**ScheduledExecutorService**

* [ScheduledExecutorService Example](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#scheduledexecutorservice-example)
* [ScheduledExecutorService Implementations](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#scheduledexecutorservice-implementations)
* [Creating a ScheduledExecutorService](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#creating-a-scheduledexecutorservice)
* [ScheduledExecutorService Usage](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#scheduledexecutorservice-usage)
  + [schedule (Callable task, long delay, TimeUnit timeunit)](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#schedule-callable)
  + [schedule (Runnable task, long delay, TimeUnit timeunit)](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#schedule-runnable)
  + [scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#scheduleatfixedrate)
  + [scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#schedulewithfixeddelay)
* [ScheduledExecutorService Shutdown](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html#scheduledexecutorservice-shutdown)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

The java.util.concurrent.ScheduledExecutorService is an [**ExecutorService**](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html) which can schedule tasks to run after a delay, or to execute repeatedly with a fixed interval of time in between each execution. Tasks are executed asynchronously by a worker thread, and not by the thread handing the task to the ScheduledExecutorService.

**ScheduledExecutorService Example**

Here is a simple ScheduledExecutorService example:

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

First a ScheduledExecutorService is created with 5 threads in. Then an anonymous implementation of the Callable interface is created and passed to the schedule() method. The two last parameters specify that the Callable should be executed after 5 seconds.

**ScheduledExecutorService Implementations**

Since ScheduledExecutorService is an interface, you will have to use its implementation in the java.util.concurrent package, in order to use it. ScheduledExecutorService as the following implementation:

* ScheduledThreadPoolExecutor

**Creating a ScheduledExecutorService**

How you create an ScheduledExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ScheduledExecutorService instances too. Here is an example:

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

**ScheduledExecutorService Usage**

Once you have created a ScheduledExecutorService you use it by calling one of its methods:

* schedule (Callable task, long delay, TimeUnit timeunit)
* schedule (Runnable task, long delay, TimeUnit timeunit)
* scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)
* scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

I will briefly cover each of these methods below.

**schedule (Callable task, long delay, TimeUnit timeunit)**

This method schedules the given Callable for execution after the given delay.

The method returns a ScheduledFuture which you can use to either cancel the task before it has started executing, or obtain the result once it is executed.

Here is an example:

ScheduledExecutorService scheduledExecutorService =

Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =

scheduledExecutorService.schedule(new Callable() {

public Object call() throws Exception {

System.out.println("Executed!");

return "Called!";

}

},

5,

TimeUnit.SECONDS);

System.out.println("result = " + scheduledFuture.get());

scheduledExecutorService.shutdown();

This example outputs:

Executed!

result = Called!

**schedule (Runnable task, long delay, TimeUnit timeunit)**

This method works like the method version taking a Callable as parameter, except a Runnable cannot return a value, so the ScheduledFuture.get() method returns null when the task is finished.

**scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)**

This method schedules a task to be executed periodically. The task is executed the first time after the initialDelay, and then recurringly every time the period expires.

If any execution of the given task throws an exception, the task is no longer executed. If no exceptions are thrown, the task will continue to be executed until the ScheduledExecutorService is shut down.

If a task takes longer to execute than the period between its scheduled executions, the next execution will start after the current execution finishes. The scheduled task will not be executed by more than one thread at a time.

**scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)**

This method works very much like scheduleAtFixedRate() except that the period is interpreted differently.

In the scheduleAtFixedRate() method the period is interpreted as a delay between the start of the previous execution, until the start of the next execution.

In this method, however, the period is interpreted as the delay between the **end** of the previous execution, until the start of the next. The delay is thus between finished executions, not between the beginning of executions.

**ScheduledExecutorService Shutdown**

Just like an ExecutorService, the ScheduledExecutorService needs to be shut down when you are finished using it. If not, it will keep the JVM running, even when all other threads have been shut down.

You shut down a ScheduledExecutorService using the shutdown() or shutdownNow() methods which are inherited from the ExecutorService interface. See the [**ExecutorService Shutdown**](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html#executorservice-shutdown) section for more information.

**Semaphore**

* [Semaphore Usage](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html#semaphore-usage)
  + [Guarding Critical Sections](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html#guarding-critical-sections)
  + [Sending Signals Between Threads](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html#sending-signals-between-threads)
* [Fairness](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html#fairness)
* [More Methods](http://tutorials.jenkov.com/java-util-concurrent/semaphore.html#more-methods)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

The java.util.concurrent.Semaphore class is a [**counting semaphore**](http://tutorials.jenkov.com/java-concurrency/semaphores.html#counting). That means that it has two main methods:

* acquire()
* release()

The counting semaphore is initialized with a given number of "permits". For each call to acquire() a permit is taken by the calling thread. For each call to release() a permit is returned to the semaphore. Thus, at most N threads can pass the acquire() method without any release() calls, where N is the number of permits the semaphore was initialized with. The permits are just a simple counter. Nothing fancy here.

**Semaphore Usage**

As semaphore typically has two uses:

1. To guard a critical section against entry by more than N threads at a time.
2. To send signals between two threads.

**Guarding Critical Sections**

If you use a semaphore to guard a critical section, the thread trying to enter the critical section will typically first try to acquire a permit, enter the critical section, and then release the permit again after. Like this:

Semaphore semaphore = new Semaphore(1);

//critical section

semaphore.acquire();

...

semaphore.release();

**Sending Signals Between Threads**

If you use a semaphore to send signals between threads, then you would typically have one thread call the acquire() method, and the other thread to call the release() method.

If no permits are available, the acquire() call will block until a permit is released by another thread. Similarly, a release() calls is blocked if no more permits can be released into this semaphore.

Thus it is possible to coordinate threads. For instance, if acquire was called after Thread 1 had inserted an object in a shared list, and Thread 2 had called release() just before taking an object from that list, you had essentially created a blocking queue. The number of permits available in the semaphore would correspond to the maximum number of elements the blocking queue could hold.

**Fairness**

No guarantees are made about [**fairness**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html) of the threads acquiring permits from the Semaphore. That is, there is no guarantee that the first thread to call acquire() is also the first thread to obtain a permit. If the first thread is blocked waiting for a permit, then a second thread checking for a permit just as a permit is released, may actually obtain the permit ahead of thread 1.

If you want to enforce fairness, the Semaphore class has a constructor that takes a boolean telling if the semaphore should enforce fairness. Enforcing fairness comes at a performance / concurrency penalty, so don't enable it unless you need it.

Here is how to create a Semaphore in fair mode:

Semaphore semaphore = new Semaphore(1, true);

**More Methods**

The java.util.concurrent.Semaphore class has lots more methods. For instance:

* availablePermits()
* acquireUninterruptibly()
* drainPermits()
* hasQueuedThreads()
* getQueuedThreads()
* tryAcquire()
* etc.

Check out the JavaDoc for more details on these methods.

The Java BlockingQueue interface in the java.util.concurrent package represents a queue which is thread safe to put into, and take instances from. In this text I will show you how to use this BlockingQueue.

This text will not discuss how to implement a BlockingQueue in Java yourself. If you are interested in that, I have a text on [**Blocking Queues**](http://tutorials.jenkov.com/java-concurrency/blocking-queues.html) in my more theoretical [**Java Concurrency Tutorial**](http://tutorials.jenkov.com/java-concurrency/index.html).

**BlockingQueue Usage**

A BlockingQueue is typically used to have on thread produce objects, which another thread consumes. Here is a diagram that illustrates this principle:

|  |
| --- |
| A BlockingQueue with one thread putting into it, and another thread taking from it. |
| **A BlockingQueue with one thread putting into it, and another thread taking from it.** |

The producing thread will keep producing new objects and insert them into the queue, until the queue reaches some upper bound on what it can contain. It's limit, in other words. If the blocking queue reaches its upper limit, the producing thread is blocked while trying to insert the new object. It remains blocked until a consuming thread takes an object out of the queue.

The consuming thread keeps taking objects out of the blocking queue, and processes them. If the consuming thread tries to take an object out of an empty queue, the consuming thread is blocked until a producing thread puts an object into the queue.

**BlockingQueue Methods**

A BlockingQueue has 4 different sets of methods for inserting, removing and examining the elements in the queue. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | add(o) | offer(o) | put(o) | offer(o, timeout, timeunit) |
| **Remove** | remove(o) | poll() | take() | poll(timeout, timeunit) |
| **Examine** | element() | peek() |  |  |

The 4 different sets of behaviour means this:

1. **Throws Exception**:   
   If the attempted operation is not possible immediately, an exception is thrown.
2. **Special Value**:   
   If the attempted operation is not possible immediately, a special value is returned (often true / false).
3. **Blocks**:   
   If the attempted operation is not possible immedidately, the method call blocks until it is.
4. **Times Out**:   
   If the attempted operation is not possible immedidately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

It is not possible to insert null into a BlockingQueue. If you try to insert null, the BlockingQueue will throw a NullPointerException.

It is also possible to access all the elements inside a BlockingQueue, and not just the elements at the start and end. For instance, say you have queued an object for processing, but your application decides to cancel it. You can then call e.g. remove(o) to remove a specific object in the queue. However, this is not done very efficiently, so you should not use these Collection methods unless you really have to.

**BlockingQueue Implementations**

Since BlockingQueue is an interface, you need to use one of its implementations to use it. The java.util.concurrent package has the following implementations of the BlockingQueue interface (in Java 6):

* [**ArrayBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/arrayblockingqueue.html)
* [**DelayQueue**](http://tutorials.jenkov.com/java-util-concurrent/delayqueue.html)
* [**LinkedBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/linkedblockingqueue.html)
* [**PriorityBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/priorityblockingqueue.html)
* [**SynchronousQueue**](http://tutorials.jenkov.com/java-util-concurrent/synchronousqueue.html)

Click the links in the list to read more about each implementation. If a link cannot be clicked, that implementation has not yet been described. Check back again in the future, or check out the JavaDoc's for more detail.

**Java BlockingQueue Example**

Here is a Java BlockingQueue example. The example uses the ArrayBlockingQueue implementation of the BlockingQueue interface.

First, the BlockingQueueExample class which starts a Producer and a Consumer in separate threads. The Producer inserts strings into a shared BlockingQueue, and the Consumer takes them out.

public class BlockingQueueExample {

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

BlockingQueue queue = new ArrayBlockingQueue(1024);

Producer producer = new Producer(queue);

Consumer consumer = new Consumer(queue);

new Thread(producer).start();

new Thread(consumer).start();

Thread.sleep(4000);

}

}

Here is the Producer class. Notice how it sleeps a second between each put() call. This will cause the Consumer to block, while waiting for objects in the queue.

public class Producer implements Runnable{

protected BlockingQueue queue = null;

public Producer(BlockingQueue queue) {

this.queue = queue;

}

public void run() {

try {

queue.put("1");

Thread.sleep(1000);

queue.put("2");

Thread.sleep(1000);

queue.put("3");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Here is the Consumer class. It just takes out the objects from the queue, and prints them to System.out.

public class Consumer implements Runnable{

protected BlockingQueue queue = null;

public Consumer(BlockingQueue queue) {

this.queue = queue;

}

public void run() {

try {

System.out.println(queue.take());

System.out.println(queue.take());

System.out.println(queue.take());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

# Exchanger

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

# Exchanger

The java.util.concurrent.Exchanger class represents a kind of rendezvous point where two threads can exchange objects. Here is an illustration of this mechanism:

|  |
| --- |
| Two threads exchanging objects via an Exchanger. |
| **Two threads exchanging objects via an Exchanger.** |

Exchanging objects is done via one of the two exchange() methods. Here is an example:

Exchanger exchanger = new Exchanger();

ExchangerRunnable exchangerRunnable1 =

new ExchangerRunnable(exchanger, "A");

ExchangerRunnable exchangerRunnable2 =

new ExchangerRunnable(exchanger, "B");

new Thread(exchangerRunnable1).start();

new Thread(exchangerRunnable2).start();

Here is the ExchangerRunnable code:

public class ExchangerRunnable implements Runnable{

Exchanger exchanger = null;

Object object = null;

public ExchangerRunnable(Exchanger exchanger, Object object) {

this.exchanger = exchanger;

this.object = object;

}

public void run() {

try {

Object previous = this.object;

this.object = this.exchanger.exchange(this.object);

System.out.println(

Thread.currentThread().getName() +

" exchanged " + previous + " for " + this.object

);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

This example prints out this:

Thread-0 exchanged A for B

Thread-1 exchanged B for A

# PriorityBlockingQueue

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

The PriorityBlockingQueue class implements the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The PriorityBlockingQueue is an unbounded concurrent queue. It uses the same ordering rules as thejava.util.PriorityQueue class. You cannot insert null into this queue.

All elements inserted into the PriorityBlockingQueue must implement the java.lang.Comparable interface. The elements thus order themselves according to whatever priority you decide in your Comparableimplementation.

Notice that the PriorityBlockingQueue does not enforce any specific behaviour for elements that have equal priority (compare() == 0).

Also notice, that in case you obtain an Iterator from a PriorityBlockingQueue, the Iterator does not guarantee to iterate the elements in priority order.

Here is an example of how to use the PriorityBlockingQueue:

BlockingQueue queue = new PriorityBlockingQueue();

//String implements java.lang.Comparable

queue.put("Value");

String value = queue.take();

# SynchronousQueue

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

The SynchronousQueue class implements the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The SynchronousQueue is a queue that can only contain a single element internally. A thread inseting an element into the queue is blocked until another thread takes that element from the queue. Likewise, if a thread tries to take an element and no element is currently present, that thread is blocked until a thread insert an element into the queue.

Calling this class a queue is a bit of an overstatement. It's more of a rendesvouz point.

**BlockingDeque**

* [BlockingDeque Usage](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html#blocking-deque-usage)
  + [BlockingDeque methods](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html#blocking-deque-methods)
* [BlockingDeque Extends BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html#blockingdeque-extends-blockingqueue)
* [BlockingDeque Implementations](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html#blockingdeque-implementations)
* [BlockingDeque Code Example](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html#blockingdeque-code-example)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

The BlockingDeque interface in the java.util.concurrent class represents a deque which is thread safe to put into, and take instances from. In this text I will show you how to use this BlockingDeque.

The BlockingDeque class is a Deque which blocks threads tring to insert or remove elements from the deque, in case it is either not possible to insert or remove elements from the deque.

A deque is short for "Double Ended Queue". Thus, a deque is a queue which you can insert and take elements from, from both ends.

**BlockingDeque Usage**

A BlockingDeque could be used if threads are both producing and consuming elements of the same queue. It could also just be used if the producting thread needs to insert at both ends of the queue, and the consuming thread needs to remove from both ends of the queue. Here is an illustration of that:

|  |
| --- |
| A BlockingDeque - threads can put and take from both ends of the deque. |
| **A BlockingDeque - threads can put and take from both ends of the deque.** |

A thread will produce elements and insert them into either end of the queue. If the deque is currently full, the inserting thread will be blocked until a removing thread takes an element out of the deque. If the deque is currently empty, a removing thread will be blocked until an inserting thread inserts an element into the deque.

**BlockingDeque methods**

A BlockingDeque has 4 different sets of methods for inserting, removing and examining the elements in the deque. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | addFirst(o) | offerFirst(o) | putFirst(o) | offerFirst(o, timeout, timeunit) |
| **Remove** | removeFirst(o) | pollFirst(o) | takeFirst(o) | pollFirst(timeout, timeunit) |
| **Examine** | getFirst(o) | peekFirst(o) |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | addLast(o) | offerLast(o) | putLast(o) | offerLast(o, timeout, timeunit) |
| **Remove** | removeLast(o) | pollLast(o) | takeLast(o) | pollLast(timeout, timeunit) |
| **Examine** | getLast(o) | peekLast(o) |  |  |

The 4 different sets of behaviour means this:

1. **Throws Exception**:   
   If the attempted operation is not possible immediately, an exception is thrown.
2. **Special Value**:   
   If the attempted operation is not possible immediately, a special value is returned (often true / false).
3. **Blocks**:   
   If the attempted operation is not possible immedidately, the method call blocks until it is.
4. **Times Out**:   
   If the attempted operation is not possible immedidately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

**BlockingDeque Extends BlockingQueue**

The BlockingDeque interface extends the BlockingQueue interface. That means that you can use a BlockingDeque as a BlockingQueue. If you do so, the various inserting methods will add the elements to the end of the deque, and the removing methods will remove the elements from the beginning of the deque. The inserting and removing methods of the BlockingQueue interface, that is.

Here is a table of what the methods of the BlockingQueue does in a BlockingDeque implementation:

|  |  |
| --- | --- |
| **BlockingQueue** | **BlockingDeque** |
| add() | addLast() |
| offer() x 2 | offerLast() x 2 |
| put() | putLast() |
|  |  |
| remove() | removeFirst() |
| poll() x 2 | pollFirst() |
| take() | takeFirst() |
|  |  |
| element() | getFirst() |
| peek() | peekFirst() |

**BlockingDeque Implementations**

Since BlockingDeque is an interface, you need to use one of its many implementations to use it. The java.util.concurrent package has the following implementations of the BlockingDeque interface:

* [**LinkedBlockingDeque**](http://tutorials.jenkov.com/java-util-concurrent/linkedblockingdeque.html)

**BlockingDeque Code Example**

Here is a small code example of how to use the BlockingDeque methods:

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

# DelayQueue

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-06-05 |

DelayQueue class implements the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The DelayQueue blocks the elements internally until a certain delay has expired. The elements must implement the interface java.util.concurrent.Delayed. Here is how the interface looks:

public interface Delayed extends Comparable<Delayed< {

public long getDelay(TimeUnit timeUnit);

}

The value returned by the getDelay() method should be the delay remaining before this element can be released. If 0 or a negative value is returned, the delay will be considered expired, and the element released at the next take() etc. call on the DelayQueue.

The TimeUnit instance passed to the getDelay() method is an Enum that tells which time unit the delay should be returned in. The TimeUnit enum can take these values:

DAYS

HOURS

MINUTES

SECONDS

MILLISECONDS

MICROSECONDS

NANOSECONDS

The Delayed interface also extends the java.lang.Comparable interface, as you can see, which means that Delayed objects can be compared to each other. This is probably used internally in the DelayQueue to order the elements in the queue, so they are released ordered by their expiration time.

Here is an example of how to use the DelayQueue:

public class DelayQueueExample {

public static void main(String[] args) {

DelayQueue queue = new DelayQueue();

Delayed element1 = new DelayedElement();

queue.put(element1);

Delayed element2 = queue.take();

}

}

The DelayedElement is an implementation of the Delayed interface that I have created. It is not part of the java.util.concurrent package. You will have to create your own implementation of the Delayed interface to use the DelayQueue class.

# LinkedBlockingDeque

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

The LinkedBlockingDeque class implements the [**BlockingDeque**](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html) interface. Read the [**BlockingDeque**](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html) text for more information about the interface.

The word Deque comes from the term "Double Ended Queue". A Deque is thus a queue where you can insert and remove elements from both ends of the queue.

The LinkedBlockingDeque is a Deque which will block if a thread attempts to take elements out of it while it is empty, regardless of what end the thread is attempting to take elements from.

Here is how to instantiate and use a LinkedBlockingDeque:

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

# LinkedBlockingQueue

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

The LinkedBlockingQueue class implements the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The LinkedBlockingQueue keeps the elements internally in a linked structure (linked nodes). This linked structure can optionally have an upper bound if desired. If no upper bound is specified, Integer.MAX\_VALUEis used as the upper bound.

The LinkedBlockingQueue stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use a LinkedBlockingQueue:

BlockingQueue<String> unbounded = new LinkedBlockingQueue<String>();

BlockingQueue<String> bounded = new LinkedBlockingQueue<String>(1024);

bounded.put("Value");

String value = bounded.take();

# CountDownLatch

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

A java.util.concurrent.CountDownLatch is a concurrency construct that allows one or more threads to wait for a given set of operations to complete.

A CountDownLatch is initialized with a given count. This count is decremented by calls to the countDown()method. Threads waiting for this count to reach zero can call one of the await() methods. Calling await()blocks the thread until the count reaches zero.

Below is a simple example. After the Decrementer has called countDown() 3 times on the CountDownLatch, the waiting Waiter is released from the await() call.

CountDownLatch latch = new CountDownLatch(3);

Waiter waiter = new Waiter(latch);

Decrementer decrementer = new Decrementer(latch);

new Thread(waiter) .start();

new Thread(decrementer).start();

Thread.sleep(4000);

public class Waiter implements Runnable{

CountDownLatch latch = null;

public Waiter(CountDownLatch latch) {

this.latch = latch;

}

public void run() {

try {

latch.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Waiter Released");

}

}

public class Decrementer implements Runnable {

CountDownLatch latch = null;

public Decrementer(CountDownLatch latch) {

this.latch = latch;

}

public void run() {

try {

Thread.sleep(1000);

this.latch.countDown();

Thread.sleep(1000);

this.latch.countDown();

Thread.sleep(1000);

this.latch.countDown();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**CyclicBarrier**

* [Creating a CyclicBarrier](http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html#creating-a-cyclicbarrier)
* [Waiting at a CyclicBarrier](http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html#waiting-at-a-cyclicbarrier)
* [CyclicBarrier Action](http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html#cyclicbarrier-action)
* [CyclicBarrier Example](http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html#cyclicbarrier-example)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

The java.util.concurrent.CyclicBarrier class is a synchronization mechanism that can synchronize threads progressing through some algorithm. In other words, it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue. Here is a diagram illustrating that:

|  |
| --- |
| Two threads waiting for each other at CyclicBarriers. |
| **Two threads waiting for each other at CyclicBarriers.** |

The threads wait for each other by calling the await() method on the CyclicBarrier. Once N threads are waiting at the CyclicBarrier, all threads are released and can continue running.

**Creating a CyclicBarrier**

When you create a CyclicBarrier you specify how many threads are to wait at it, before releasing them. Here is how you create a CyclicBarrier:

CyclicBarrier barrier = new CyclicBarrier(2);

**Waiting at a CyclicBarrier**

Here is how a thread waits at a CyclicBarrier:

barrier.await();

You can also specify a timeout for the waiting thread. When the timeout has passed the thread is also released, even if not all N threads are waiting at the CyclicBarrier. Here is how you specify a timeout:

barrier.await(10, TimeUnit.SECONDS);

The waiting threads waits at the CyclicBarrier until either:

* The last thread arrives (calls await() )
* The thread is interrupted by another thread (another thread calls its interrupt() method)
* Another waiting thread is interrupted
* Another waiting thread times out while waiting at the CyclicBarrier
* The CyclicBarrier.reset() method is called by some external thread.

**CyclicBarrier Action**

The CyclicBarrier supports a barrier action, which is a Runnable that is executed once the last thread arrives. You pass the Runnable barrier action to the CyclicBarrier in its constructor, like this:

Runnable barrierAction = ... ;

CyclicBarrier barrier = new CyclicBarrier(2, barrierAction);

**CyclicBarrier Example**

Here is a code example that shows you how to use a CyclicBarrier:

Runnable barrier1Action = new Runnable() {

public void run() {

System.out.println("BarrierAction 1 executed ");

}

};

Runnable barrier2Action = new Runnable() {

public void run() {

System.out.println("BarrierAction 2 executed ");

}

};

CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);

CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);

CyclicBarrierRunnable barrierRunnable1 =

new CyclicBarrierRunnable(barrier1, barrier2);

CyclicBarrierRunnable barrierRunnable2 =

new CyclicBarrierRunnable(barrier1, barrier2);

new Thread(barrierRunnable1).start();

new Thread(barrierRunnable2).start();

Here is the CyclicBarrierRunnable class:

public class CyclicBarrierRunnable implements Runnable{

CyclicBarrier barrier1 = null;

CyclicBarrier barrier2 = null;

public CyclicBarrierRunnable(

CyclicBarrier barrier1,

CyclicBarrier barrier2) {

this.barrier1 = barrier1;

this.barrier2 = barrier2;

}

public void run() {

try {

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() +

" waiting at barrier 1");

this.barrier1.await();

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() +

" waiting at barrier 2");

this.barrier2.await();

System.out.println(Thread.currentThread().getName() +

" done!");

} catch (InterruptedException e) {

e.printStackTrace();

} catch (BrokenBarrierException e) {

e.printStackTrace();

}

}

}

Here is the console output for an execution of the above code. Note that the sequence in which the threads gets to write to the console may vary from execution to execution. Sometimes Thread-0 prints first, sometimes Thread-1 prints first etc.

Thread-0 waiting at barrier 1

Thread-1 waiting at barrier 1

BarrierAction 1 executed

Thread-1 waiting at barrier 2

Thread-0 waiting at barrier 2

BarrierAction 2 executed

Thread-0 done!

Thread-1 done!

**ava Fork and Join using ForkJoinPool**

* [Fork and Join Explained](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#fork-and-join-explained)
  + [Fork](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#fork)
  + [Join](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#join)
* [The ForkJoinPool](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#the-forkjoinpool)
  + [Creating a ForkJoinPool](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#creating-a-forkjoinpool)
  + [Submitting Tasks to the ForkJoinPool](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#submittting-tasks-to-forkjoinpool)
* [RecursiveAction](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#recursive-action)
* [RecursiveTask](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#recursive-task)
* [ForkJoinPool Critique](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html#forkjoinpool-critique)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-02-03 |

The ForkJoinPool was added to Java in Java 7. The ForkJoinPool is similar to the [**Java ExecutorService**](http://tutorials.jenkov.com/java-util-concurrent/executorservice.html)but with one difference. The ForkJoinPool makes it easy for tasks to split their work up into smaller tasks which are then submitted to the ForkJoinPool too. Tasks can keep splitting their work into smaller subtasks for as long as it makes to split up the task. It may sound a bit abstract, so in this fork and join tutorial I will explain how the ForkJoinPool works, and how splitting tasks up work.

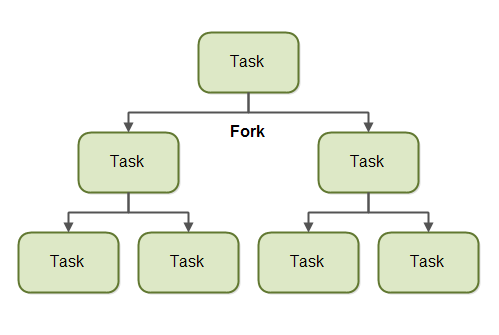
**Fork and Join Explained**

Before we look at the ForkJoinPool I want to explain how the fork and join principle works in general.

The fork and join principle consists of two steps which are performed recursively. These two steps are the fork step and the join step.

**Fork**

A task that uses the fork and join principle can *fork* (split) itself into smaller subtasks which can be executed concurrently. This is illustrated in the diagram below:



By splitting itself up into subtasks, each subtask can be executed in parallel by different CPUs, or different threads on the same CPU.

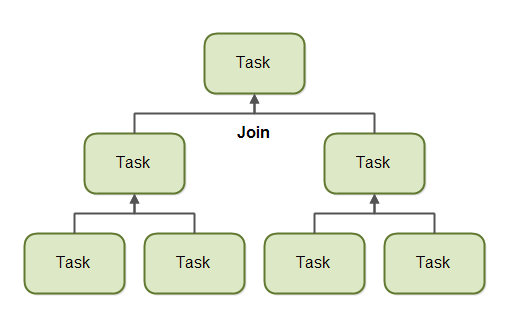
A task only splits itself up into subtasks if the work the task was given is large enough for this to make sense. There is an overhead to splitting up a task into subtasks, so for small amounts of work this overhead may be greater than the speedup achieved by executing subtasks concurrently.

The limit for when it makes sense to fork a task into subtasks is also called a threshold. It is up to each task to decide on a sensible threshold. It depends very much on the kind of work being done.

**Join**

When a task has split itself up into subtasks, the task waits until the subtasks have finished executing.

Once the subtasks have finished executing, the task may *join* (merge) all the results into one result. This is illustrated in the diagram below:



Of course, not all types of tasks may return a result. If the tasks do not return a result then a task just waits for its subtasks to complete. No result merging takes place then.

**The ForkJoinPool**

The ForkJoinPool is a special thread pool which is designed to work well with fork-and-join task splitting. The ForkJoinPool located in the java.util.concurrent package, so the full class name isjava.util.concurrent.ForkJoinPool.

**Creating a ForkJoinPool**

You create a ForkJoinPool using its constructor. As a parameter to the ForkJoinPool constructor you pass the indicated level of parallelism you desire. The parallelism level indicates how many threads or CPUs you want to work concurrently on on tasks passed to the ForkJoinPool. Here is a ForkJoinPool creation example:

ForkJoinPool forkJoinPool = new ForkJoinPool(4);

This example creates a ForkJoinPool with a parallelism level of 4.

**Submitting Tasks to the ForkJoinPool**

You submit tasks to a ForkJoinPool similarly to how you submit tasks to an ExecutorService. You can submit two types of tasks. A task that does not return any result (an "action"), and a task which does return a result (a "task"). These two types of tasks are represented by the RecursiveAction andRecursiveTask classes. How to use both of these tasks and how to submit them will be covered in the following sections.

**RecursiveAction**

A RecursiveAction is a task which does not return any value. It just does some work, e.g. writing data to disk, and then exits.

A RecursiveAction may still need to break up its work into smaller chunks which can be executed by independent threads or CPUs.

You implement a RecursiveAction by subclassing it. Here is a RecursiveAction example:

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveAction;

public class MyRecursiveAction extends RecursiveAction {

private long workLoad = 0;

public MyRecursiveAction(long workLoad) {

this.workLoad = workLoad;

}

@Override

protected void compute() {

//if work is above threshold, break tasks up into smaller tasks

if(this.workLoad > 16) {

System.out.println("Splitting workLoad : " + this.workLoad);

List<MyRecursiveAction> subtasks =

new ArrayList<MyRecursiveAction>();

subtasks.addAll(createSubtasks());

for(RecursiveAction subtask : subtasks){

subtask.fork();

}

} else {

System.out.println("Doing workLoad myself: " + this.workLoad);

}

}

private List<MyRecursiveAction> createSubtasks() {

List<MyRecursiveAction> subtasks =

new ArrayList<MyRecursiveAction>();

MyRecursiveAction subtask1 = new MyRecursiveAction(this.workLoad / 2);

MyRecursiveAction subtask2 = new MyRecursiveAction(this.workLoad / 2);

subtasks.add(subtask1);

subtasks.add(subtask2);

return subtasks;

}

}

This example is very simplified. The MyRecursiveAction simply takes a fictive workLoad as parameter to its constructor. If the workLoad is above a certain threshold, the work is split into subtasks which are also scheduled for execution (via the .fork() method of the subtasks. If the workLoad is below a certain threshold then the work is carried out by the MyRecursiveAction itself.

You can schedule a MyRecursiveAction for execution like this:

MyRecursiveAction myRecursiveAction = new MyRecursiveAction(24);

forkJoinPool.invoke(myRecursiveAction);

**RecursiveTask**

A RecursiveTask is a task that returns a result. It may split its work up into smaller tasks, and merge the result of these smaller tasks into a collective result. The splitting and merging may take place on several levels. Here is a RecursiveTask example:

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveTask;

public class MyRecursiveTask extends RecursiveTask<Long> {

private long workLoad = 0;

public MyRecursiveTask(long workLoad) {

this.workLoad = workLoad;

}

protected Long compute() {

//if work is above threshold, break tasks up into smaller tasks

if(this.workLoad > 16) {

System.out.println("Splitting workLoad : " + this.workLoad);

List<MyRecursiveTask> subtasks =

new ArrayList<MyRecursiveTask>();

subtasks.addAll(createSubtasks());

for(MyRecursiveTask subtask : subtasks){

subtask.fork();

}

long result = 0;

for(MyRecursiveTask subtask : subtasks) {

result += subtask.join();

}

return result;

} else {

System.out.println("Doing workLoad myself: " + this.workLoad);

return workLoad \* 3;

}

}

private List<MyRecursiveTask> createSubtasks() {

List<MyRecursiveTask> subtasks =

new ArrayList<MyRecursiveTask>();

MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad / 2);

MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad / 2);

subtasks.add(subtask1);

subtasks.add(subtask2);

return subtasks;

}

}

This example is similar to the RecursiveAction example except it returns a result. The classMyRecursiveTask extends RecursiveTask<Long> which means that the result returned from the task is a Long.

The MyRecursiveTask example also breaks the work down into subtasks, and schedules these subtasks for execution using their fork() method.

Additionally, this example then receives the result returned by each subtask by calling the join() method of each subtask. The subtask results are merged into a bigger result which is then returned. This kind of joining / mergining of subtask results may occur recursively for several levels of recursion.

You can schedule a RecursiveTask like this:

MyRecursiveTask myRecursiveTask = new MyRecursiveTask(128);

long mergedResult = forkJoinPool.invoke(myRecursiveTask);

System.out.println("mergedResult = " + mergedResult);

Notice how you get the final result out from the ForkJoinPool.invoke() method call.

**ForkJoinPool Critique**

It seems not everyone is equally happy with the new ForkJoinPool in Java 7. While searching for experiences with, and opinions about, the ForkJoinPool, I came across the following critique:

[**A Java Fork-Join Calamity**](http://coopsoft.com/ar/CalamityArticle.html)

It is well worth a read before you plan to use the ForkJoinPool in your own projects.

**AtomicStampedReference**

* [Creating an AtomicStampedReference](http://tutorials.jenkov.com/java-util-concurrent/atomicstampedreference.html#creating-an-atomicstampedreference)
  + [Creating a Typed AtomicStampedReference](http://tutorials.jenkov.com/java-util-concurrent/atomicstampedreference.html#creating-a-typed-atomicstampedreference)
* [Getting the AtomicStampedReference Reference](http://tutorials.jenkov.com/java-util-concurrent/atomicstampedreference.html#getting-the-atomicstampedreference-reference)
* [Getting the AtomicStampedReference Stamp](http://tutorials.jenkov.com/java-util-concurrent/atomicstampedreference.html#getting-the-atomicstampedreference-stamp)
* [Getting Reference and Stamp Atomically](http://tutorials.jenkov.com/java-util-concurrent/atomicstampedreference.html#getting-reference-and-stamp-atomically)
* [Setting the AtomicStampedReference Reference](http://tutorials.jenkov.com/java-util-concurrent/atomicstampedreference.html#setting-the-atomicstampedreference-reference)
* [Comparing and Setting the AtomicStampedReference Reference](http://tutorials.jenkov.com/java-util-concurrent/atomicstampedreference.html#comparing-and-setting-the-AtomicStampedReference-reference)
* [AtomicStampedReference and the A-B-A Problem](http://tutorials.jenkov.com/java-util-concurrent/atomicstampedreference.html#atomicstampedreference-and-the-a-b-a-problem)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2015-03-26 |

The AtomicStampedReference class provides an object reference variable which can be read and written atomically. By atomic is meant that multiple threads attempting to change the same AtomicStampedReference will not make the AtomicStampedReference end up in an inconsistent state.

The AtomicStampedReference is different from the [**AtomicReference**](http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html) in that the AtomicStampedReferencekeeps both an object reference and a stamp internally. The reference and stamp can be swapped using a single atomic compare-and-swap operation, via the compareAndSet() method.

The AtomicStampedReference is designed to be able to solve the A-B-A problem which is not possible to solve with an AtomicReference alone. The A-B-A problem is explained later in this text.

**Creating an AtomicStampedReference**

You can create an AtomicStampedReference instance like this:

Object initialRef = null;

int initialStamp = 0;

AtomicStampedReference atomicStampedReference =

new AtomicStampedReference(intialRef, initialStamp);

**Creating a Typed AtomicStampedReference**

You can use [**Java generics**](http://tutorials.jenkov.com/java-generics/index.html) to create a typed AtomicStampedReference. Here is a typedAtomicStampedReference example:

String initialRef = null;

int initialStamp = 0;

AtomicStampedReference<String> atomicStampedStringReference =

new AtomicStampedReference<String>(

initialRef, initialStamp

);

**Getting the AtomicStampedReference Reference**

You can get the reference stored in an AtomicStampedReference using the AtomicStampedReference'sgetReference() method. If you have an untyped AtomicStampedReference then the getReference() method returns an Object reference. If you have a typed AtomicStampedReference then getReference() returns a reference to the type you declared on the AtomicStampedReference variable when you created it.

Here is first an untyped AtomicStampedReference getReference() example:

String initialRef = "first text";

AtomicStampedReference atomicStampedReference =

new AtomicStampedReference(initialRef, 0);

String reference = (String) atomicStampedReference.getReference();

Notice how it is necessary to cast the reference returned by getReference() to a String becausegetReference() returns an Object reference when the AtomicStampedReference is untyped.

Here is a typed AtomicStampedReference example:

String initialRef = "first text";

AtomicStampedReference<String> atomicStampedReference =

new AtomicStampedReference<String>(

initialRef, 0

);

String reference = atomicStampedReference.getReference();

Notice how it is no longer necessary to cast the referenced returned by getReference() because the compiler knows it will return a String reference.

**Getting the AtomicStampedReference Stamp**

The AtomicStampedReference also contains a getStamp() method which can be used to obtain the internally stored stamp. Here is a getStamp() example:

String initialRef = "first text";

AtomicStampedReference atomicStampedReference =

new AtomicStampedReference(initialRef, 0);

int stamp = atomicStampedReference.getStamp();

**Getting Reference and Stamp Atomically**

You can obtain both reference and stamp from an AtomicStampedReference in a single, atomic operation using the get() method. The get() method returns the reference as return value from the method. The stamp is inserted into an int[] array that is passed as parameter to the get() method. Here is a get()example:

String initialRef = "text";

String initialStamp = 0;

AtomicStampedReference atomicStampedReference =

new AtomicStampedReference(

initialRef, initialStamp

);

int[] stampHolder = new int[1];

Object ref = atomicStampedReference.get(stampHolder);

System.out.println("ref = " + ref);

System.out.println("stamp = " + stampHolder[0]);

Being able to obtain both reference and stamp as a single atomic operation is important for some types of concurrent algorithms.

**Setting the AtomicStampedReference Reference**

You can set the reference stored in an AtomicStampedReference instance using its set() method. In an untyped AtomicStampedReference instance the set() method takes an Object reference as first parameter. In a typed AtomicStampedReference the set() method takes whatever type as parameter you declared as its type when you declared the AtomicStampedReference.

Here is an AtomicStampedReference set() example:

AtomicStampedReference atomicStampedReference =

new AtomicStampedReference(null, 0);

String newRef = "New object referenced";

int newStamp = 1;

atomicStampedReference.set(newRef, newStamp);

There is no difference to see in the use of the set() method for an untyped or typed reference. The only real difference you will experience is that the compiler will restrict the types you can set on a typedAtomicStampedReference.

**Comparing and Setting the AtomicStampedReference Reference**

The AtomicStampedReference class contains a useful method named compareAndSet(). ThecompareAndSet() method can compare the reference stored in the AtomicStampedReference instance with an expected reference, and the stored stamp with an expected stamp, and if they two references and stamps are the same (not equal as in equals() but same as in ==), then a new reference can be set on the AtomicStampedReference instance.

If compareAndSet() sets a new reference in the AtomicStampedReference the compareAndSet() method returns true. Otherwise compareAndSet() returns false.

Here is an AtomicStampedReference compareAndSet() example:

String initialRef = "initial value referenced";

int initialStamp = 0;

AtomicStampedReference<String> atomicStringReference =

new AtomicStampedReference<String>(

initialRef, initialStamp

);

String newRef = "new value referenced";

int newStamp = initialStamp + 1;

boolean exchanged = atomicStringReference

.compareAndSet(initialRef, newRef, initialStamp, newStamp);

System.out.println("exchanged: " + exchanged); //true

exchanged = atomicStringReference

.compareAndSet(initialRef, "new string", newStamp, newStamp + 1);

System.out.println("exchanged: " + exchanged); //false

exchanged = atomicStringReference

.compareAndSet(newRef, "new string", initialStamp, newStamp + 1);

System.out.println("exchanged: " + exchanged); //false

exchanged = atomicStringReference

.compareAndSet(newRef, "new string", newStamp, newStamp + 1);

System.out.println("exchanged: " + exchanged); //true

This example first creates an AtomicStampedReference and then uses compareAndSet() to swap the reference and stamp.

After the first compareAndSet() call the example attempts to swap the reference and stamp two times without success. The first time the initialRef is passed as expected reference, but the internally stored reference is newRef at this time, so the compareAndSet() call fails. The second time the initialStamp is passed as the expected stamp, but the internally stored stamp is newStamp at this time, so the compareAndSet() call fails.

The final compareAndSet() call will succeed, because the expected reference is newRef and the expected stamp is newStamp.

**AtomicStampedReference and the A-B-A Problem**

The AtomicStampedReference is designed to solve the A-B-A problem. The A-B-A problem is when a reference is changed from pointing to A, then to B and then back to A.

When using compare-and-swap operations to change a reference atomically, and making sure that only one thread can change the reference from an old reference to a new, detecting the A-B-A situation is impossible.

The A-B-A problem can occur in [**non-blocking algorithms**](http://tutorials.jenkov.com/java-util-concurrent/non-blocking-algorithms.html). Non-blocking algorithms often use a reference to an ongoing modification to the guarded data structure, to signal to other threads that a modification is currently ongoing. If thread 1 sees that there is no ongoing modification (reference points to null), another thread may submit a modification (reference is now non-null), complete the modification and swap the reference back to null without thread 1 detecting it. Exactly how the A-B-A problem occurs in non-blocking algorithsm is explained in more detail in my tutorial about [**non-blocking algorithms**](http://tutorials.jenkov.com/java-util-concurrent/non-blocking-algorithms.html).

By using an AtomicStampedReference instead of an AtomicReference it is possible to detect the A-B-A situation. Thread 1 can copy the reference and stamp out of the AtomicStampedReference atomically using get(). If another thread changes the reference from A to B and then back to A, then the stamp will have changed (provided threads update the stamp sensibly - e.g increment it).

The code below shows how to detect the A-B-A situation using the AtomicStampedReference:

int[] stampHolder = new int[1];

Object ref = atomicStampedReference.get(stampHolder);

if(ref == null){

//prepare optimistic modification

}

//if another thread changes the reference and stamp here,

//it can be detected

int[] stampHolder2 = new int[1];

Object ref2 = atomicStampedReference.get(stampHolder);

if(ref == ref2 && stampHolder[0] == stampHolder2[0]){

//no modification since optimistic modification was prepared

} else {

//retry from scratch.

}

**AtomicReference**

* [Creating an AtomicReference](http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html#creating-an-atomicreference)
  + [Creating a Typed AtomicReference](http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html#creating-a-typed-atomicreference)
* [Getting the AtomicReference Reference](http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html#getting-the-atomicreference-reference)
* [Setting the AtomicReference Reference](http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html#setting-the-atomicreference-reference)
* [Comparing and Setting the AtomicReference Reference](http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html#comparing-and-setting-the-atomicreference-reference)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2016-01-26 |

The AtomicReference class provides an object reference variable which can be read and written atomically. By atomic is meant that multiple threads attempting to change the same AtomicReference (e.g. with a compare-and-swap operation) will not make the AtomicReference end up in an inconsistent state. AtomicReference even has an advanced compareAndSet() method which lets you compare the reference to an expected value (reference) and if they are equal, set a new reference inside the AtomicReferenceobject.

**Creating an AtomicReference**

You can create an AtomicReference instance like this:

AtomicReference atomicReference = new AtomicReference();

If you need to create the AtomicReference with an initial reference, you can do so like this:

String initialReference = "the initially referenced string";

AtomicReference atomicReference = new AtomicReference(initialReference);

**Creating a Typed AtomicReference**

You can use Java generics to create a typed AtomicReference. Here is a typed AtomicReference example:

AtomicReference<String> atomicStringReference =

new AtomicReference<String>();

You can also set an initial value for a typed AtomicReference. Here is a typed AtomicReference instantiation example with an initial value:

String initialReference = "the initially referenced string";

AtomicReference<String> atomicStringReference =

new AtomicReference<String>(initialReference);

**Getting the AtomicReference Reference**

You can get the reference stored in an AtomicReference using the AtomicReference's get() method. If you have an untyped AtomicReference then the get() method returns an Object reference. If you have a typed AtomicReference then get() returns a reference to the type you declared on the AtomicReference variable when you created it.

Here is first an untyped AtomicReference get() example:

AtomicReference atomicReference = new AtomicReference("first value referenced");

String reference = (String) atomicReference.get();

Notice how it is necessary to cast the reference returned by get() to a String because get() returns an Object reference when the AtomicReference is untyped.

Here is a typed AtomicReference example:

AtomicReference<String> atomicReference =

new AtomicReference<String>("first value referenced");

String reference = atomicReference.get();

Notice how it is no longer necessary to cast the referenced returned by get() because the compiler knows it will return a String reference.

**Setting the AtomicReference Reference**

You can set the reference stored in an AtomicReference instance using its set() method. In an untyped AtomicReference instance the set() method takes an Object reference as parameter. In a typed AtomicReference the set() method takes whatever type as parameter you declared as its type when you declared the AtomicReference.

Here is an AtomicReference set() example:

AtomicReference atomicReference =

new AtomicReference();

atomicReference.set("New object referenced");

There is no difference to see in the use of the set() method for an untyped or typed reference. The only real difference you will experience is that the compiler will restrict the types you can set on a typedAtomicReference.

**Comparing and Setting the AtomicReference Reference**

The AtomicReference class contains a useful method named compareAndSet(). The compareAndSet()method can compare the reference stored in the AtomicReference instance with an expected reference, and if they two references are the same (not equal as in equals() but same as in ==), then a new reference can be set on the AtomicReference instance.

If compareAndSet() sets a new reference in the AtomicReference the compareAndSet() method returns true. Otherwise compareAndSet() returns false.

Here is an AtomicReference compareAndSet() example:

String initialReference = "initial value referenced";

AtomicReference<String> atomicStringReference =

new AtomicReference<String>(initialReference);

String newReference = "new value referenced";

boolean exchanged = atomicStringReference.compareAndSet(initialReference, newReference);

System.out.println("exchanged: " + exchanged);

exchanged = atomicStringReference.compareAndSet(initialReference, newReference);

System.out.println("exchanged: " + exchanged);

This example creates a typed AtomicReference with an initial reference. Then it calls comparesAndSet() two times to compare the stored reference to the initial reference, and set a new reference if the stored reference is equal to the initial reference. The first time the two references are the same, so a new reference is set on the AtomicReference. The second time the stored reference is the new reference just set in the call to compareAndSet() before, so the stored reference is of course not equal to the initial reference. Thus, a new reference is not set on the AtomicReference and the compareAndSet() method returns false.

**AtomicBoolean**

* [Creating an AtomicBoolean](http://tutorials.jenkov.com/java-util-concurrent/atomicboolean.html#creating-an-atomicboolean)
* [Getting the AtomicBoolean's Value](http://tutorials.jenkov.com/java-util-concurrent/atomicboolean.html#getting-the-atomicboolean-value)
* [Setting the AtomicBoolean's Value](http://tutorials.jenkov.com/java-util-concurrent/atomicboolean.html#setting-the-atomicboolean-value)
* [Swapping the AtomicBoolean's Value](http://tutorials.jenkov.com/java-util-concurrent/atomicboolean.html#swapping-the-atomicboolean-value)
* [Compare and Set AtomicBoolean's Value](http://tutorials.jenkov.com/java-util-concurrent/atomicboolean.html#compare-and-set-atomicboolean-value)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-10-06 |

The AtomicBoolean class provides you with a boolean variable which can be read and written atomically, and which also contains advanced atomic operations like compareAndSet(). The AtomicBoolean class is located in the java.util.concurrent.atomic package, so the full class name isjava.util.concurrent.atomic.AtomicBoolean . This text describes the version of AtomicBoolean found in Java 8, but the first version was added in Java 5.

The reasoning behind the AtomicBoolean design is explained in my Java Concurrency tutorial in the text about [**Compare and Swap**](http://tutorials.jenkov.com/java-concurrency/compare-and-swap.html).

**Creating an AtomicBoolean**

You create an AtomicBoolean like this:

AtomicBoolean atomicBoolean = new AtomicBoolean();

This example creates a new AtomicBoolean with the value false;

If you need to set an explicit initial value for the AtomicBoolean instance, you can pass the initial value to the AtomicBoolean constructor like this:

AtomicBoolean atomicBoolean = new AtomicBoolean(**true**);

**Getting the AtomicBoolean's Value**

You can get the value of an AtomicBoolean using the get() method. Here is an example:

AtomicBoolean atomicBoolean = new AtomicBoolean(true);

boolean value = atomicBoolean.get();

After executing this code the value variable will contain the value true.

**Setting the AtomicBoolean's Value**

You can set the value of an AtomicBoolean using the set() method. Here is an example:

AtomicBoolean atomicBoolean = new AtomicBoolean(true);

atomicBoolean.set(false);

After executing this code the AtomicBoolean variable will contain the value false.

**Swapping the AtomicBoolean's Value**

You can swap the value of an AtomicBoolean using the getAndSet() method. The getAndSet() method returns the AtomicBoolean's current value, and sets a new value for it. Here is an example:

AtomicBoolean atomicBoolean = new AtomicBoolean(true);

boolean oldValue = atomicBoolean.getAndSet(false);

After executing this code the oldValue variable will contain the value true, and the AtomicBoolean instance will contain the value false. The code effectively swaps the value false for the AtomicBoolean's current value which is true.

**Compare and Set AtomicBoolean's Value**

The method compareAndSet() allows you to compare the current value of the AtomicBoolean to an expected value, and if current value is equal to the expected value, a new value can be set on the AtomicBoolean. The compareAndSet() method is atomic, so only a single thread can execute it at the same time. Thus, the compareAndSet() method can be used to implemented simple synchronizers like locks.

Here is a compareAndSet() example:

AtomicBoolean atomicBoolean = new AtomicBoolean(true);

boolean expectedValue = true;

boolean newValue = false;

boolean wasNewValueSet = atomicBoolean.compareAndSet(

expectedValue, newValue);

This example compares the current value of the AtomicBoolean to true and if the two values are equal, sets the new value of the AtomicBoolean to false .

**ScheduledThreadPoolExecutor – Task Scheduling with Executors**

March 25, 2015 by Lokesh Gupta

The Java Executor Framework provides the [ThreadPoolExecutor](http://howtodoinjava.com/2015/03/24/java-thread-pool-executor-example/)class to execute Callable and Runnable tasks with a pool of threads, which avoid you writing lots of boiler plate complex code. The way executors work is when you send a task to the executor, it’s executed as soon as possible. But there may be used cases when you are not interested in executing a task as soon as possible. Rather You may want to execute a task after a period of time or to execute a task periodically. For these purposes, the Executor framework provides the [ScheduledThreadPoolExecutor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html)class.

**Task to be executed**

Let’s write a very basic task which we can use for demo purpose.

|  |
| --- |
| class Task implements Runnable  {      private String name;        public Task(String name) {          this.name = name;      }        public String getName() {          return name;      }        @Override      public void run()      {          try {              System.out.println("Doing a task during : " + name + " - Time - " + new Date());          }          catch (Exception e) {              e.printStackTrace();          }      }  } |

**Execute a task after a period of time**

|  |
| --- |
| package com.howtodoinjava.demo.multithreading;    import java.util.Date;  import java.util.concurrent.Executors;  import java.util.concurrent.ScheduledExecutorService;  import java.util.concurrent.TimeUnit;    public class ScheduledThreadPoolExecutorExample  {      public static void main(String[] args)      {          ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);          Task task1 = new Task ("Demo Task 1");          Task task2 = new Task ("Demo Task 2");            System.out.println("The time is : " + new Date());            executor.schedule(task1, 5 , TimeUnit.SECONDS);          executor.schedule(task2, 10 , TimeUnit.SECONDS);            try {                executor.awaitTermination(1, TimeUnit.DAYS);          } catch (InterruptedException e) {                e.printStackTrace();          }            executor.shutdown();      }  }    Output:    The time is : Wed Mar 25 16:14:07 IST 2015  Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:14:12 IST 2015  Doing a task during : Demo Task 2 - Time - Wed Mar 25 16:14:17 IST 2015 |

As with class ThreadPoolExecutor, to create a scheduled executor, Java recommends the utilization of the Executorsclass. In this case, you have to use the newScheduledThreadPool() method. You have passed the number 1 as a parameter to this method. This parameter is the number of threads you want to have in the pool.

To execute a task in this scheduled executor after a period of time, you have to use the schedule() method. This method receives the following three parameters:

* The task you want to execute
* The period of time you want the task to wait before its execution
* The unit of the period of time, specified as a constant of the TimeUnit class

Also note that You can also use the Runnable interface to implement the tasks, because the schedule() method of the ScheduledThreadPoolExecutor class accepts both types of tasks.

Moreover ,although the ScheduledThreadPoolExecutor class is a child class of the ThreadPoolExecutor class and, therefore, inherits all its features, Java recommends the utilization of ScheduledThreadPoolExecutor only for scheduled tasks.

Finally, you can configure the behavior of the ScheduledThreadPoolExecutor class when you call the shutdown()method and there are pending tasks waiting for the end of their delay time. The default behavior is that those tasks will be executed despite the finalization of the executor. You can change this behavior using the setExecuteExistingDelayedTasksAfterShutdownPolicy() method of the ScheduledThreadPoolExecutor class. With false, at the time of shutdown(), pending tasks won’t get executed.

**Execute a task periodically**

Now let’s learn how to use ScheduledThreadPoolExecutor to schedule a periodic task.

|  |
| --- |
| public class ScheduledThreadPoolExecutorExample  {      public static void main(String[] args)      {          ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);          Task task1 = new Task ("Demo Task 1");            System.out.println("The time is : " + new Date());            ScheduledFuture<?> result = executor.scheduleAtFixedRate(task1, 2, 5, TimeUnit.SECONDS);            try {              TimeUnit.MILLISECONDS.sleep(20000);          }          catch (InterruptedException e) {              e.printStackTrace();          }            executor.shutdown();      }  }    Output:    The time is : Wed Mar 25 16:20:12 IST 2015  Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:20:14 IST 2015  Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:20:19 IST 2015  Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:20:24 IST 2015  Doing a task during : Demo Task 1 - Time - Wed Mar 25 16:20:29 IST 2015 |

In this example, we have created ScheduledExecutorService instance just like above example using newScheduledThreadPool() method. Then we have used the scheduledAtFixedRate() method. This method accepts four parameters:

* the task you want to execute periodically,
* the delay of time until the first execution of the task,
* the period between two executions,
* and the time unit of the second and third parameters.

An important point to consider is that the period between two executions is the period of time between these two executions that begins. If you have a periodic task that takes 5 seconds to execute and you put a period of 3 seconds, you will have two instances of the task executing at a time.

ScheduledThreadPoolExecutor provides other methods to schedule periodic tasks. It is the scheduleWithFixedRate() method. It has the same parameters as the scheduledAtFixedRate() method, but there is a difference worth noticing. In the scheduledAtFixedRate() method, the third parameter determines the period of time between the starting of two executions. In the scheduledWithFixedRate() method, parameter determines the period of time between the end of an execution of the task and the beginning of the next execution.

You can also configure the behavior of an instance of the ScheduledThreadPoolExecutor class with the shutdown()method. The default behavior is that the scheduled tasks finish when you call that method. You can change this behavior using the setContinueExistingPeriodicTasksAfterShutdownPolicy() method of the ScheduledThreadPoolExecutor class with a true value. The periodic tasks won’t finish upon calling the shutdown()method.