Threads

Table of Contents

[1. Java Concurrency 3](#_Toc518474707)

[1.1 Life Cycle of a Thread 3](#_Toc518474708)

[1.2 Major operations on the Threads – Inter Thread Communication 4](#_Toc518474709)

[1.3 Thread Local 5](#_Toc518474710)

[1.4 Executor Interface 6](#_Toc518474711)

[1.5 Executor Interface 6](#_Toc518474712)

[1.6 Scheduled Executor Service 9](#_Toc518474713)

[1.7 Fixed Thread Pool 10](#_Toc518474714)

[1.8 Cached Thread Pool 12](#_Toc518474715)

[1.9 Single Thread Executor 13](#_Toc518474716)

[1.10 New Scheduled Thread Pool 15](#_Toc518474717)

[1.11 Thread Pool Executor 16](#_Toc518474718)

[2. Executor Framework 23](#_Toc518474719)

[2.1 Thread Pool 23](#_Toc518474720)

[2.2 Different types of Thread pool 23](#_Toc518474721)

[2.2.1 Fixed Size pool 23](#_Toc518474722)

[2.2.2 Cached Pool 23](#_Toc518474723)

[2.2.3 Fork / Join pool 23](#_Toc518474724)

[3. Concurrency - Blocking Queue 24](#_Toc518474725)

[4. Concurrency – ConcurrentMap 28](#_Toc518474726)

[5. Reentrant Locks 31](#_Toc518474727)

[6. Similarities between Runnable and Callable interface 36](#_Toc518474728)

[7. Difference between Runnable and Callable interface 37](#_Toc518474729)

[8. Semaphore Variable in Java 37](#_Toc518474730)

[9. Object Class – Wait, Notify and notify all 37](#_Toc518474731)

[10. Race Condition 37](#_Toc518474732)

[11. Thread Dead Lock 38](#_Toc518474733)

[12. Semaphore 38](#_Toc518474734)

[13. Thread Sleep vs. Wait 38](#_Toc518474735)

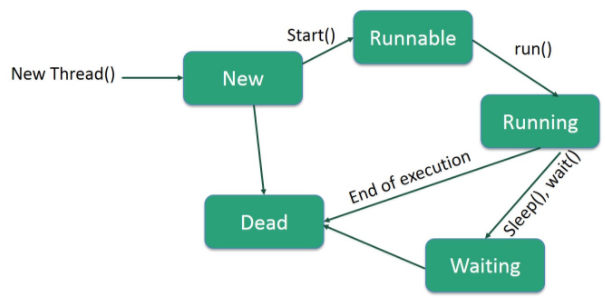
[14. CountDownLatch 39](#_Toc518474736)

[15. Cyclic Barrier 40](#_Toc518474737)

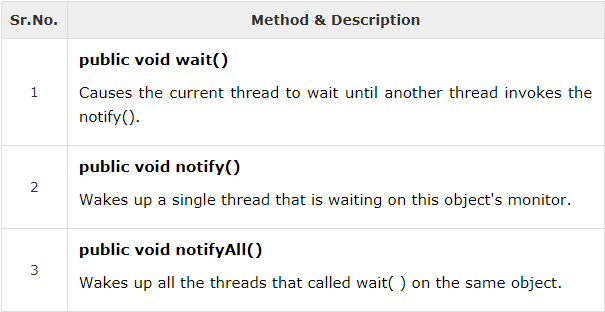
***Threads***

# Java Concurrency

## 1.1 Life Cycle of a Thread



## Major operations on the Threads – Inter Thread Communication



All the above mentioned methods have been declared as final in the Object class. All the three methods can be called only from within a **synchronized** context.

## Thread Local



***Example :***

**package** com.vmurthy.threadDemo;

**class** RunnableTest **implements** Runnable{

**int** counter;

ThreadLocal<Integer> threadCounter = **new** ThreadLocal<Integer>();

**public** **void** run()

{

**for**(**int** i = 0; i<=2; i++){

System.*out*.println(" Integer count: "+ counter);

System.*out*.println(" ThreadLocalCounter count: "+ threadCounter.get());

**if**(threadCounter.get() != **null**)

{

threadCounter.set(threadCounter.get() + 1);

}**else**{

threadCounter.set(0);

}

counter++;

}

}

}

**public** **class** ThreadLocalDemo {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

RunnableTest runnableObj = **new** RunnableTest();

Thread t1 = **new** Thread(runnableObj);

Thread t2 = **new** Thread(runnableObj);

Thread t3 = **new** Thread(runnableObj);

Thread t4 = **new** Thread(runnableObj);

t1.start();

t2.start();

t3.start();

t4.start();

**try**{

t1.join();

t2.join();

t3.join();

t4.join();

}

**catch**(Exception e)

{

System.*out*.println("Exception :"+e);

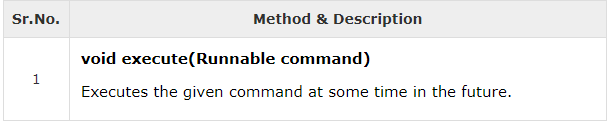
}

}

}

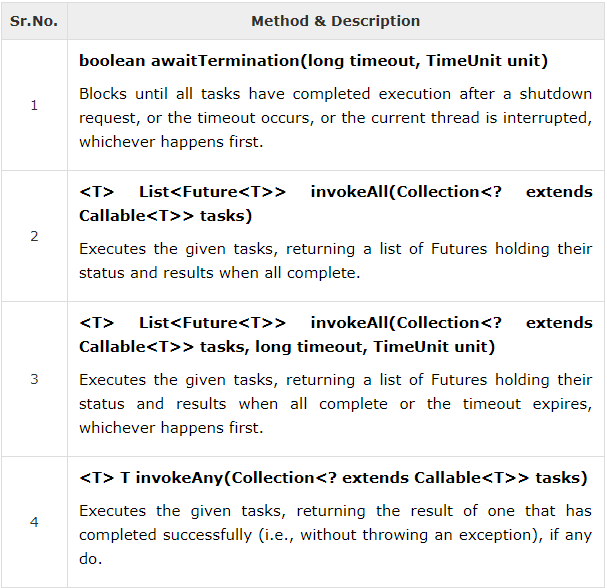
## Executor Interface

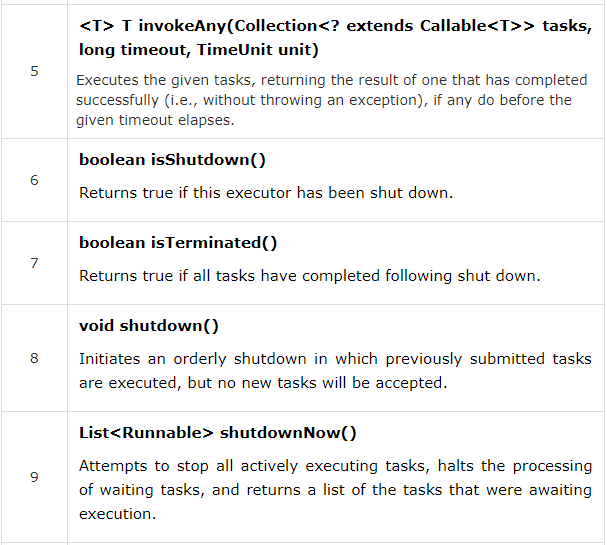
The executor interface supports the launching of new tasks.

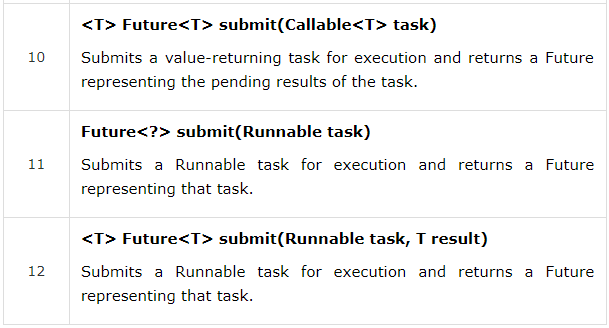


## Executor Interface

The *java.util.concurrent.ExecutorService* is a sub interface of Executor which adds more features to manage the life cycle of both the individual tasks and the Executor itself.

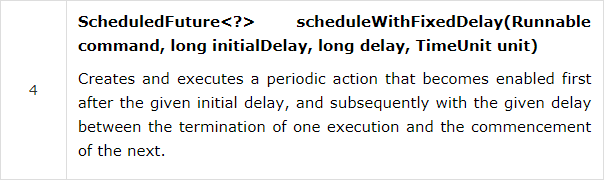


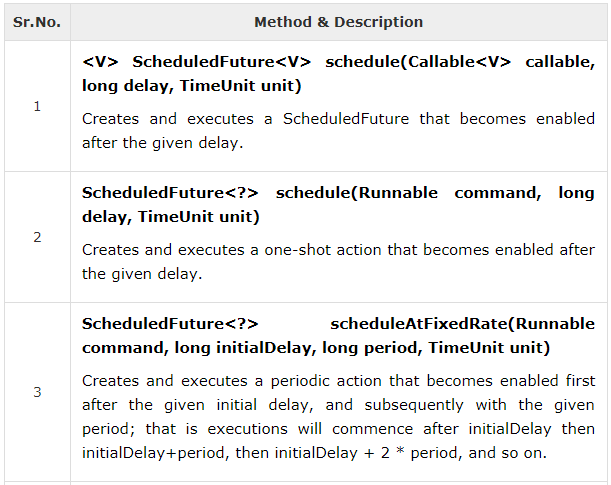




## Scheduled Executor Service

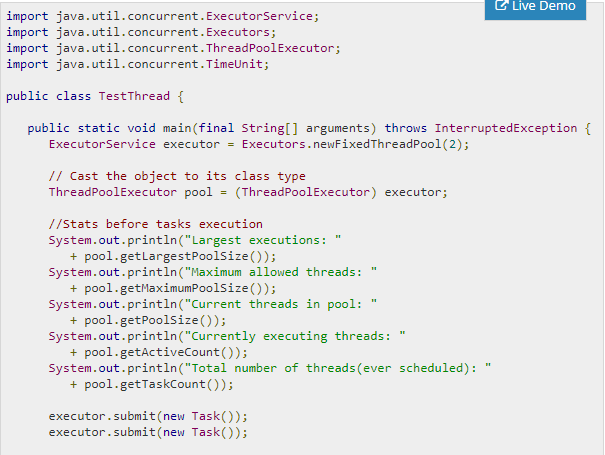
Java.util.concurrent.ScheduledExecutorService interface is a sub interface of ExecutorService interface which supports the periodic and/ or future execution of tasks.





## Fixed Thread Pool

This thread pool can be created by calling the static method newFixedThreadPool of ***Executors*** class. If any of the thread contained in the pool terminates, a new thread will be instantiated to take its place.

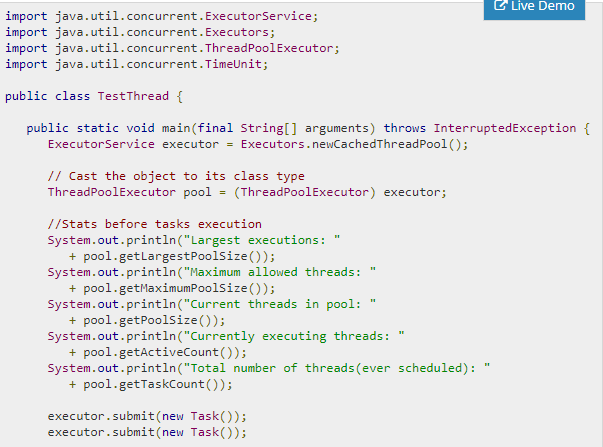




## Cached Thread Pool

A cached thread pool can be obtained by calling the ***newCachedThreadPool*** static method of the Executors class. ***newCachedThreadPool*** method creates an executor having a dynamic thread pool. Such an executor can be used in an application where we have requirement to create many short lived threads.

Example :



## Single Thread Executor

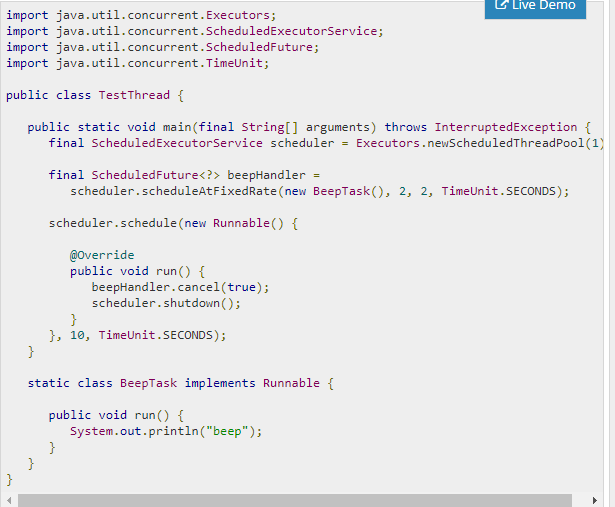
It Creates an Executor that uses a single worker thread operating off an unbounded queue, and uses the provided ThreadFactory to create a new thread when needed. Unlike the otherwise equivalent newFixedThreadPool(1, threadFactory) the returned executor is guaranteed not to be reconfigurable to use additional threads.



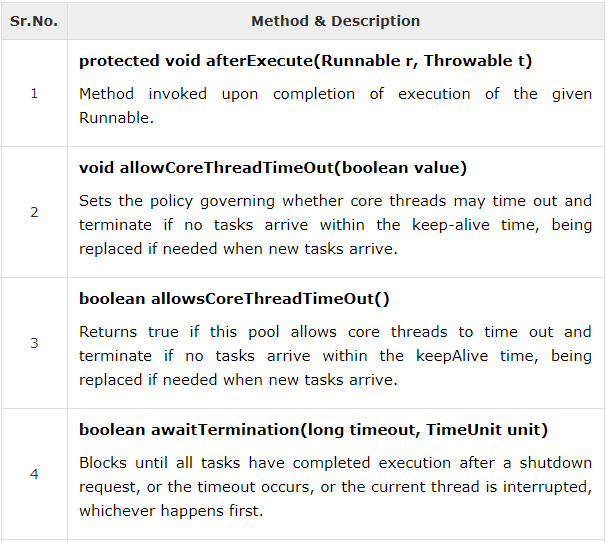


## New Scheduled Thread Pool

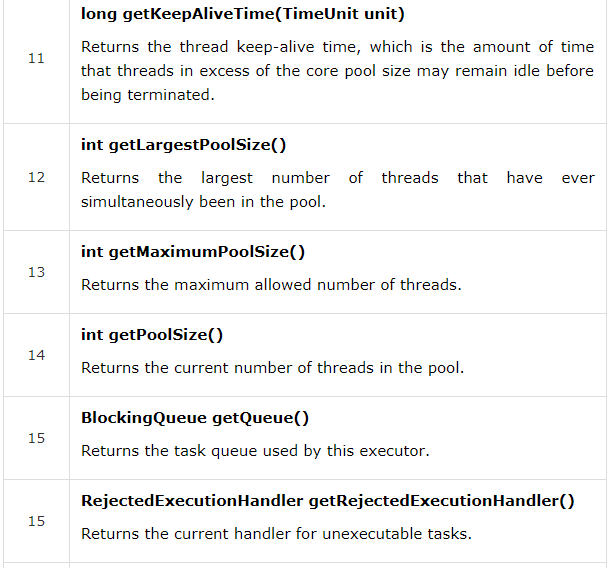
It creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

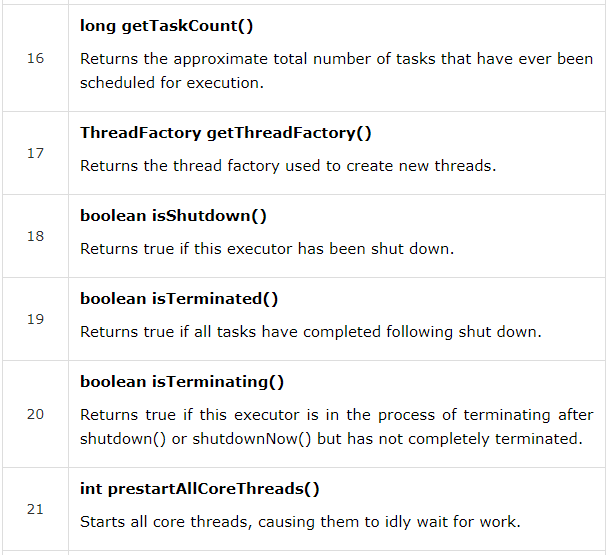


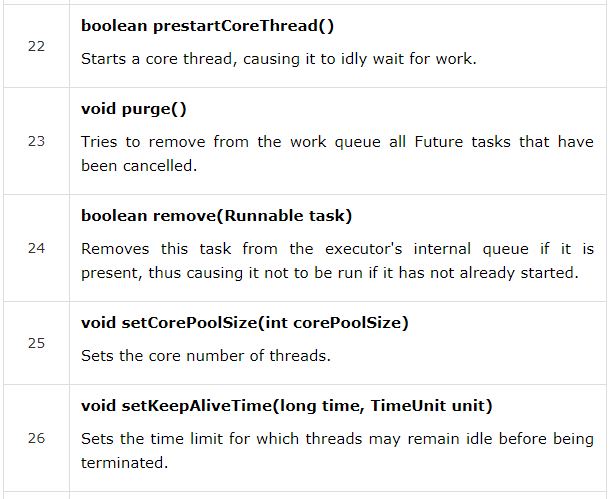
## Thread Pool Executor

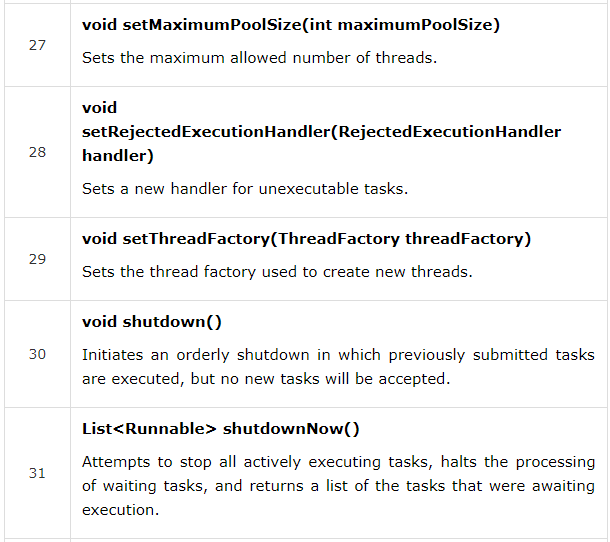
The java.util.concurrent.ThreadPoolExecutor is an ExecutorService to execute each of the submitted task using any one among the thread from the pool, normally configured using the factory methods of the Executors class. It also provides various utility method to check the statistics and monitor the tasks. 

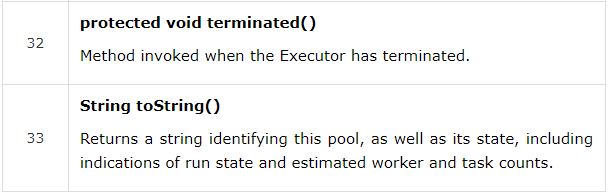












# Executor Framework

It provides classes and interface that helps us to decouple the process of creating and managing the threads for the rest of the application logic.

Access to the features defined in the Executor framework is mediated via ***Executor*** class.

Concurrent programs are more often to handle large number of requests. But spawning a thread for every client call will not be adequate and it would also increase processing overhead and also consumes excessive memory.

## 2.1 Thread Pool

It is basically a pool with a predefined number of threads.

Pool size is often defined by the number of CORES, /IO intensive or Compute intensive.

* *I/O intensive:* These are the type of threads which require frequent read or write operation onto a network drive, disks or some slower media.
* *Compute Intensive:* These threads mostly continue execution until it is interrupted by an external event or by itself.

## 2.2 Different types of Thread pool

Java Executor framework provides various ways to organize the thread pool.

### 2.2.1 Fixed Size pool

In this model we re use a fixed number of threads amortizing creation overhead. In the event if we receive a new client request and all the available threads are occupied, then the newly arrived thread will be placed into a work queue and the next available thread will be assigned to the request in the work queue.

### 2.2.2 Cached Pool

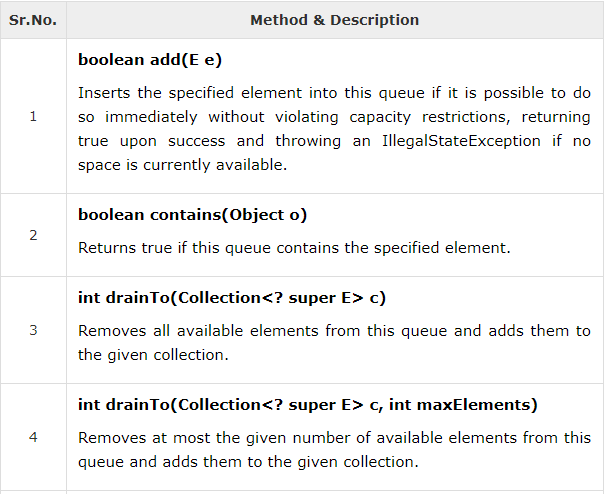
Here the thread gets created on demand and once the processing is completed if the thread is idle for more than 60 Seconds the same will be destroyed from the pool.

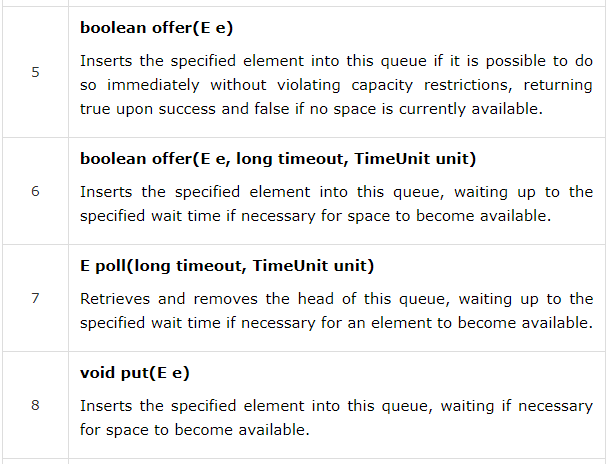
### 2.2.3 Fork / Join pool

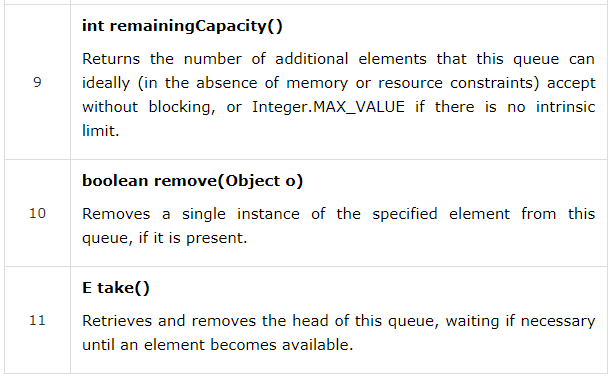
This model supports work stealing queues that maximize CPU utilization. Usually when the client request is received it is placed into a deque and thread starts processing the same once the thread burns through the complete deque, the thread starts stealing the request from the next deque from the bottom in order to avoid conflict with the thread processing the deque from the top.

# Concurrency - Blocking Queue

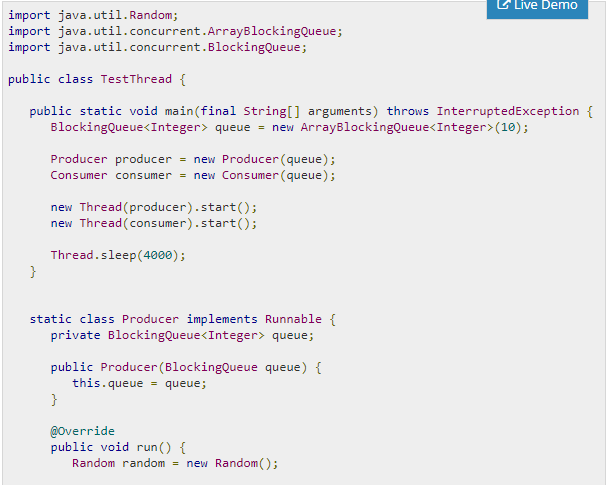
A java.util.concurrency.BlockingQueue is a sub interface of Queue interface, and additionally supports operations such as waiting for the queue to become non empty before retrieving an element ad also waiting for queue to become empty before storing an element into the queue.



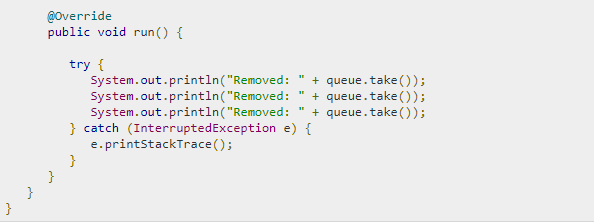




Example:







# Concurrency – ConcurrentMap

A java.util.concurrent.ConcurrentMap is a sub interface of Map interface, supports atomic operations on underlying map variable. It has get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. This interface ensures thread safety and atomicity guarantees.

Difference between HashMap and ConcurrentHashMap in Java

In this section, we will see some more details about HashMap and ConcurrentHashMap and compare them on various parameters like thread-safety, synchronization, performance, ease of use etc.

**1) As I said the earlier first significant difference between HashMap** and ConcurrentHashMap is that later is thread-safe and can be used in a concurrent environment without external synchronization. Though it doesn't provide the same level of synchronization as achieved by using Hashtable but it's enough for the most practical purpose.

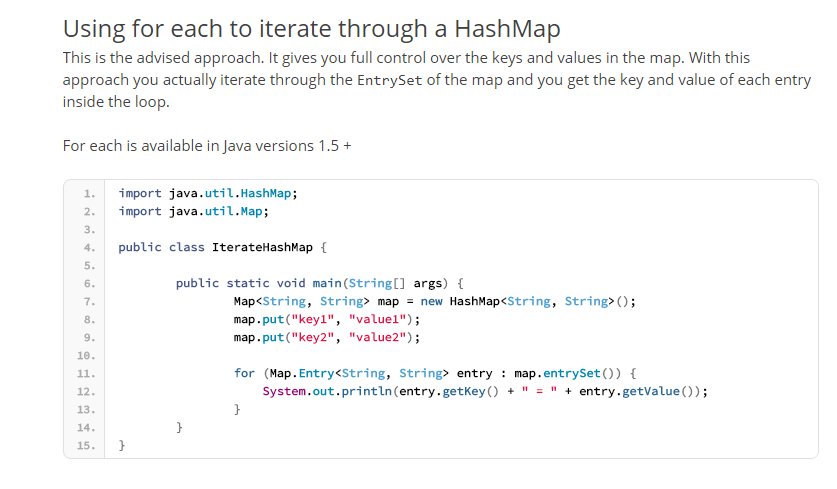
***2)You can make HashMap synchronized by wrapping*** it on Collections.synchornizedMap(HashMap) which will return a collection which is almost equivalent to Hashtable, where every modification operation on Map is locked on Map object while in case of ConcurrentHashMap, thread-safety is achieved by dividing whole Map into different partition based upon Concurrency level and only locking particular portion instead of locking the whole Map.

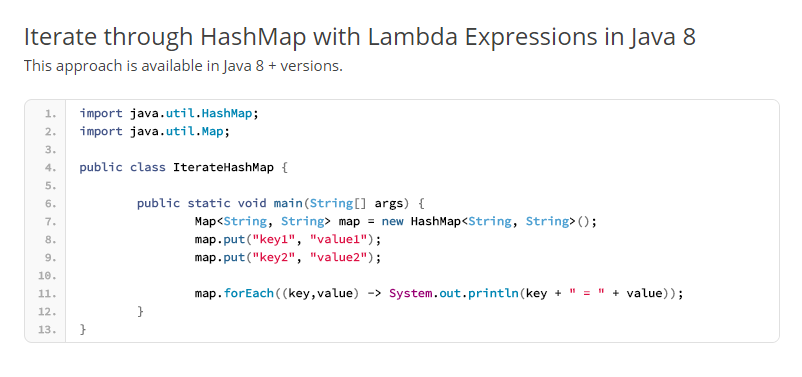
***Difference between ConcurrentHashMap and HashMap in Java Collection3)*** ConcurrentHashMap is more scalable and performs better than Synchronized HashMap in the multi-threaded environment while in Single threaded environment both HashMap and ConcurrentHashMap gives comparable performance, where HashMap only slightly better.

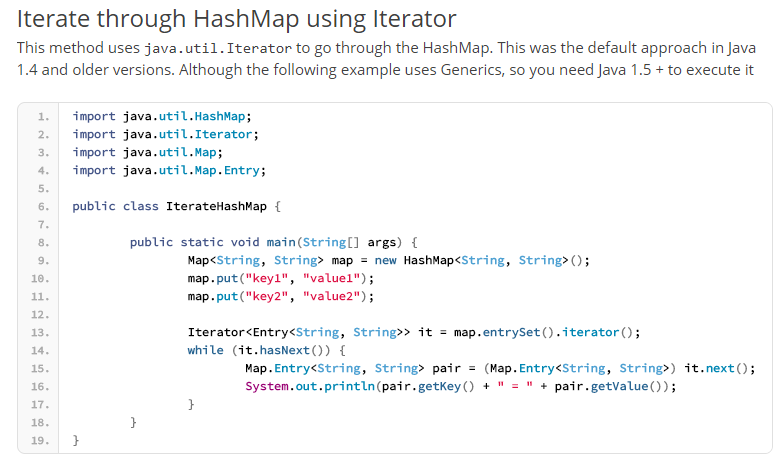
In Summary Main difference between ConcurrentHashMap and HashMap in Java Collection turns out to be thread-safety, Scalability, and Synchronization. ConcurrentHashMap is a better choice than synchronized HashMap if you are using them as cache, which is the most popular use case of a Map in Java application. ConcurrentHashMap is more scalable and outperforms when a number of reader threads outnumber the number of writer threads.

Note : HashMap is not Syncronized

Iterating through Hash Map :







# Reentrant Locks

The traditional way to achieve thread synchronization in Java is by the use of synchronized keyword. While it provides a certain basic synchronization, the synchronized keyword is quite rigid in its use. For example, a thread can take a lock only once. Synchronized blocks don’t offer any mechanism of a waiting queue and after the exit of one thread, any thread can take the lock. This could lead to starvation of resources for some other thread for a very long period of time.

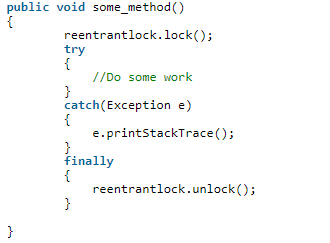
Reentrant Locks are provided in Java to provide synchronization with greater flexibility.

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

As the name says, ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

These locks are used in the following way:



The unlock statement is always called in the finally block to ensure that the lock is released even if an exception is thrown in the method body(try block).

ReentrantLock Methods

***lock():*** Call to the lock() method increments the hold count by 1 and gives the lock to the thread if the shared resource is initially free.

***unlock():*** Call to the unlock() method decrements the hold count by 1. When this count reaches zero, the resource is released.

***tryLock():*** If the resource is not held by any other thread, then call to tryLock() returns true and the hold count is incremented by one. If the resource is not free then the method returns false and the thread is not blocked but it exits.

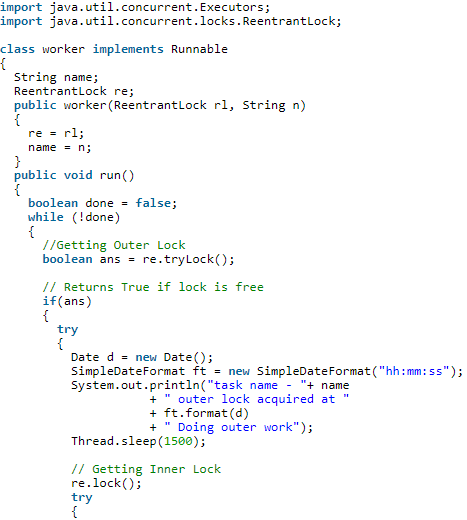
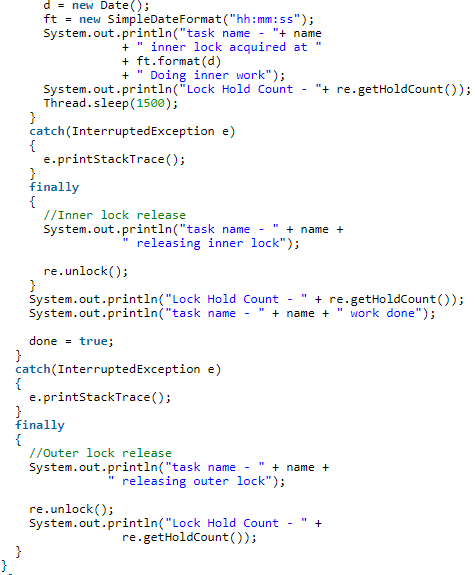
***tryLock(long timeout, TimeUnit unit):*** As per the method, the thread waits for a certain time period as defined by arguments of the method to acquire the lock on the resource before exiting.

***lockInterruptibly():*** This method acquires the lock if the resource is free while allowing for the thread to be interrupted by some other thread while acquiring the resource. It means that if the current thread is waiting for lock but some other thread requests the lock, then the current thread will be interrupted and return immediately without acquiring lock.

***getHoldCount():*** This method returns the count of the number of locks held on the resource.

***isHeldByCurrentThread():*** This method returns true if the lock on the resource is held by the current thread.

Example :



# Similarities between Runnable and Callable interface

1. The most common thing among them is that both are used to encapsulate code that needs to be run in parallel two different threads.
2. The second similarity is that both can be executed using Executors Framework introduced in Java 5. Executor framework defines interface ExecutorService which can accept and execute callable and runnable.
3. The runnable thread can be converted into Callable thread using the utility class of ExecutorService. But Callable instance cannot be executed in the same way as that of runnable interface using thread. The callable instance can be executed only through ExecutorService instance (submit method).

Example: Callable callable = Executors.callable(Runnable task);

1. The last similarity between them is that they are SAM type. That is they areSingle Abstract Method meaning they contain only one abstract method hence they can be executed using lambda expression.

New Thread(() -> System.out.println(“Executing Run Method Through Lambda Expression)).start()

# Difference between Runnable and Callable interface

1. The first key difference is that the run method of the Runnable interface is “void” it will not be able to return anything. Whereas the Callable thread can return an object representing the outcome of the execution. The Object is termed as Future object, it represents the lifecycle of the task and provides methods to check if the task was completed successfully or was cancelled.
2. The call VS run method – in order use the Callable interface the call method needs to be implemented. While in the case of creating a Runnable method the “run” method need to be implemented.
3. The Runnable thread can be executed through Thread class and can also be executed through the Executor service by converting the Runnable thread into a Callable Thread. Whereas the Callable thread can only be executed through the Executor Services.

# Semaphore Variable in Java

A Semaphore controls access to a shared resource through the use of a counter. That is if the counter value is greater than zero the access is granted if the value is lesser than zero the access is denied.

Counting semaphore and bounded semaphore are different types of semaphores.

# Object Class – Wait, Notify and notify all

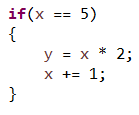
All the above mentioned methods functionality is based on the locks that are associated to objects and not on the threads hence they are contained in the object class rather than in thread class.

# Race Condition

It occurs when two or more thread try to access the shared data and change it at the same time. As the thread scheduling algorithm can swap between the thread in any order. It is therefore difficult to predict the order in which the threads are executed. Hence the value being changed depends upon the underlying thread scheduling algorithm. i.e. Both threads are racing to access the shared data.

Problem often occurs when one thread does a “*check-and-act*” (if the value of ix is 5 then do something) and another thread does something in between the check and act hence the result would not be as expected.

*Example code snippet*:



In the above mentioned code segment say two threads are executing, the first thread evaluates the if condition and enters the if block at the same time if the second thread modifies the value of the variable “x” the outcome which is expected to be 10 will not occur.

In order to overcome racing condition the shared segment of code is expected to be surrounded by a locking mechanism which ensures that at any given point in time only one thread operates on the shared resource.

# Thread Dead Lock

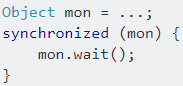
Producer Consumer problem is a variant of thread dead lock scenario. Say the consumer thread gains access to the “Stock” object and since the number of stocks is zero it does not complete itself or release the lock which in turn prevents the producer thread to gain access to the shared resource to do its job and hence results in a dead lock scenario.

# Semaphore

It is just a variable which is used to solve critical section problem. There are two types of Semaphore, First one is the counting semaphore which can take a non-negative integer value and the second one is the binary semaphore which can have only 0 or 1’s.

# Thread Sleep vs. Wait

A [wait](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait%28%29) can be "woken up" by another thread calling [notify](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notify%28%29) on the monitor which is being waited on whereas a [sleep](http://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#sleep%28long%29) cannot. Also a wait (and notify) must happen in a block synchronized on the monitor object whereas sleep does not:

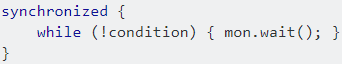


At this point the currently executing thread waits *and releases the monitor*. Another thread may do.



(On the same mon object) and the first thread (assuming it is the only thread waiting on the monitor) will wake up. You can also call [notify All](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notifyAll%28%29) if more than one thread is waiting on the monitor - this will wake *all of them up*. However, only one of the threads will be able to grab the monitor (remember that the wait is in a synchronized block) and carry on - the others will then be blocked until they can acquire the monitor's lock.

Another point is that you call wait on [Object](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) itself (i.e. you wait on an object's monitor) whereas you call sleep on [Thread](http://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html). Yet another point is that you can get *spurious wakeups* from wait (i.e. the thread which is waiting resumes for no apparent reason). You should **always**wait**whilst spinning on some condition** as follows:

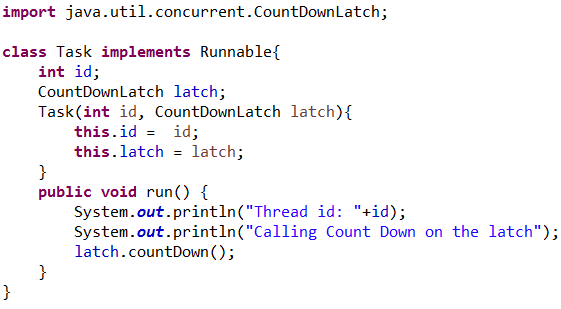


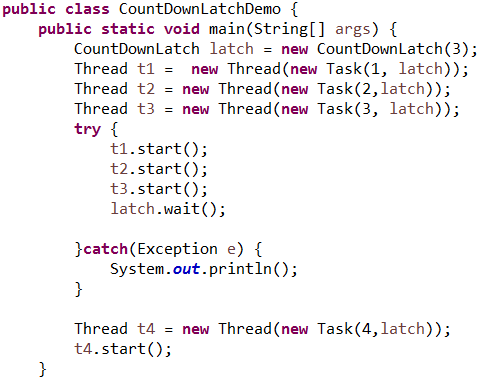
# CountDownLatch

It is used to make sure that one task waits for another sets of threads to complete before starting its execution.

We create a countdown latch by specifying an integer value representing the number of sub threads which are to be competed for the main thread to start its execution. When every sub tasks complete its execution it would request a count down on the latch by calling the “*latch.countDown()*” method to decrement the counter and when the counter value becomes zero the dependent thread starts its execution.

*Program*: CountDownLatchDemo.java





# Cyclic Barrier