[**How to use ConcurrentHashMap in Java - Example Tutorial and Working**](http://javarevisited.blogspot.in/2013/02/concurrenthashmap-in-java-example-tutorial-working.html)

ConcurrentHashMap in Java is introduced as an alternative of Hashtable in Java 1.5 as part of Java concurrency package. Prior to Java 1.5 if you need a Map implementation, which can be safely used in a concurrent and multi-threaded Java program, then, you only have [Hashtable](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html) or [synchronized Map](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html) because HashMap is not [thread-safe](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html). WithConcurrentHashMap, now you have a better choice; because not only it can be safely used in the concurrent multi-threaded environment but also provides better performance over Hashtable and synchronizedMap. ConcurrentHashMap performs better than earlier two because it only locks a portion of Map, instead of whole Map, which is the case with [Hashtable and synchronized Map](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html). CHM allows concurred read operations and the same time maintains integrity by synchronizing write operations. We have seen basics of ConcurrentHashMap on [Top 5 Java Concurrent Collections from JDK 5 and 6](http://javarevisited.blogspot.sg/2013/02/concurrent-collections-from-jdk-56-java-example-tutorial.html) and in this Java tutorial, we will learn:

       How ConcurrentHashMap works in Java or how it is implemented in Java.

       When to use ConcurrentHashMap in Java

       ConcurrentHashMap examples in Java

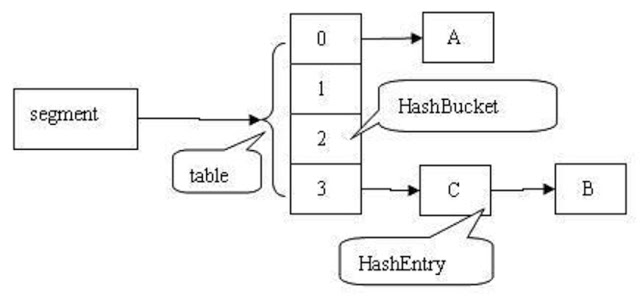
       And some important properties of CHM.

How ConcurrentHashMap is implemented in Java

ConcurrentHashMap is introduced as an alternative of Hashtable and provided all functions supported by Hashtable with an additional feature called "concurrency level", which allows ConcurrentHashMap to partition Map. ConcurrentHashMap allows multiple readers to read concurrently without any [blocking](http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html). This is achieved by partitioning Map into different parts based on concurrency level and locking only a portion of Map during updates. Default concurrency level is 16, and accordingly Map is divided into 16 part and each part is governed with a different lock. This means, 16 thread can operate on Map simultaneously until they are operating on different part of Map. This makes ConcurrentHashMap high performance despite keeping thread-safety intact.  Though, it comes with a caveat. Since update operations like put(), remove(), putAll() or clear() is not [synchronized](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), **concurrent retrieval may not reflect most recent change on Map**.

In case of putAll() or clear(), which operates on whole Map, concurrent read may reflect insertion and removal of only some entries. Another important point to remember is iteration over CHM, [Iterator](http://javarevisited.blogspot.com/2011/10/java-iterator-tutorial-example-list.html) returned by keySet of ConcurrentHashMap are weekly consistent and they only reflect state of ConcurrentHashMap and certain point and may not reflect any recent change. Iterator of ConcurrentHashMap's keySet area also [fail-safe](http://javarevisited.blogspot.in/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and doesn’t throw ConcurrentModificationExceptoin..

Default concurrency level is 16 and can be changed, by providing a number which make sense and work for you while creating ConcurrentHashMap. Since concurrency level is used for internal sizing and indicate number of concurrent update without contention, so, if you just have few writers or thread to update Map keeping it low is much better. ConcurrentHashMap also uses ReentrantLock to internally lock its segments.

[](https://3.bp.blogspot.com/-YDubLkYFi4c/VurLyW0NUoI/AAAAAAAAFJE/caY7JsRvmjUgV0cDMi0N0FeGRND6WGowQ/s1600/Internal%2BImplementation%2Bof%2BConcurrentHashMap%2Bin%2BJava.jpg)

**ConcurrentHashMap putifAbsent example in Java**

ConcurrentHashMap examples are similar to [Hashtable examples](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html), we have seen earlier,  but worth knowing is the use of putIfAbsent() method. Many times we need to insert entry into Map if it's not present already, and we wrote following kind of code:

synchronized(map){

**if** (map**.**get(key) **==** *null*){

**return** map**.**put(key, value);

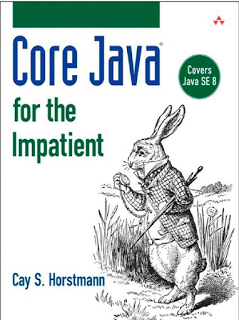
  } **else**{

**return** map**.**get(key);

  }

}

Though this code will work fine in [HashMap and Hashtable](http://java67.blogspot.sg/2012/08/5-difference-between-hashtable-hashmap-Java-collection.html), This won't work in ConcurrentHashMap; because, during put operation whole map is not locked, and while one thread is putting value, other thread's get() call can still return null which result in one thread overriding value inserted by other thread. Ofcourse, you can wrap whole code in [synchronized block](http://java67.blogspot.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html) and make it [thread-safe](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) but that will only make your code single threaded. ConcurrentHashMap provides putIfAbsent(key, value) which does same thing but atomically and thus eliminates above race condition.   
  
  
See [Core Java for Inpatients](http://aax-us-east.amazon-adsystem.com/x/c/Qt7w9x6vRwA8RnffbPFct6MAAAFfvSg7lwEAAAFKAdJsvvM/https:/assoc-redirect.amazon.com/g/r/http:/www.amazon.com/Core-Java-Impatient-Cay-Horstmann/dp/0321996321/ref=as_at?creativeASIN=0321996321&linkCode=w61&imprToken=kIDwkapXhAk0d6q4mYfezw&slotNum=0&tag=javamysqlanta-20) for more details about how to use this method effectively:

[](http://aax-us-east.amazon-adsystem.com/x/c/Qt7w9x6vRwA8RnffbPFct6MAAAFfvSg7lwEAAAFKAdJsvvM/https:/assoc-redirect.amazon.com/g/r/http:/www.amazon.com/Core-Java-Impatient-Cay-Horstmann/dp/0321996321/ref=as_at?creativeASIN=0321996321&linkCode=w61&imprToken=kIDwkapXhAk0d6q4mYfezw&slotNum=1&tag=javamysqlanta-20)

When to use ConcurrentHashMap in Java

[Java ConcurrentHashMap Example Tutorial and internal working](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)

ConcurrentHashMap is best suited when you have multiple readers and few writers. If writers outnumber reader, or writer is equal to reader, than performance of ConcurrentHashMap effectively reduces to [synchronized map](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html) or [Hashtable](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html). Performance of CHM drops, because you got to lock all portion of Map, and effectively each reader will wait for another writer, operating on that portion of Map. ConcurrentHashMap is a good choice for caches, which can be initialized during application start up and later accessed my many request processing threads. As javadoc states, CHM is also a [good replacement of Hashtable](http://javarevisited.blogspot.sg/2013/02/concurrent-collections-from-jdk-56-java-example-tutorial.html) and should be used whenever possible, keeping in mind, that CHM provides slightly weeker form of synchronization than Hashtable.

Summary

Now we know What is ConcurrentHashMap in Java and when to use ConcurrentHashMap, it’s time to know and revise some important points about CHM in Java.

1. ConcurrentHashMap allows concurrent read and thread-safe update operation.

2. During the update operation, ConcurrentHashMap only locks a portion of Map instead of whole Map.

3. The concurrent update is achieved by internally dividing Map into the small portion which is defined by concurrency level.

4. Choose concurrency level carefully as a significantly higher number can be a waste of time and space and the lower number may introduce thread contention in case writers over number concurrency level.

5. All operations of ConcurrentHashMap are [thread-safe](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html).

6. Since ConcurrentHashMap implementation doesn't lock whole Map, there is chance of read overlapping with update operations like put() and remove(). In that case result returned by get() method will reflect most recently completed operation from there start.

7. Iterator returned by ConcurrentHashMap is weekly consistent, [fail-safe](http://javarevisited.blogspot.com/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and never throw ConcurrentModificationException. In Java.

8. ConcurrentHashMap doesn't allow null as key or value.

9. You can use ConcurrentHashMap in place of [Hashtable](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html) but with caution as CHM doesn't lock whole Map.

10. During putAll() and clear() operations, the concurrent read may only reflect insertion or deletion of some entries.

That’s all on **What is ConcurrentHashMap in Java** and when to use it. We have also seen little bit about internal working of ConcurrentHashMap and how it achieves it’s thread-safety and better performance over Hashtable and synchronized Map. Use ConcurrentHashMap in Java program, when there will be more reader than writers and it’s a good choice for creating cache in Java as well.

# How ConcurrentHashMap Works Internally in Java

[**JOIN FOR FREE**](https://dzone.com/articles/how-concurrenthashmap-works-internally-in-java)

**ConcurrentHashMap:** It allows concurrent access to the map. Part of the map called *Segment (internal data structure)*is only getting locked while adding or updating the map. So ConcurrentHashMap allows concurrent threads to read the value without locking at all. This data structure was introduced to improve performance.

**Concurrency-Level:**Defines the number which is an estimated number of concurrently updating threads. The implementation performs internal sizing to try to accommodate this     many threads.

**Load-Factor:** It's a threshold, used to control resizing.

**Initial Capacity:** The implementation performs internal sizing to accommodate these many elements.

A ConcurrentHashMap is divided into number of segments, and the example which I am explaining here used default as 32 on initialization.

A ConcurrentHashMap has internal final class called Segment so we can say that ConcurrentHashMap is internally divided in segments of size 32, so at max 32 threads can work at a time. It means each thread can work on a each segment during high concurrency and atmost 32 threads can operate at max which simply maintains 32 locks to guard each bucket of the ConcurrentHashMap.

The definition of Segment is as below:

/\*\* Inner Segment class plays a significant role \*\*/

protected static final class Segment {

protected int count;

protected synchronized int getCount() {

return this.count;

}

protected synchronized void synch() {}

}

/\*\* Segment Array declaration \*\*/

public final Segment[] segments = new Segment[32];

As we all know that Map is a kind of data structure which stores data in key-value pair which is array of inner class Entry, see as below:

static class Entry implements Map.Entry {

protected final Object key;

protected volatile Object value;

protected final int hash;

protected final Entry next;

Entry(int hash, Object key, Object value, Entry next) {

this.value = value;

this.hash = hash;

this.key = key;

this.next = next;

}

// Code goes here like getter/setter

}

And ConcurrentHashMap class has an array defined as below of type Entry class:

 protected transient Entry[] table;

This Entry array is getting initialized when we are creating an instance of ConcurrentHashMap, even using a default constructor called internally as below:

public ConcurrentHashMap(int initialCapacity, float loadFactor) {

//Some code

int cap = getCapacity();

this.table = newTable(cap); // here this.table is Entry[] table

}

protected Entry[] newTable(int capacity) {

this.threshold = ((int)(capacity \* this.loadFactor / 32.0F) + 1);

return new Entry[capacity];

}

Here, threshold is getting initialized for re-sizing purpose.

**Inserting (Put) Element in ConcurrentHashMap:**

Most important thing to understand the put method of ConcurrentHashMap, that how ConcurrentHashMap works when we are adding the element. As we know put method takes two arguments both of type Object as below:

put(Object key, Object value)

So it wil 1st calculate the hash of key as below:

int hashVal = hash(key);

static int hash(Object x) {

int h = x.hashCode();

return (h << 7) - h + (h >>> 9) + (h >>> 17);

}

After getting the hashVal we can decide the Segment as below:

Segment seg = segments[(hash & 0x1F)];     // segments is an array defined above   
      
Since it's all about concurrency, we need synchronized block on the above Segment as below:

synchronized (seg) {

// code to add

int index = hash & table.length - 1; // hash we have calculated for key and table is Entry[] table

Entry first = table[index];

for (Entry e = first; e != null; e = e.next) {

if ((e.hash == hash) && (eq(key, e.key))) { // if key already exist means updating the value

Object oldValue = e.value;

e.value = value;

return oldValue;

}

}

Entry newEntry = new Entry(hash, key, value, first); // new entry, i.e. this key not exist in map

table[index] = newEntry; // Putting the Entry object at calculated Index

}

**Size of ConcurrentHashMap**

Now when we are asking for size() of the ConcurrentHashMap the size comes out as below:

for (int i = 0; i < this.segments.length; i++) {

c += this.segments[i].getCount(); //here c is an integer initialized with zero

}

**Getting (get) Element From ConcurrentHashMap**

When we are getting an element from ConcurrentHashMap we are simply passing key and hash of key is getting calculated. The defintion goes something like as below:

public Object get(Object key){

//some code here

int index = hash & table.length - 1; //hash we have calculated for key and calculating index with help of hash

Entry first = table[index]; //table is Entry[] table

for (Entry e = first; e != null; e = e.next) {

if ((e.hash == hash) && (eq(key, e.key))) {

Object value = e.value;

if (value == null) {

break;

}

return value;

}

}

//some code here

}

**Note:** No need to put any lock when getting the element from ConcurrentHashMap.

**Removing Element From ConcurrentHashMap**

Now question is how remove works with ConcurrentHashMap, so let us understand it. Remove basically takes one argument 'Key' as an argument or takes two argument 'Key' and 'Value' as below:

Object remove(Object key);

boolean remove(Object key, Object value);

Now let us understand how this works internally. The method remove (Object key) internally calls remove (Object key, Object value) where it passed 'null' as a value. Since we are going to remove an element from a Segment, we need a lock on the that Segment.

Object remove(Object key, Object value) {

Segment seg = segments[(hash & 0x1F)]; //hash we have calculated for key

synchronized (seg) {

Entry[] tab = this.table; //table is Entry[] table

int index = hash & tab.length - 1; //calculating index with help of hash

Entry first = tab[index]; //Getting the Entry Object

Entry e = first;

while(true) {

if ((e.hash == hash) && (eq(key, e.key))) {

break;

}

e = e.next;

}

Object oldValue = e.value;

Entry head = e.next;

for (Entry p = first; p != e; p = p.next) {

head = new Entry(p.hash, p.key, p.value, head);

}

table[index] = head;

seg.count -= 1;

}

return oldValue;

}

# Example of ConcurrentHashMap in Java

By Arvind Rai, April 21, 2013 | Modified on July 30, 2016

On this page we will provide example of ConcurrentHashMap in java. ConcurrentHashMap is thread safe but does not use locking on complete map. It is fast and has better performance in comparison to Hashtable in concurrent environment. Find some methods of ConcurrentHashMap.   
  
**get()**: Pass the key as an argument and it will return associated value.   
**put()**: Pass key and value and it will map.   
**putIfAbsent()**: Pass key and value and it will map only if key is not already present.   
**remove()**: Removes the entry for the given key.

##### Contents

* [Java ConcurrentHashMap Internal Working](https://www.concretepage.com/java/example_concurrenthashmap_java#internal)
* [ConcurrentHashMap Example](https://www.concretepage.com/java/example_concurrenthashmap_java#example)
* [ConcurrentHashMap vs Hashtable](https://www.concretepage.com/java/example_concurrenthashmap_java#hashtable)
* [ConcurrentHashMap vs HashMap](https://www.concretepage.com/java/example_concurrenthashmap_java#hashmap)

### Java ConcurrentHashMap Internal Working

1. **Concurrency for retrieval**: Retrieval of elements from ConcurrentHashMap does not use locking. It may overlap with update operation. We get the elements of last successfully completed update operation. In case of aggregate operations such as putAll and clear(), concurrent retrieval may show insertion or removal of only some elements.  
  
2. **Iteration of ConcurrentHashMap**: Iterators and Enumerations also return the elements which have been concurrently added while iterating. ConcurrentHashMap does not throw ConcurrentModificationException.   
  
3. **Concurrency for updates**: Concurrent updates are thread safe. ConcurrentHashMap constructor has an optional concurrency level argument. The default value is 16. This is the estimated number of concurrently updating threads. It is used in internal sizing to accommodate concurrently updating threads. Hash table is internally partitioned into the concurrency level number so that it can avoid updating concurrent thread contention.

### ConcurrentHashMap Example

Find the example.   
**ConcurrentHashMapDemo.java**

package com.concretepage;

import java.util.Iterator;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ConcurrentHashMapDemo {

private final ConcurrentHashMap<Integer,String> conHashMap = new ConcurrentHashMap<Integer,String>();

public static void main(String[] args) {

ExecutorService service = Executors.newFixedThreadPool(3);

ConcurrentHashMapDemo ob = new ConcurrentHashMapDemo();

service.execute(ob.new WriteThreasOne());

service.execute(ob.new WriteThreasTwo());

service.execute(ob.new ReadThread());

service.shutdownNow();

}

class WriteThreasOne implements Runnable {

@Override

public void run() {

for(int i= 1; i<=10; i++) {

conHashMap.putIfAbsent(i, "A"+ i);

}

}

}

class WriteThreasTwo implements Runnable {

@Override

public void run() {

for(int i= 1; i<=5; i++) {

conHashMap.put(i, "B"+ i);

}

}

}

class ReadThread implements Runnable {

@Override

public void run() {

Iterator<Integer> ite = conHashMap.keySet().iterator();

while(ite.hasNext()){

Integer key = ite.next();

System.out.println(key+" : " + conHashMap.get(key));

}

}

}

}

**Output**

1 : B1

2 : B2

3 : B3

4 : B4

5 : B5

6 : A6

7 : A7

8 : A8

9 : A9

10 : A10

### ConcurrentHashMap vs Hashtable

1. ConcurrentHashMap is based on hash table. It allows to put null object but Hashtable allows only not-null object.   
  
2. The object which is being used as key must override hashCode() and equals() methods.   
3. The methods such as get(), put(), remove() are synchronized in Hashtable whereas ConcurrentHashMap does not use synchronized methods for concurrency.   
  
4. In concurrent modification, the iteration of Hashtable elements will throw ConcurrentModificationExceptionwhereas ConcurrentHashMap does not.

### ConcurrentHashMap vs HashMap

1. ConcurrentHashMap and HashMap both are based on hash table.   
     
   2. ConcurrentHashMap supports full concurrency of retrieval. HashMap can be synchronized using Collections.synchronizedMap() .   
     
   3. ConcurrentHashMap provides concurrency level for updates that can be changed while instantiating.   
     
   4. In concurrent modification HashMap throws ConcurrentModificationException whereas ConcurrentHashMap does not.

# ConcurrentHashMap: usage and functionality

On the previous page, we saw how the [ConcurrentHashMap](https://www.javamex.com/tutorials/synchronization_concurrency_8_hashmap.shtml) offers a means of improving concurrency beyond that of normal hash maps. In many cases, ConcurrentHashMap can be used as a drop-in replacement for a synchronized HashMap, and offers a means of avoiding synchronization in the traditional sense. (A couple of subtle differences are that ConcurrentHashMap will generally take up more memory, and that it cannot take null as a key.) Let's consider a web server that counts the number of instances of particular queries. We'll hold a map of query strings to integers and define an incrementCount() method which we can call at the moment of serving a particular query:

public final class MyServlet extends MyAbstractServlet {

private final Map<String,Integer> queryCounts =

Collections.synchronizedMap(new HashMap<String,Integer>(1000));

private void incrementCount(String q) {

Integer cnt = queryCounts.get(q);

if (cnt == null) {

queryCounts.put(q, 1);

} else {

queryCounts.put(q, cnt + 1);

}

}

}

In this example, we're using a plain old HashMap wrapped up in a synchronization wrapper. Recall that wrapping the map with Collections.synchronizedMap(...) makes it safe to access the map concurrently: each call to get(), put(), size(), containsKey() etc will synchronize on the map during the call. (One problem that we'll see in a minute is that iterating over the map does still require explicit synchronization.)

Note that this **doesn't make incrementCount() atomic**, but it does make it **safe**. That is, concurrent calls to incrementCount() will **never leave the map in a corrupted state**. But they might 'miss a count' from time to time. For example, two threads could concurrently read a current value of, say, 2 for a particular query, both independently increment it to 3, and both set it to 3, when in fact two queries have been made. Generally in the context of counting queries, we'd probably live with this: it's quite unlikely that two clients are making the selfsame query at exactly the same time, and even if they were, we wouldn't really care about missing the odd count here and there in order to improve performance.

In this example, we can improve concurrency in a single line by replacing our synchronized hash map with a ConcurrentHashMap:

private final Map<String,Integer> queryCounts =

**new ConcurrentHashMap<String,Integer>(1000)**;

Note that our incrementCount() will still have the same semantics: that is, it will never leave the map in an inconsistent state, but it could still miss a count in an unlucky case.

### Truly atomic updates

So what if we want truly atomic updates: that is, to make incrementCount() never miss a count? To do this with a traditional HashMap, we could synchronize on the map during the entire incrementCount()method, with a potential impact on throughput. With ConcurrentHashMap, we can take advantage of its concurrent update facility. ConcurrentHashMap implements the following interface:

public interface **ConcurrentMap**<K, V> extends Map<K, V> {

V putIfAbsent(K key, V value);

boolean remove(Object key, Object value);

boolean replace(K key, V oldValue, V newValue);

V replace(K key, V value);

}

In our case, the interesting methods are the replace() methods, which are effectively compare-and-set operations for a map. So we can implement our incrementCount() method as follows. Note that we do now need to change the signature of our queryCounts map and declare it as a ConcurrentMap:

public final class MyServlet extends MyAbstractServlet {

private final ConcurrentMap<String,Integer> queryCounts =

new ConcurrentHashMap<String,Integer>(1000);

private void incrementCount(String q) {

Integer oldVal, newVal;

do {

oldVal = queryCounts.get(q);

newVal = (oldVal == null) ? 1 : (oldVal + 1);

} while (!queryCounts.replace(q, oldVal, newVal));

}

}

This code is very similar to the code to update an AtomicInteger: we read the current value of the count, calculate the new count, and then say to the ConcurrentHashMap: "please map this key to this new value, if and only if the previously mapped value was this". If the call returns false to say that we were wrong about the previously mapped value, indicating in effect that another thread has "snuck in", then we simply loop round and try again. As with AtomicInteger updates, this is very efficient because we rarely expect another thread to sneak in, and where it does, we can keep hold of the CPU rather than having to sleep while the other thread releases the lock.

### Iterating over the map

In our case of counting web queries so far, you may be wondering "what's the big deal"? Of course, there is the argument that on a busy server, anything that helps improve throughput is a big deal. But in this case, most of the operations on the map are very quick and occur only once per query, so the map won't be highly contended. In this case, a bigger benefit comes when we want to [iterate over the map](https://www.javamex.com/tutorials/synchronization_concurrency_8_hashmap3_iteration.shtml).

# Java Thread Pool

**Java Thread pool** represents a group of worker threads that are waiting for the job and reuse many times.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

#### Advantage of Java Thread Pool

**Better performance** It saves time because there is no need to create new thread.

#### Real time usage

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

#### Example of Java Thread Pool

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

*File: WorkerThread.java*

1. **import** java.util.concurrent.ExecutorService;
2. **import** java.util.concurrent.Executors;
3. **class** WorkerThread **implements** Runnable {
4. **private** String message;
5. **public** WorkerThread(String s){
6. **this**.message=s;
7. }
8. **public** **void** run() {
9. System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);
10. processmessage();//call processmessage method that sleeps the thread for 2 seconds
11. System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name
12. }
13. **private** **void** processmessage() {
14. **try** {  Thread.sleep(2000);  } **catch** (InterruptedException e) { e.printStackTrace(); }
15. }
16. }

*File: JavaThreadPoolExample.java*

1. **public** **class** TestThreadPool {
2. **public** **static** **void** main(String[] args) {
3. ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads
4. **for** (**int** i = 0; i < 10; i++) {
5. Runnable worker = **new** WorkerThread("" + i);
6. executor.execute(worker);//calling execute method of ExecutorService
7. }
8. executor.shutdown();
9. **while** (!executor.isTerminated()) {   }
11. System.out.println("Finished all threads");
12. }
13. }

[download this example](https://www.javatpoint.com/src/multi/threadpool.zip)

Output:

pool-1-thread-1 (Start) message = 0

pool-1-thread-2 (Start) message = 1

pool-1-thread-3 (Start) message = 2

pool-1-thread-5 (Start) message = 4

pool-1-thread-4 (Start) message = 3

pool-1-thread-2 (End)

pool-1-thread-2 (Start) message = 5

pool-1-thread-1 (End)

pool-1-thread-1 (Start) message = 6

pool-1-thread-3 (End)

pool-1-thread-3 (Start) message = 7

pool-1-thread-4 (End)

pool-1-thread-4 (Start) message = 8

pool-1-thread-5 (End)

pool-1-thread-5 (Start) message = 9

pool-1-thread-2 (End)

pool-1-thread-1 (End)

pool-1-thread-4 (End)

pool-1-thread-3 (End)

pool-1-thread-5 (End)

Finished all threads

**ThreadPoolExecutor – Java Thread Pool Example**

SEPTEMBER 18, 2016 BY [PANKAJ](https://www.journaldev.com/author/pankaj) [47 COMMENTS](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice#comments)

**Java thread pool** manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed. We can use ThreadPoolExecutor to create thread pool in java.

[](https://cdn.journaldev.com/wp-content/uploads/2012/12/threadpoolexecutor-example-executorservice-java-thread-pool.jpg)

Java thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue. **java.util.concurrent.Executors** provide implementation of **java.util.concurrent.Executor**interface to create the thread pool in java. Let’s write a simple program to explain it’s working.

First we need to have a Runnable class, named WorkerThread.java

package com.journaldev.threadpool;

public class WorkerThread implements Runnable {

private String command;

public WorkerThread(String s){

this.command=s;

}

@Override

public void run() {

System.out.println(Thread.currentThread().getName()+" Start. Command = "+command);

processCommand();

System.out.println(Thread.currentThread().getName()+" End.");

}

private void processCommand() {

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public String toString(){

return this.command;

}

}

**ExecutorService Example**

Here is the test program class SimpleThreadPool.java, where we are creating fixed thread pool from **Executors framework**.

package com.journaldev.threadpool;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class SimpleThreadPool {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(5);

for (int i = 0; i < 10; i++) {

Runnable worker = new WorkerThread("" + i);

executor.execute(worker);

}

executor.shutdown();

while (!executor.isTerminated()) {

}

System.out.println("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.

Here is the output of the above program.

pool-1-thread-2 Start. Command = 1

pool-1-thread-4 Start. Command = 3

pool-1-thread-1 Start. Command = 0

pool-1-thread-3 Start. Command = 2

pool-1-thread-5 Start. Command = 4

pool-1-thread-4 End.

pool-1-thread-5 End.

pool-1-thread-1 End.

pool-1-thread-3 End.

pool-1-thread-3 Start. Command = 8

pool-1-thread-2 End.

pool-1-thread-2 Start. Command = 9

pool-1-thread-1 Start. Command = 7

pool-1-thread-5 Start. Command = 6

pool-1-thread-4 Start. Command = 5

pool-1-thread-2 End.

pool-1-thread-4 End.

pool-1-thread-3 End.

pool-1-thread-5 End.

pool-1-thread-1 End.

Finished all threads

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.

**ThreadPoolExecutor Example**

**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.

Here is our custom implementation of RejectedExecutionHandler interface.

package com.journaldev.threadpool;

import java.util.concurrent.RejectedExecutionHandler;

import java.util.concurrent.ThreadPoolExecutor;

public class RejectedExecutionHandlerImpl implements RejectedExecutionHandler {

@Override

public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {

System.out.println(r.toString() + " is rejected");

}

}

ThreadPoolExecutor provides several methods using which we can find out the current state of executor, pool size, active thread count and task count. So I have a monitor thread that will print the executor information at certain time interval.

package com.journaldev.threadpool;

import java.util.concurrent.ThreadPoolExecutor;

public class MyMonitorThread implements Runnable

{

private ThreadPoolExecutor executor;

private int seconds;

private boolean run=true;

public MyMonitorThread(ThreadPoolExecutor executor, int delay)

{

this.executor = executor;

this.seconds=delay;

}

public void shutdown(){

this.run=false;

}

@Override

public void run()

{

while(run){

System.out.println(

String.format("[monitor] [%d/%d] Active: %d, Completed: %d, Task: %d, isShutdown: %s, isTerminated: %s",

this.executor.getPoolSize(),

this.executor.getCorePoolSize(),

this.executor.getActiveCount(),

this.executor.getCompletedTaskCount(),

this.executor.getTaskCount(),

this.executor.isShutdown(),

this.executor.isTerminated()));

try {

Thread.sleep(seconds\*1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

Here is the thread pool implementation example using **ThreadPoolExecutor**.

package com.journaldev.threadpool;

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.Executors;

import java.util.concurrent.ThreadFactory;

import java.util.concurrent.ThreadPoolExecutor;

import java.util.concurrent.TimeUnit;

public class WorkerPool {

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

//RejectedExecutionHandler implementation

RejectedExecutionHandlerImpl rejectionHandler = new RejectedExecutionHandlerImpl();

//Get the ThreadFactory implementation to use

ThreadFactory threadFactory = Executors.defaultThreadFactory();

//creating the ThreadPoolExecutor

ThreadPoolExecutor executorPool = new ThreadPoolExecutor(2, 4, 10, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(2), threadFactory, rejectionHandler);

//start the monitoring thread

MyMonitorThread monitor = new MyMonitorThread(executorPool, 3);

Thread monitorThread = new Thread(monitor);

monitorThread.start();

//submit work to the thread pool

for(int i=0; i<10; i++){

executorPool.execute(new WorkerThread("cmd"+i));

}

Thread.sleep(30000);

//shut down the pool

executorPool.shutdown();

//shut down the monitor thread

Thread.sleep(5000);

monitor.shutdown();

}

}

Notice that while initializing the ThreadPoolExecutor, we are keeping initial pool size as 2, maximum pool size to 4 and work queue size as 2. So if there are 4 running tasks and more tasks are submitted, the work queue will hold only 2 of them and rest of them will be handled by RejectedExecutionHandlerImpl.

Here is the output of above program that confirms above statement.

pool-1-thread-1 Start. Command = cmd0

pool-1-thread-4 Start. Command = cmd5

cmd6 is rejected

pool-1-thread-3 Start. Command = cmd4

pool-1-thread-2 Start. Command = cmd1

cmd7 is rejected

cmd8 is rejected

cmd9 is rejected

[monitor] [0/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated: false

[monitor] [4/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated: false

pool-1-thread-4 End.

pool-1-thread-1 End.

pool-1-thread-2 End.

pool-1-thread-3 End.

pool-1-thread-1 Start. Command = cmd3

pool-1-thread-4 Start. Command = cmd2

[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated: false

[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated: false

pool-1-thread-1 End.

pool-1-thread-4 End.

[monitor] [4/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated: false

[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated: true

[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated: true

Notice the change in active, completed and total completed task count of the executor. We can invoke **shutdown()** method to finish execution of all the submitted tasks and terminate the thread pool.

If you want to schedule a task to run with delay or periodically then you can use **ScheduledThreadPoolExecutor** class. Read more about them at [**Java Schedule Thread Pool Executor**](https://www.journaldev.com/2340/java-scheduler-scheduledexecutorservice-scheduledthreadpoolexecutor-example).

Introduction to Thread Pools in Java

Last modified: August 27, 2017

by [Eugen Paraschiv](http://www.baeldung.com/author/eugen/)

* [**guava**](http://www.baeldung.com/category/guava/)
* [**Java**](http://www.baeldung.com/category/java/)**+**

If you're new here, you may want to [check out the "API Discoverability with Spring and Spring HATEOAS" live Webinar](http://www.baeldung.com/webinar). Thanks for visiting!

**If you’re working with Spring, check out "REST With Spring":**

[**>> CHECK OUT THE COURSE**](http://www.baeldung.com/rest-with-spring-course#certification-class)

**1. Introduction**

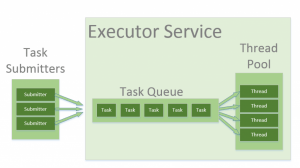
This article is a look at thread pools in Java – starting with the different implementations in the standard Java library and then looking at Google’s Guava library.

**2. The Thread Pool**

In Java, threads are mapped to system-level threads which are operating system’s resources. If you create threads uncontrollably, you may run out of these resources quickly.

The context switching between threads is done by the operating system as well – in order to emulate parallelism. A simplistic view is that – the more threads you spawn, the less time each thread spends doing actual work.

The Thread Pool pattern helps to save resources in a multithreaded application, and also to contain the parallelism in certain predefined limits.

When you use a thread pool, you **write your concurrent code in the form of parallel tasks and submit them for execution to an instance of a thread pool**. This instance controls several re-used threads for executing these tasks.  
[](http://www.baeldung.com/wp-content/uploads/2016/08/2016-08-10_10-16-52-1024x572.png)

The pattern allows you to **control the number of threads the application is creating**, their lifecycle, as well as to schedule tasks’ execution and keep incoming tasks in a queue.

**3. Thread Pools in Java**

**3.1. *Executors*, *Executor* and *ExecutorService***

The *Executors* helper class contains several methods for creation of pre-configured thread pool instances for you. Those classes are a good place to start with – use it if you don’t need to apply any custom fine-tuning.

The *Executor* and *ExecutorService* interfaces are used to work with different thread pool implementations in Java. Usually, you should **keep your code decoupled from the actual implementation of the thread pool**and use these interfaces throughout your application.

**The *Executor* interface has a single *execute* method to submit *Runnable*instances for execution.**

**Here’s a quick example** of how you can use the *Executors* API to acquire an *Executor* instance backed by a single thread pool and an unbounded queue for executing tasks sequentially. Here, we execute a single task that simply prints “*Hello World*” on the screen. The task is submitted as a lambda (a Java 8 feature) which is inferred to be *Runnable*.

|  |  |
| --- | --- |
| 1  2 | Executor executor = Executors.newSingleThreadExecutor();  executor.execute(() -> System.out.println("Hello World")); |

The *ExecutorService* interface contains a large number of methods for **controlling the progress of the tasks and managing the termination of the service**. Using this interface, you can submit the tasks for execution and also control their execution using the returned *Future* instance.

**In the following example**, we create an *ExecutorService*, submit a task and then use the returned *Future*‘s *get* method to wait until the submitted task is finished and the value is returned:

|  |  |
| --- | --- |
| 1  2  3  4 | ExecutorService executorService = Executors.newFixedThreadPool(10);  Future<String> future = executorService.submit(() -> "Hello World");  // some operations  String result = future.get(); |

Of course, in a real-life scenario you usually don’t want to call *future.get()* right away, but defer calling it until you actually need the value of the computation.

The *submit* method is overloaded to take either *Runnable* or *Callable* both of which are functional interfaces and can be passed as lambdas (starting with Java 8).

*Runnable*‘s single method does not throw an exception and does not return value. *Callable* interface may be more convenient, as it allows to throw an exception and return a value.

Finally – to let the compiler infer the *Callable* type, simply return a value from the lambda.

For more examples on using the *ExecutorService* interface and futures, have a look at “[A Guide to the Java ExecutorService](http://www.baeldung.com/java-executor-service-tutorial)“.

**3.2. *ThreadPoolExecutor***

The *ThreadPoolExecutor* is an extensible thread pool implementation with lots of parameters and hooks for fine-tuning.

The main configuration parameters that we’ll discuss here are: ***corePoolSize***, ***maximumPoolSize***, and ***keepAliveTime***.

The pool consists of a fixed number of core threads that are kept inside all the time, and some excessive threads that may be spawned and then terminated when they are not needed anymore. The *corePoolSize* parameter is the amount of core threads which will be instantiated and kept in the pool. If all core threads are busy and more tasks are submitted, then the pool is allowed to grow up to a *maximumPoolSize*.

The *keepAliveTime* parameter is the interval of time for which the excessive threads (i.e. threads that are instantiated in excess of the *corePoolSize*) are allowed to exist in the idle state.

These parameters cover a wide range of use cases, but **the most typical configurations are predefined in the *Executors* static methods**.

**For example**, *newFixedThreadPool* method creates a *ThreadPoolExecutor* with equal *corePoolSize* and *maximumPoolSize* parameter values and a zero *keepAliveTime.*This means thatthe number of threads in this thread pool is always the same:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | ThreadPoolExecutor executor =    (ThreadPoolExecutor) Executors.newFixedThreadPool(2);  executor.submit(() -> {      Thread.sleep(1000);      return null;  });  executor.submit(() -> {      Thread.sleep(1000);      return null;  });  executor.submit(() -> {      Thread.sleep(1000);      return null;  });    assertEquals(2, executor.getPoolSize());  assertEquals(1, executor.getQueue().size()); |

In the example above we instantiate a *ThreadPoolExecutor* with a fixed thread count of 2. This means that if the amount of simultaneously running tasks is less or equal to two at all times, then they get executed right away. Otherwise **some of these tasks may be put into a queue to wait for their turn**.

We created three *Callable* tasks that imitate heavy work by sleeping for 1000 milliseconds. The first two tasks will be executed at once, and the third one will have to wait in the queue. We can verify it by calling the *getPoolSize()* and *getQueue().size()* methods immediately after submitting the tasks.

Another pre-configured *ThreadPoolExecutor* can be created with the *Executors.newCachedThreadPool()* method. This method does not receive a number of threads at all. The *corePoolSize* is actually set to 0, and the *maximumPoolSize* is set to *Integer.MAX\_VALUE* for this instance. The *keepAliveTime* is 60 seconds for this one.

These parameter values mean that **the cached thread pool may grow without bounds to accommodate any amount of submitted tasks**. But when the threads are not needed anymore, they will be disposed of after 60 seconds of inactivity. A typical use case is when you have a lot of short-living tasks in your application.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | ThreadPoolExecutor executor =    (ThreadPoolExecutor) Executors.newCachedThreadPool();  executor.submit(() -> {      Thread.sleep(1000);      return null;  });  executor.submit(() -> {      Thread.sleep(1000);      return null;  });  executor.submit(() -> {      Thread.sleep(1000);      return null;  });    assertEquals(3, executor.getPoolSize());  assertEquals(0, executor.getQueue().size()); |

The queue size in the example above will always be zero because internally a *SynchronousQueue* instance is used. In a *SynchronousQueue*, pairs of *insert* and *remove* operations always occur simultaneously, so the queue never actually contains anything.

The *Executors.newSingleThreadExecutor()* API creates another typical form of *ThreadPoolExecutor* containing a single thread. **The single thread executor is ideal for creating an event loop.** The *corePoolSize* and *maximumPoolSize*parameters are equal to 1, and the *keepAliveTime* is zero.

Tasks in the above example will be executed sequentially, so the flag value will be 2 after task’s completion:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | AtomicInteger counter = new AtomicInteger();    ExecutorService executor = Executors.newSingleThreadExecutor();  executor.submit(() -> {      counter.set(1);  });  executor.submit(() -> {      counter.compareAndSet(1, 2);  }); |

Additionally, this *ThreadPoolExecutor* is decorated with an immutable wrapper, so it cannot be reconfigured after creation. Note that also this is the reason we cannot cast it to a *ThreadPoolExecutor*.

**3.3. *ScheduledThreadPoolExecutor***

The *ScheduledThreadPoolExecutor* extends the *ThreadPoolExecutor* class and also implements the *ScheduledExecutorService* interface with several additional methods:

* *schedule* method allows to execute a task once after a specified delay;
* *scheduleAtFixedRate* method allows to execute a task after a specified initial delay and then execute it repeatedly with a certain period; the *period*argument is the time **measured between the starting times of the tasks**, so the execution rate is fixed;
* *scheduleWithFixedDelay* method is similar to *scheduleAtFixedRate* in that it repeatedly executes the given task, but the specified delay is **measured between the end of the previous task and the start of the next**; the execution rate may vary depending on the time it takes to execute any given task.

The *Executors.newScheduledThreadPool()* method is typically used to create a *ScheduledThreadPoolExecutor* with a given *corePoolSize*, unbounded *maximumPoolSize* and zero *keepAliveTime*. Here’s how to schedule a task for execution in 500 milliseconds:

|  |  |
| --- | --- |
| 1  2  3  4 | ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);  executor.schedule(() -> {      System.out.println("Hello World");  }, 500, TimeUnit.MILLISECONDS); |

The following code shows how to execute a task after 500 milliseconds delay and then repeat it every 100 milliseconds. After scheduling the task, we wait until it fires three times using the *CountDownLatch* lock*,* then cancel it using the *Future.cancel()* method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | CountDownLatch lock = new CountDownLatch(3);    ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);  ScheduledFuture<?> future = executor.scheduleAtFixedRate(() -> {      System.out.println("Hello World");      lock.countDown();  }, 500, 100, TimeUnit.MILLISECONDS);    lock.await(1000, TimeUnit.MILLISECONDS);  future.cancel(true); |

**3.4. *ForkJoinPool***

*ForkJoinPool* is the central part of the *fork/join* framework introduced in Java 7. It solves a common problem of **spawning multiple tasks in recursive algorithms**. Using a simple *ThreadPoolExecutor*, you will run out of threads quickly, as every task or subtask requires its own thread to run.

In a *fork/join* framework, any task can spawn (*fork*) a number of subtasks and wait for their completion using the *join* method. The benefit of the *fork/join* framework is that it **does not create a new thread for each task or subtask**, implementing the Work Stealing algorithm instead. This framework is thoroughly described in the article “[Guide to the Fork/Join Framework in Java](http://www.baeldung.com/java-fork-join)“

Let’s look at a simple example of using *ForkJoinPool* to traverse a tree of nodes and calculate the sum of all leaf values. Here’s a simple implementation of a tree consisting of a node, an *int* value and a set of child nodes:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | static class TreeNode {        int value;        Set<TreeNode> children;        TreeNode(int value, TreeNode... children) {          this.value = value;          this.children = Sets.newHashSet(children);      }  } |

Now if we want to sum all values in a tree in parallel, we need to implement a *RecursiveTask<Integer>* interface. Each task receives its own node and adds its value to the sum of values of its *children*. To calculate the sum of *children* values, the task implementation does the following:

* streams the *children* set,
* maps over this stream, creating a new *CountingTask* for each element,
* executes each subtask by forking it,
* collects the results by calling the *join* method on each forked task,
* sums the results using the *Collectors.summingInt* collector.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public static class CountingTask extends RecursiveTask<Integer> {        private final TreeNode node;        public CountingTask(TreeNode node) {          this.node = node;      }        @Override      protected Integer compute() {          return node.value + node.children.stream()            .map(childNode -> new CountingTask(childNode).fork())            .collect(Collectors.summingInt(ForkJoinTask::join));      }  } |

The code to run the calculation on an actual tree is very simple:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | TreeNode tree = new TreeNode(5,    new TreeNode(3), new TreeNode(2,      new TreeNode(2), new TreeNode(8)));    ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();  int sum = forkJoinPool.invoke(new CountingTask(tree)); |

**4. Thread Pool’s Implementation in Guava**

[Guava](https://github.com/google/guava) is a popular Google library of utilities. It has many useful concurrency classes, including several handy implementations of *ExecutorService*. The implementing classes are not accessible for direct instantiation or subclassing, so the only entry point for creating their instances is the *MoreExecutors* helper class.

**4.1. Adding Guava as a Maven Dependency**

Add the following dependency to your Maven pom file to include the Guava library to your project. You can find the latest version of Guava library in the [Maven Central](http://search.maven.org/#search%7Cgav%7C1%7Cg%3A%22com.google.guava%22%20AND%20a%3A%22guava%22)repository:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>com.google.guava</groupId>      <artifactId>guava</artifactId>      <version>19.0</version>  </dependency> |

**4.2. Direct Executor and Direct Executor Service**

Sometimes you want to execute the task either in the current thread, or in a thread pool, depending on some conditions. You would prefer to use a single *Executor*interface and just switch the implementation. Although it is not so hard to come up with an implementation of *Executor* or *ExecutorService* that executes the tasks in the current thread, it still requires writing some boilerplate code.

Gladly, Guava provides predefined instances for us.

**Here’s an example** that demonstrates execution of a task in the same thread. Although the provided task sleeps for 500 milliseconds, it **blocks the current thread**, and the result is available immediately after the *execute* call is finished:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Executor executor = MoreExecutors.directExecutor();    AtomicBoolean executed = new AtomicBoolean();    executor.execute(() -> {      try {          Thread.sleep(500);      } catch (InterruptedException e) {          e.printStackTrace();      }      executed.set(true);  });    assertTrue(executed.get()); |

The instance returned by the *directExecutor()* method is actually a static singleton, so using this method does not provide any overhead on object creation at all.

You should prefer this method to the *MoreExecutors.newDirectExecutorService()*, because that API creates a full-fledged executor service implementation on every call.

**4.3. Exiting Executor Services**

Another common problem is **shutting down the virtual machine** while a thread pool is still running its tasks. Even with a cancellation mechanism in place, there is no guarantee that the tasks will behave nicely and stop their work when the executor service shuts down. This may cause JVM to hang indefinitely while the tasks keep doing their work.

To solve this problem, Guava introduces a family of exiting executor services. They are based on **daemon threads which terminate together with the JVM**.

These services also add a shutdown hook with the *Runtime.getRuntime().addShutdownHook()* method and prevent the VM from terminating for a configured amount of time before giving up on hung tasks.

In the following example we’re submitting the task that contains an infinite loop, but we use an exiting executor service with a configured time of 100 milliseconds to wait for the tasks upon VM termination. Without the *exitingExecutorService* in place, this task would cause the VM to hang indefinitely:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | ThreadPoolExecutor executor =    (ThreadPoolExecutor) Executors.newFixedThreadPool(5);  ExecutorService executorService =    MoreExecutors.getExitingExecutorService(executor,      100, TimeUnit.MILLISECONDS);    executorService.submit(() -> {      while (true) {      }  }); |

**4.4. Listening Decorators**

Listening decorators allow you to wrap the *ExecutorService* and receive *ListenableFuture* instances upon task submission instead of simple *Future*instances. The *ListenableFuture* interface extends *Future* and has a single additional method *addListener*. This method allows adding a listener that is called upon future completion.

You’ll rarely want to use *ListenableFuture.addListener()* method directly, but it is **essential to most of the helper methods in the *Futures* utility class**.  For instance, with the *Futures.allAsList()* method you can combine several *ListenableFuture*instances in a single *ListenableFuture* that completes upon the successful completion of all the futures combined:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | ExecutorService executorService = Executors.newCachedThreadPool();  ListeningExecutorService listeningExecutorService =    MoreExecutors.listeningDecorator(executorService);    ListenableFuture<String> future1 =    listeningExecutorService.submit(() -> "Hello");  ListenableFuture<String> future2 =    listeningExecutorService.submit(() -> "World");    String greeting = Futures.allAsList(future1, future2).get()    .stream()    .collect(Collectors.joining(" "));  assertEquals("Hello World", greeting); |

**5. Conclusion**

In this article we have discussed the Thread Pool pattern and its implementations in the standard Java library and in the Google’s Guava library.

The source code for the article is available [over on GitHub](https://github.com/eugenp/tutorials/tree/master/core-java-concurrency).