as described above.
    2. Be careful while using threads for a long lived operation. It might result in the thread waiting forever and would eventually lead to resource leakage.
    3. The Thread Pool has to be ended explicitly at the end. If this is not done, then the program goes on executing and never ends. Call shutdown() on the pool to end the executor. If you try to send another task to the executor after shutdown, it will throw a RejectedExecutionException.
    4. One needs to understand the tasks to effectively tune the thread pool. If the tasks are very contrasting then it makes sense to use different thread pools for different types of tasks so as to tune them properly.
    5. You can restrict maximum number of threads that can run in JVM, reducing chances of JVM running out of memory.
    6. If you need to implement your loop to create new threads for processing, using ThreadPool will help to process faster, as ThreadPool does not create new Threads after it reached it’s max limit.
    7. After completion of Thread Processing, ThreadPool can use the same Thread to do another process(so saving the time and resources to create another Thread.)

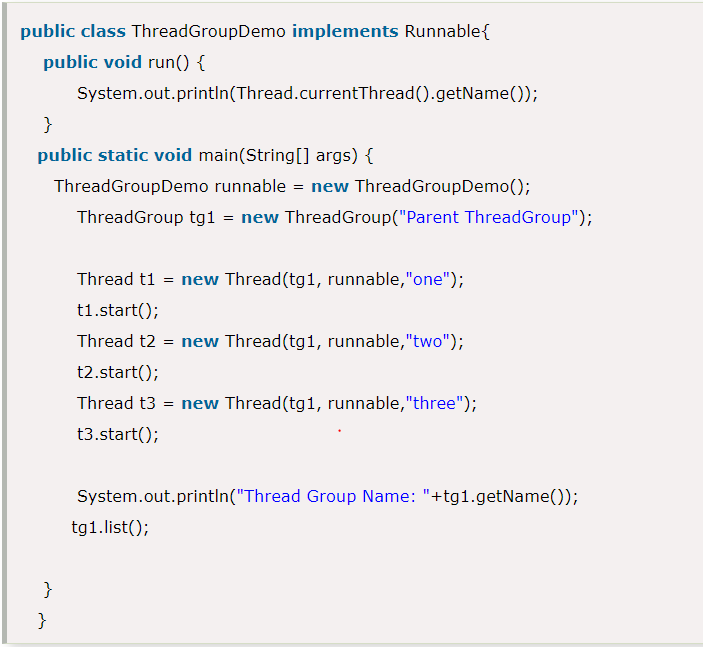


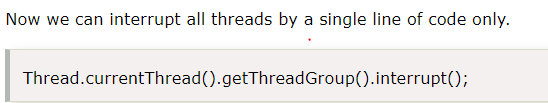
# ThreadGroup in Java

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.

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

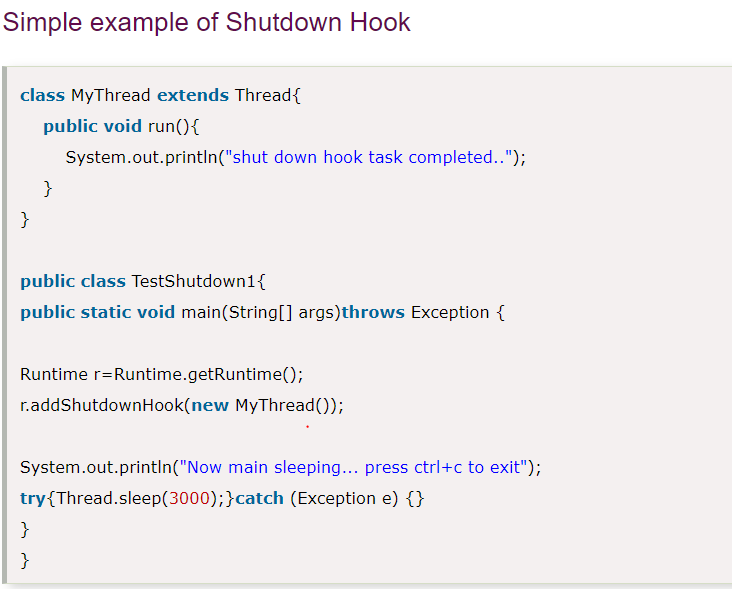
A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.





## 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.



## What is Synchronization?

In a multi-threaded environment, it is possible that more than one thread may try to access the same resource. For example, two threads trying to write in to the same text file. In the absence of any synchronization between them, it is possible that the data written to the file will be corrupt when two or more threads have write access to the same file.

Also, in the JVM, each thread stores a local copy of variables on its stack. The actual value of these variables may be changed by some other thread. But that value may not be refreshed in another thread's local copy. This may cause incorrect execution of programs and non-deterministic behavior.

To avoid such issues, Java provides us with the synchronized keyword, which acts like a lock to a particular resource. This helps achieve communication between threads such that only one thread accesses the synchronized resource and other threads wait for the resource to become free.

The synchronized keyword can be used in a few different ways, like a synchronized block:

synchronized (someObject) {

*// Thread-safe code here*

}

It can also be used with a method like this:

public synchronized void somemMethod() {

*// Thread-safe code here*

}

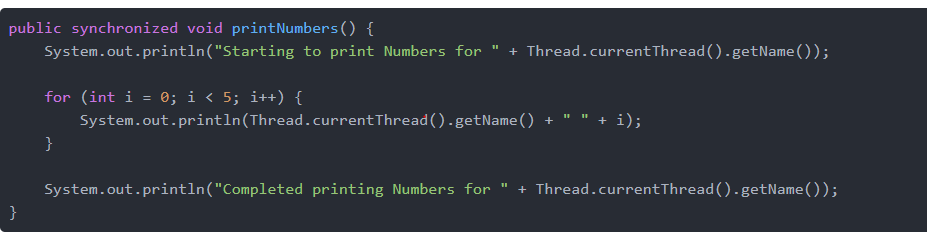
### How Synchronization Works in the JVM

When a thread tries to enter the synchronized block or method, it has to acquire a [lock](https://en.wikipedia.org/wiki/Lock_(computer_science)) on the object being synchronized. One and only one thread can acquire that lock at a time and execute code in that block.

If another thread tries to access a synchronized block before the current thread completes its execution of the block, it has to wait. When the current thread exits the block, the lock is automatically released and any waiting thread can acquire that lock and enter the synchronized block:

* For a synchronized block, the lock is acquired on the object specified in the parentheses after the synchronized keyword
* For a synchronized static method, the lock is acquired on the .class object
* For a synchronized instance method, the lock is acquired on the current instance of that class i.e. this instance

## Synchronized Methods

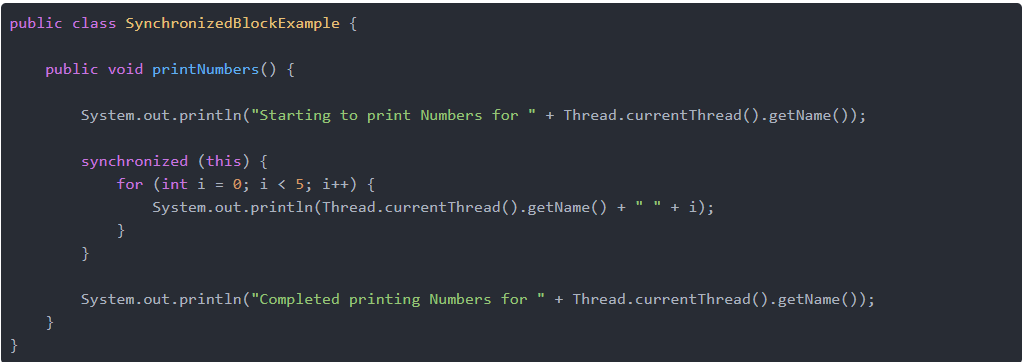


## Synchronized Blocks

However, we can try to *reduc* the amount of code to be executed in a synchronized fashion by keeping the least amount of code as possible in the scope of synchronized. There could be many scenarios where instead of synchronizing on the whole method, it is okay to just synchronize a few lines of code in the method instead.

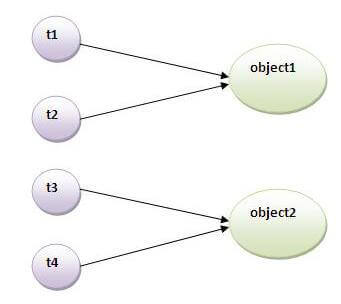
We can use the synchronized block to enclose only that portion of code instead of the whole method.

Since there is less amount of code to be executed inside the synchronized block, the lock is released by each of the threads more quickly. As a result, the other threads spend less time waiting for the lock and code throughput increases greatly



## Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.

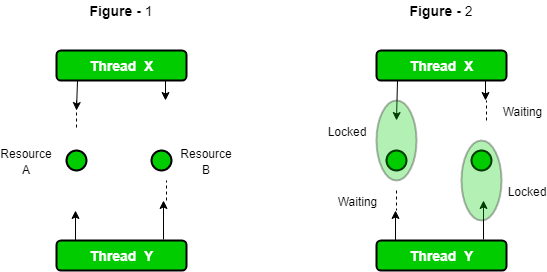


### **Problem without static synchronization**

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.If you make any static method as synchronized, the lock will be on the class not on object.

# Deadlock in Java Multithreading

[**synchronized**](http://quiz.geeksforgeeks.org/synchronized-in-java/) keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock.  
It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called [**Deadlock**](http://quiz.geeksforgeeks.org/operating-system-process-management-deadlock-introduction/). Below is a simple example of Deadlock condition.



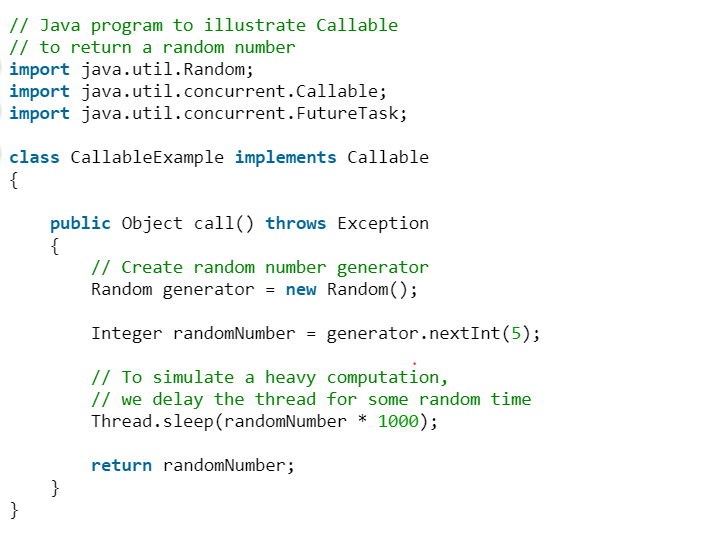
# Callable and Future in Java

**The need for Callable**

There are two ways of creating threads – one by extending the Thread class and other by creating a thread with a Runnable. However, one feature lacking in  Runnable is that we cannot make a thread return result when it terminates, i.e. when run() completes. For supporting this feature, the Callable interface is present in Java.

**Callable vs Runnable**

* For implementing Runnable, the run() method needs to be implemented which does not return anything, while for a Callable, the call() method needs to be implemented which returns a result on completion. Note that a thread can’t be created with a Callable, it can only be created with a Runnable.
* Another difference is that the call() method can throw an exception whereas run() cannot.



**Future**

When the call() method completes, answer must be stored in an object known to the main thread, so that the main thread can know about the result that the thread returned. How will the program store and obtain this result later? For this, a Future object can be used. Think of a Future as an object that holds the result – it may not hold it right now, but it will do so in the future (once the Callable returns). Thus, a Future is basically one way the main thread can keep track of the progress and result from other threads. To implement this interface, 5 methods have to be overridden,

// Java program to illustrate Callable and FutureTask

// for random number generation

import java.util.Random;

import java.util.concurrent.Callable;

import java.util.concurrent.FutureTask;

class CallableExample implements Callable

{

  public Object call() throws Exception

  {

    Random generator = new Random();

    Integer randomNumber = generator.nextInt(5);

    Thread.sleep(randomNumber \* 1000);

    return randomNumber;

  }

}

public class CallableFutureTest

{

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

  {

    // FutureTask is a concrete class that

    // implements both Runnable and Future

    FutureTask[] randomNumberTasks = new FutureTask[5];

    for (int i = 0; i < 5; i++)

    {

      Callable callable = new CallableExample();

      // Create the FutureTask with Callable

      randomNumberTasks[i] = new FutureTask(callable);

      // As it implements Runnable, create Thread

      // with FutureTask

      Thread t = new Thread(randomNumberTasks[i]);

      t.start();

    }

    for (int i = 0; i < 5; i++)

    {

      // As it implements Future, we can call get()

      System.out.println(randomNumberTasks[i].get());

      // This method blocks till the result is obtained

      // The get method can throw checked exceptions

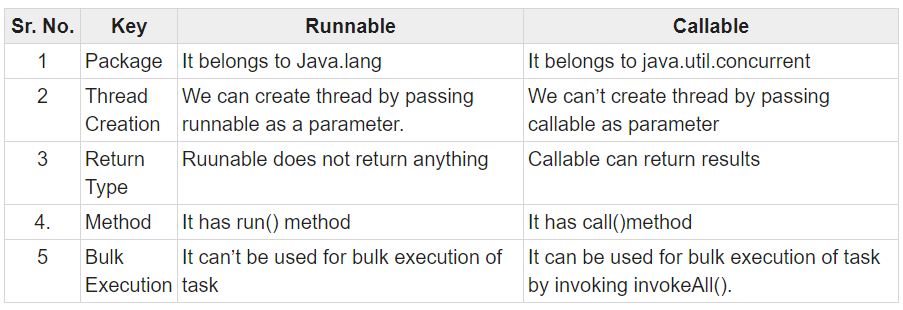
      // like when it is interrupted. This is the reason

      // for adding the throws clause to main

    }

  }

}



## Callable vs Runnable interface in Java

As I explained major differences between a Callable and Runnable interface in the last section. Sometimes this question is also asked as the difference between call() and run() method in Java. All the points discussed here is equally related to that question as well. Let's see them in point format for better understanding :  
  
1) The Runnable interface is older than Callable, there from JDK 1.0, while Callable is added on Java 5.0.  
  
2) Runnable interface has run() method to define task while Callable interface uses call() method for task definition.  
  
3) run() method does not return any value, it's return type is void while call method returns value. The Callable interface is a [generic parameterized interface](http://javarevisited.blogspot.sg/2012/08/how-to-write-parametrized-class-method-Generic-example.html) and Type of value is provided when an instance of Callable implementation is created.  
  
4) Another difference on run and call method is that run method can not [throw](http://java67.blogspot.sg/2012/10/difference-between-throw-vs-throws-in.html) checked exception while call method can throw checked exception in Java.