java.util.concurrent
Java Concurrency Utilities

http://tutorials.jenkov.com/java-util-concurrent/index.html

1. java.util.concurrent - Java Concurrency Utilities

Java 5 added a new Java package to the Java platform, the java.util.concurrent package. This package contains a set of classes that makes it easier to develop concurrent (multithreaded) applications in Java. Before this package was added, you would have to program your utility classes yourself.
In this tutorial I will take you through the new java.util.concurrent classes, one by one, so you can learn how to use them. I will use the versions in Java 6. I am not sure if there are any differences to the Java 5 versions.
I will not explain the core issues of concurrency in Java - the theory behind it, that is. If you are interested in that, check out my Java Concurrency tutorial.

Work in Progress

This tutorial is very much "work in progress", so if you spot a missing class or interface, please be patient. I will add it when I get the time to do it.

Table of Contents

Here is a list of the topics covered in this java.util.concurrent trail. This list (menu) is also present at the top right of every page in the trail.

Feel Free to Contact Me

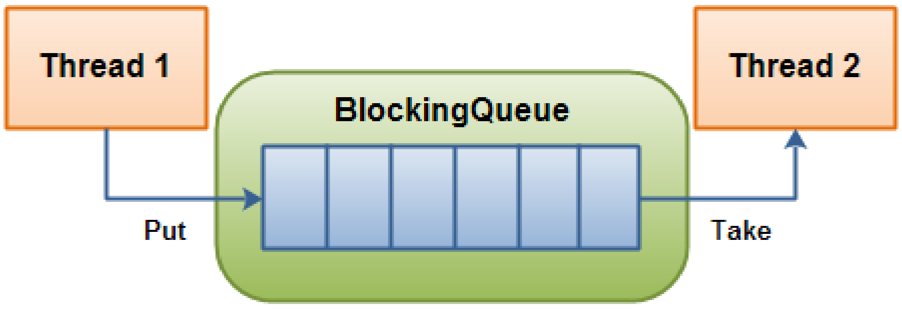
If you disagree with anything I write here about the java.util.concurrent utilities, or just have comments, questions, etc, feel free to send me an email. You wouldn't be the first to do so. You can find my email address on the about page.

2. BlockingQueue

The Java BlockingQueue interface in the java.util.concurrent class 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 in my more theoretical Java Concurrency Tutorial.

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.
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(o)	take(o)	poll(timeout, timeunit)
Examine	element(o)	peek(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).

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
- DelayQueue
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue

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();
            }
        }
}
```

3. ArrayBlockingQueue

The ArrayBlockingQueue class implements the BlockingQueue interface. Read the BlockingQueue text for more information about the interface.
ArrayBlockingQueue is a bounded, blocking queue that stores the elements internally in an array. That it is bounded means that it cannot store unlimited amounts of elements. There is an upper bound on the number of elements it can store at the same time. You set the upper bound at instantiation time, and after that it cannot be changed.
The ArrayBlockingQueue 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 an ArrayBlockingQueue:

```
BlockingQueue queue = new ArrayBlockingQueue(1024);

queue.put("1");

Object object = queue.take();
```

Here is a BlockingQueue example that uses Java Generics. Notice how you can put and take String's instead of :

```
BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1024);

queue.put("1");

String string = queue.take();
```

4. DelayQueue

DelayQueue class implements the BlockingQueue interface. Read the BlockingQueue text for more information about the interface.

The DelayQueue keeps 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.

5. LinkedBlockingQueue

The LinkedBlockingQueue class implements the BlockingQueue interface. Read the BlockingQueue 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_VALUE is 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();
```

6. PriorityBlockingQueue

The PriorityBlockingQueue class implements the BlockingQueue interface. Read the BlockingQueue text for more information about the interface.

The PriorityBlockingQueue is an unbounded concurrent queue. It uses the same ordering rules as the java.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 Comparable implementation.

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();
```

7. SynchronousQueue

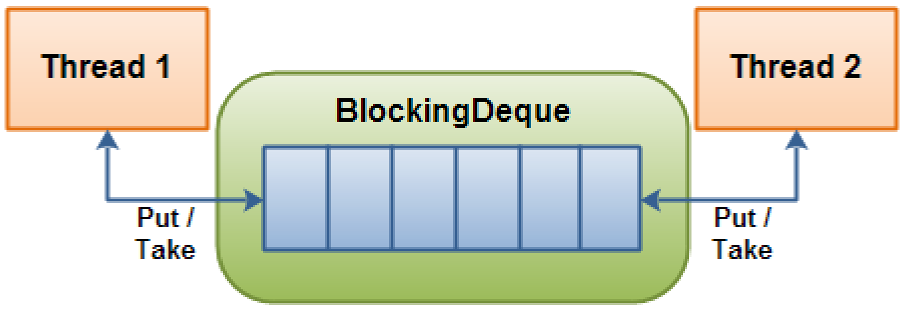
The SynchronousQueue class implements the BlockingQueue interface. Read the BlockingQueue text for more information about the interface.
The SynchronousQueue is a queue that can only contain a single element internally. A thread inseting an element into the queue is blocked until another thread takes 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.

8. BlockingDeque

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 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

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();
```

9. LinkedBlockingDeque

The LinkedBlockingDeque class implements the BlockingDeque interface. Read the BlockingDeque 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();
```

10. ConcurrentMap

java.util.concurrent.ConcurrentMap

The java.util.concurrent.ConcurrentMap interface represents a Map which is capable of handling concurrent access (puts and gets) to it.
The ConcurrentMap has a few extra atomic methods in addition to the methods it inherits from its superinterface, java.util.Map.

ConcurrentMap Implementations

Since ConcurrentMap is an interface, you need to use one of its implementations in order to use it. The java.util.concurrent package contains the following implementations of the ConcurrentMap interface:

- ConcurrentHashMap

ConcurrentHashMap

The ConcurrentHashMap is very similar to the java.util.HashTable class, except that ConcurrentHashMap offers better concurrency than HashTable does. ConcurrentHashMap does not lock the Map while you are reading from it. Additionally, ConcurrentHashMap does not lock the entire Map when writing to it. It only locks the part of the Map that is being written to, internally.
Another difference is that ConcurrentHashMap does not throw ConcurrentModificationException if the ConcurrentHashMap is changed while being iterated. The Iterator is not designed to be used by more than one thread though.
Checkout the official JavaDoc for more details about ConcurrentMap and ConcurrentHashMap.

ConcurrentMap Example

Here is an example of how to use the ConcurrentMap interface. The example uses a ConcurrentHashMap implementation:

```
ConcurrentMap concurrentMap = new ConcurrentHashMap();

concurrentMap.put("key", "value");

Object value = concurrentMap.get("key");
```

11. ConcurrentNavigableMap

The java.util.concurrent.ConcurrentNavigableMap class is a java.util.NavigableMap with support for concurrent access, and which has concurrent access enabled for its submaps. The "submaps" are the maps returned by various methods like headMap(), subMap() and tailMap().
Rather than re-explain all methods found in the NavigableMap I will just look at the methods added by ConcurrentNavigableMap.

headMap()

The headMap(T toKey) method returns a view of the map containing the keys which are strictly less than the given key.
If you make changes to the original map, these changes are reflected in the head map.
Here is an example illustrating the use of the headMap() method.

```
ConcurrentNavigableMap map = new ConcurrentSkipListMap();

map.put("1", "one");
map.put("2", "two");
map.put("3", "three");

ConcurrentNavigableMap headMap = map.headMap("2");
```

The headMap will point to a ConcurrentNavigableMap which only contains the key "1", since only this key is strictly less than "2".
See the JavaDoc for more specific details of how this method works, and how its overloaded versions work.

tailMap()

The tailMap(T fromKey) method returns a view of the map containing the keys which are greater than or equal to the given fromKey.
If you make changes to the original map, these changes are reflected in the tail map.
Here is an example illustrating the use of the tailMap() method:

```
ConcurrentNavigableMap map = new ConcurrentSkipListMap();

map.put("1", "one");
map.put("2", "two");
map.put("3", "three");

ConcurrentNavigableMap tailMap = map.tailMap("2");
```

The tailMap will contain the keys "2" and "3" because these two keys are greather than or equal to the given key, "2".
See the JavaDoc for more specific details of how this method works, and how its overloaded versions work.

subMap()

The subMap() method returns a view of the original map which contains all keys from (including), to (excluding) two keys given as parameters to the method. Here is an example:

```
ConcurrentNavigableMap map = new ConcurrentSkipListMap();

map.put("1", "one");
map.put("2", "two");
map.put("3", "three");

ConcurrentNavigableMap subMap = map.subMap("2", "3");
```

The returned submap contains only the key "2", because only this key is greater than or equal to "2", and smaller than "3".

More Methods

The ConcurrentNavigableMap interface contains a few more methods that might be of use. For instance:

- descendingKeySet()
- descendingMap()
- navigableKeySet()

See the official JavaDoc for more information on these methods.

12. CountDownLatch

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.

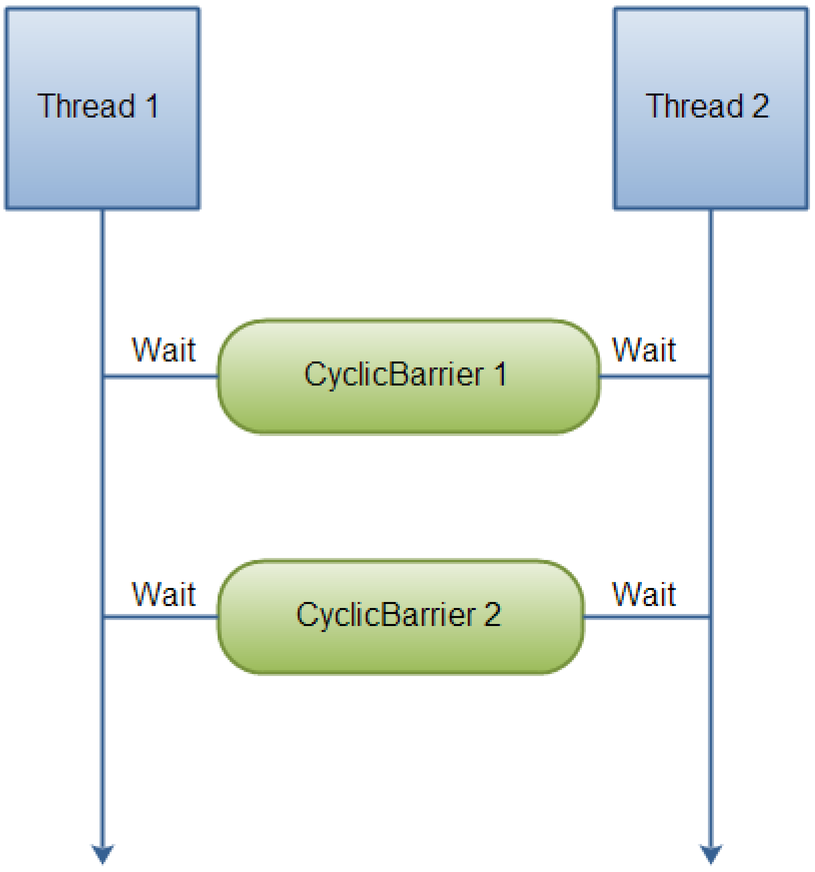
A CountDownLatch is initialized with a given count. This count is decremented by calls to the countDown() method. Threads waiting for this count to reach zero can call one of the await() methods. Calling await() blocks the thread until the count reaches zero.

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();
        }
    }
}
```

13. CyclicBarrier

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.

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.

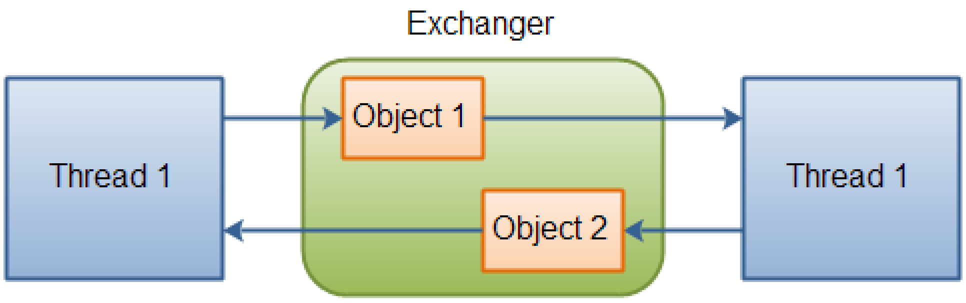
```
Thread-0 waiting at barrier 1
Thread-1 waiting at barrier 1
BarrierAction 1 executed
Thread-1 waiting at barrier 2
Thread-0 waiting at barrier 2
```

```
BarrierAction 2 executed
Thread-0 done!
Thread-1 done!
```

14. Exchanger

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.

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
```

15. Semaphore

The java.util.concurrent.Semaphore class is a counting semaphore. 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 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.

16. ExecutorService

The java.util.concurrent.ExecutorService interface represents an asynchronous execution mechanism which is capable of executing tasks in the background. An ExecutorService is thus very similar to a thread pool. In fact, the implementation of ExecutorService present in the java.util.concurrent package is a thread pool implementation.

ExecutorService Example

Here is a simple Java ExectorService example:

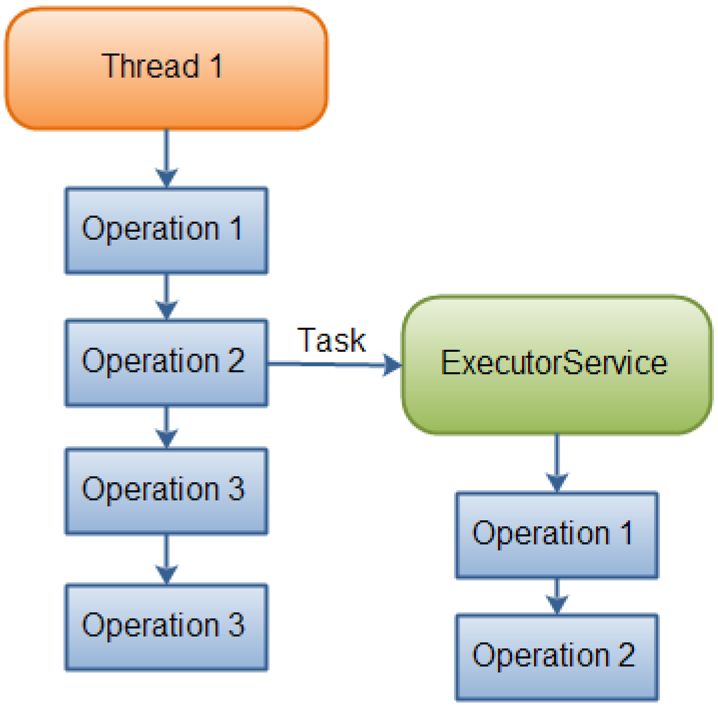
```
ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

First an ExecutorService is created using the newFixedThreadPool() factory method. This creates a thread pool with 10 threads executing tasks.

Second, an anonymous implementation of the Runnable interface is passed to the execute() method. This causes the Runnable to be executed by one of the threads in the ExecutorService.

Task Delegation

Here is a diagram illustrating a thread delegating a task to an ExecutorService for asynchronous execution:



A thread delegating a task to an ExecutorService for asynchronous execution.

Once the thread has delegated the task to the ExecutorService, the thread continues its own execution independent of the execution of that task.

ExecutorService Implementations

Since ExecutorService is an interface, you need to its implementations in order to make any use of it. The ExecutorService has the following implementation in the java.util.concurrent package:

- ThreadPoolExecutor
- ScheduledThreadPoolExecutor

Creating an ExecutorService

How you create an ExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ExecutorService instances too. Here are a few examples of creating an ExecutorService:

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);
```

ExecutorService Usage

There are a few different ways to delegate tasks for execution to an ExecutorService:

- execute(Runnable)

- submit(Runnable)
- submit(Callable)
- invokeAny(...)
- invokeAll(...)

I will take a look at each of these methods in the following sections.

execute(Runnable)

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. Here is an example of executing a Runnable with an ExecutorService:

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use a Callable for that (explained in the following sections).

submit(Runnable)

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Future object can be used to check if the Runnable as finished executing.

Here is a ExecutorService submit() example:

```
Future future = executorService.submit(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

future.get();    //returns null if the task has finished correctly.
```

submit(Callable)

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.

The Callable's result can be obtained via the Future object returned by the submit(Callable) method. Here is an ExecutorService Callable example:

```
Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

System.out.println("future.get() = " + future.get());
```

The above code example will output this:

```
Asynchronous Callable
future.get() = Callable Result
```

invokeAny()

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.

If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.

Here is a code example:

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 2";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 3";
    }
});

String result = executorService.invokeAny(callables);

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

executorService.shutdown();
```

This code example will print out the object returned by one of the Callable's in the given collection. I have tried running it a few times, and the result changes. Sometimes it is "Task 1", sometimes "Task 2" etc.

invokeAll()

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.
Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a Future to tell the difference.
Here is a code example:

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 2";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 3";
    }
});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){
    System.out.println("future.get = " + future.get());
}

executorService.shutdown();
```

ExecutorService Shutdown

When you are done using the ExecutorService you should shut it down, so the threads do not keep running.
For instance, if your application is started via a main() method and your main thread exits your

application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.
To terminate the threads inside the ExecutorService you call its shutdown() method. The ExecutorService will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down. All tasks submitted to the ExecutorService before shutdown() is called, are executed.
If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

17. ThreadPoolExecutor

The java.util.concurrent.ThreadPoolExecutor is an implementation of the ExecutorService interface. The ThreadPoolExecutor executes the given task (Callable or Runnable) using one of its internally pooled threads.

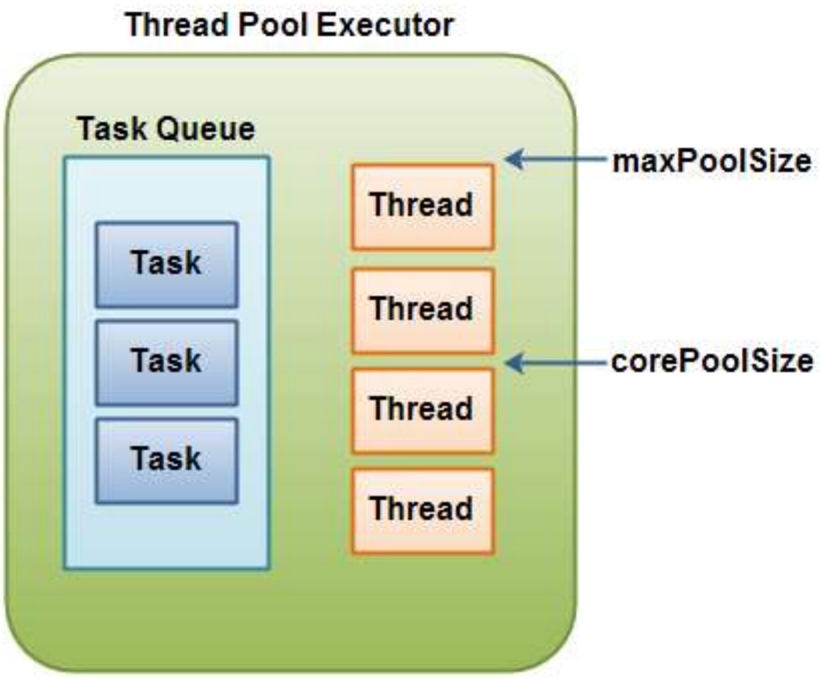
The thread pool contained inside the ThreadPoolExecutor can contain a varying amount of threads. The number of threads in the pool is determined by these variables:

- corePoolSize
- maximumPoolSize

If less than corePoolSize threads are created in the the thread pool when a task is delegated to the thread pool, then a new thread is created, even if idle threads exist in the pool.

If the internal queue of tasks is full, and corePoolSize threads or more are running, but less than maximumPoolSize threads are running, then a new thread is created to execute the task.

Here is a diagram illustrating the ThreadPoolExecutor principles:



A ThreadPoolExecutor

Creating a ThreadPoolExecutor

The ThreadPoolExecutor has several constructors available. For instance:

```
int    corePoolSize    =       5;
int    maxPoolSize     =      10;
long keepAliveTime = 5000;

ExecutorService threadPoolExecutor =
                new ThreadPoolExecutor(
                    corePoolSize,
                    maxPoolSize,
                    keepAliveTime,
                    TimeUnit.MILLISECONDS,
                    new LinkedBlockingQueue<Runnable>()
                    );
```

However, unless you need to specify all these parameters explicitly for your ThreadPoolExecutor, it is often easier to use one of the factory methods in the java.util.concurrent.Executors class, as shown in the ExecutorService text.

18. ScheduledExecutorService

The java.util.concurrent.ScheduledExecutorService is an ExecutorService 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 section for more information.

19. Java Fork and Join using ForkJoinPool

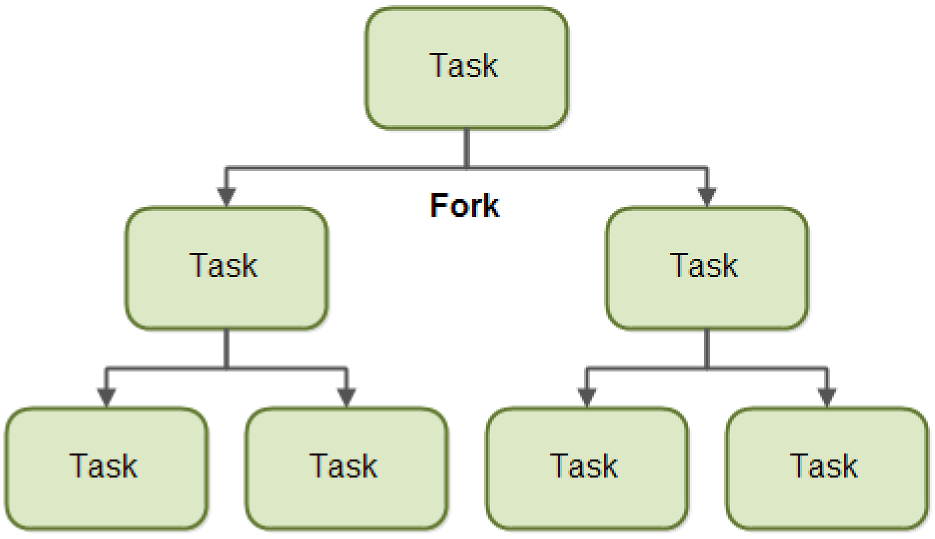
The ForkJoinPool was added to Java in Java 7. The ForkJoinPool is similar to the Java ExecutorService but with one difference. The ForkJoinPool makes it easy for tasks to split their work up into smaller tasks which are then submitted to the ForkJoinPool too. Tasks can keep splitting their work into 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:

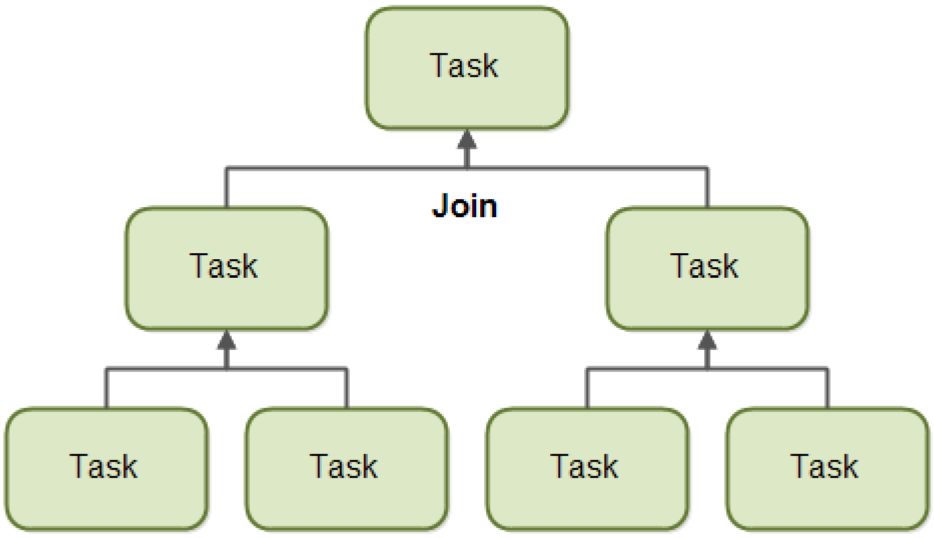


By splitting itself up into subtasks, each subtask can be executed in parallel by different CPUs, or different threads on the same CPU.
A task only splits itself up into subtasks if the work the task was given is large enough for this to make sense. There is an overhead to splitting up a task into subtasks, 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 is java.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 and RecursiveTask 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 class MyRecursiveTask 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

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

20. Lock

A java.util.concurrent.locks.Lock is a thread synchronization mechanism just like synchronized blocks. A Lock is, however, more flexible and more sophisticated than a synchronized block.
By the way, in my Java Concurrency tutorial I have described how to implement your own locks, in case you are interested (or need it). See my text on Locks for more details.

Java Lock Example

Since Lock is an interface, you need to use one of its implementations to use a Lock in your applications. Here is a simple usage example:

```
Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();
```

First a Lock is created. Then it's lock() method is called. Now the Lock instance is locked. Any other thread calling lock() will be blocked until the thread that locked the lock calls unlock(). Finally unlock() is called, and the Lock is now unlocked so other threads can lock it.

Java Lock Implementations

The java.util.concurrent.locks package has the following implementations of the Lock interface:

- ReentrantLock

Main Differences Between Locks and Synchronized Blocks

The main differences between a Lock and a synchronized block are:

- A synchronized block makes no guarantees about the sequence in which threads waiting to entering it are granted access.
- You cannot pass any parameters to the entry of a synchronized block. Thus, having a timeout trying to get access to a synchronized block is not possible.
- The synchronized block must be fully contained within a single method. A Lock can have it's calls to lock() and unlock() in separate methods.

Lock Methods

The Lock interface has the following primary methods:

- lock()

- lockInterruptibly()
- tryLock()
- tryLock(long timeout, TimeUnit timeUnit)
- unlock()

The lock() method locks the Lock instance if possible. If the Lock instance is already locked, the thread calling lock() is blocked until the Lock is unlocked.

The lockInterruptibly() method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.

The tryLock() method attempts to lock the Lock instance immediately. It returns true if the locking succeeds, false if Lock is already locked. This method never blocks.

The tryLock(long timeout, TimeUnit timeUnit) works like the tryLock() method, except it waits up the given timeout before giving up trying to lock the Lock.

The unlock() method unlocks the Lock instance. Typically, a Lock implementation will only allow the thread that has locked the Lock to call this method. Other threads calling this method may result in an unchecked exception (RuntimeException).

21. ReadWriteLock

A java.util.concurrent.locks.ReadWriteLock is an advanced thread lock mechanism. It allows multiple threads to read a certain resource, but only one to write it, at a time.

The idea is, that multiple threads can read from a shared resource without causing concurrency errors. The concurrency errors first occur when reads and writes to a shared resource occur concurrently, or if multiple writes take place concurrently.

In this text I only cover Java's built-in ReadWriteLock. If you want to read more about the theory behind the implemenation of a ReadWriteLock, you can read it in my text on Read Write Locks in my Java Concurrency tutorial.

ReadWriteLock Locking Rules

The rules by which a thread is allowed to lock the ReadWriteLock either for reading or writing the guarded resource, are as follows:

- **Read Lock** If no threads have locked the ReadWriteLock for writing, and no thread have requested a write lock (but not yet obtained it). Thus, multiple threads can lock the lock for reading.
- **Write Lock** If no threads are reading or writing. Thus, only one thread at a time can lock the lock for writing.

ReadWriteLock Implementations

ReadWriteLock is an interface. Thus, to use a ReadWriteLock

The java.util.concurrent.locks package contains the following ReadWriteLock implementation:

- ReentrantReadWriteLock

ReadWriteLock Code Example

Here is a simple code example that shows how to create a ReadWriteLock and how to lock it for reading and writing:

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();


readWriteLock.readLock().lock();

    // multiple readers can enter this section
    // if not locked for writing, and not writers waiting
    // to lock for writing.

readWriteLock.readLock().unlock();
readWriteLock.writeLock().lock();

        // only one writer can enter this section,
        // and only if no threads are currently reading.

readWriteLock.writeLock().unlock();
```

Notice how the ReadWriteLock actually internally keeps two Lock instances. One guarding read access, and one guarding write access.

22. AtomicBoolean

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 is java.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.

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 .

23. AtomicInteger

The AtomicInteger class provides you with a boolean variable which can be read and written atomically, and which also contains advanced atomic operations like compareAndSet(). The AtomicInteger class is located in the java.util.concurrent.atomic package, so the full class name is java.util.concurrent.atomic.AtomicInteger . This text describes the version of AtomicInteger found in Java 8, but the first version was added in Java 5.
The reasoning behind the AtomicInteger design is explained in my Java Concurrency tutorial in the text about Compare and Swap.

Creating an AtomicInteger

Creating an AtomicInteger is done like this:

```
AtomicInteger atomicInteger = new AtomicInteger();
```

This example creates an AtomicInteger with the initial value 0 .
If you want to create an AtomicInteger with an initial value, you can do so like this:

```
AtomicInteger atomicInteger = new AtomicInteger(123);
```

This example passes a value of 123 as parameter to the AtomicInteger contructor, which sets the initial value of the AtomicInteger instance to 123 .

Getting the AtomicInteger Value

You can get the value of an AtomicInteger instance via the get() method. Here is an AtomicInteger.get() example:

```
AtomicInteger atomicInteger = new AtomicInteger(123);

int theValue = atomicInteger.get();
```

Setting the AtomicInteger Value

You can set the value of an AtomicInteger instance via the set() method. Here is an AtomicInteger.set() example:

```
AtomicInteger atomicInteger = new AtomicInteger(123);

atomicInteger.set(234);
```

This example creates an AtomicInteger example with an initial value of 123, and then sets its value to 234 in the next line.

Compare and Set the AtomicInteger Value

The AtomicInteger class also has an atomic compareAndSet() method. This method compares the current value of the AtomicInteger instance to an expected value, and if the two values are equal, sets a new value for the AtomicInteger instance. Here is an AtomicInteger.compareAndSet()

example:

```
AtomicInteger atomicInteger = new AtomicInteger(123);

int expectedValue = 123;
int newValue         = 234;
atomicInteger.compareAndSet(expectedValue, newValue);
```

This example first creates an AtomicInteger instance with an initial value of 123 . Then it compares the value of the AtomicInteger to the expected value 123 and if they are equal the new value of the AtomicInteger becomes 234;

Adding to the AtomicInteger Value

The AtomicInteger class contains a few methods you can use to add a value to the AtomicInteger and get its value returned. These methods are:

- addAndGet()
- getAndAdd()
- getAndIncrement()
- incrementAndGet()

The first method, addAndGet() adds a number to the AtomicInteger and returns its value after the addition. The second method, getAndAdd() also adds a number to the AtomicInteger but returns the value the AtomicInteger had before the value was added. Which of these two methods you should use depends on your use case. Here are two examples:

```
AtomicInteger atomicInteger = new AtomicInteger();


System.out.println(atomicInteger.getAndAdd(10));
System.out.println(atomicInteger.addAndGet(10));
```

This example will print out the values 0 and 20. First the example gets the value of the AtomicInteger before adding 10 to. Its value before addition is 0. Then the example adds 10 to the AtomicInteger and gets the value after the addition. The value is now 20.

You can also add negative numbers to the AtomicInteger via these two methods. The result is effectively a subtraction.

The methods getAndIncrement() and incrementAndGet() works like getAndAdd() and addAndGet() but just add 1 to the value of the AtomicInteger.

Subtracting From the AtomicInteger Value

The AtomicInteger class also contains a few methods for subtracting values from the AtomicInteger value atomically. These methods are:

- decrementAndGet()
- getAndDecrement()

The decrementAndGet() subtracts 1 from the AtomicInteger value and returns its value after the subtraction. The getAndDecrement() also subtracts 1 from the AtomicInteger value but returns the value the AtomicInteger had before the subtraction.

24. AtomicLong

The AtomicLong class provides you with a boolean variable which can be read and written atomically, and which also contains advanced atomic operations like compareAndSet(). The AtomicLong class is located in the java.util.concurrent.atomic package, so the full class name is java.util.concurrent.atomic.AtomicLong . This text describes the version of AtomicLong found in Java 8, but the first version was added in Java 5.
The reasoning behind the AtomicLong design is explained in my Java Concurrency tutorial in the text about Compare and Swap.

Creating an AtomicLong

Creating an AtomicLong is done like this:

```
AtomicLong atomicLong = new AtomicLong();
```

This example creates an AtomicLong with the initial value 0 .
If you want to create an AtomicLong with an initial value, you can do so like this:

```
AtomicLong atomicLong = new AtomicLong(123);
```

This example passes a value of 123 as parameter to the AtomicLong contructor, which sets the initial value of the AtomicLong instance to 123 .

Getting the AtomicLong Value

You can get the value of an AtomicLong instance via the get() method. Here is an AtomicLong.get() example:

```
AtomicLong atomicLong = new AtomicLong(123);

long theValue = atomicLong.get();
```

Setting the AtomicLong Value

You can set the value of an AtomicLong instance via the set() method. Here is an AtomicLong.set() example:

```
AtomicLong atomicLong = new AtomicLong(123);

atomicLong.set(234);
```

This example creates an AtomicLong example with an initial value of 123, and then sets its value to 234 in the next line.

Compare and Set the AtomicLong Value

The AtomicLong class also has an atomic compareAndSet() method. This method compares the current value of the AtomicLong instance to an expected value, and if the two values are equal, sets a new value for the AtomicLong instance. Here is an AtomicLong.compareAndSet() example:

```
AtomicLong atomicLong = new AtomicLong(123);

long expectedValue = 123;
long newValue           = 234;
atomicLong.compareAndSet(expectedValue, newValue);
```

This example first creates an AtomicLong instance with an initial value of 123 . Then it compares the value of the AtomicLong to the expected value 123 and if they are equal the new value of the AtomicLong becomes 234;

Adding to the AtomicLong Value

The AtomicLong class contains a few methods you can use to add a value to the AtomicLong and get its value returned. These methods are:

- addAndGet()
- getAndAdd()
- getAndIncrement()
- incrementAndGet()

The first method, addAndGet() adds a number to the AtomicLong and returns its value after the addition. The second method, getAndAdd() also adds a number to the AtomicLong but returns the value the AtomicLong had before the value was added. Which of these two methods you should use depends on your use case. Here are two examples:

- AtomicLong atomicLong = new AtomicLong();

- System.out.println(atomicLong.getAndAdd(10));
- System.out.println(atomicLong.addAndGet(10));

This example will print out the values 0 and 20. First the example gets the value of the AtomicLong before adding 10 to. Its value before addition is 0. Then the example adds 10 to the AtomicLong and gets the value after the addition. The value is now 20.

You can also add negative numbers to the AtomicLong via these two methods. The result is effectively a subtraction.
The methods getAndIncrement() and incrementAndGet() works like getAndAdd() and addAndGet() but just add 1 to the value of the AtomicLong.

Subtracting From the AtomicLong Value

The AtomicLong class also contains a few methods for subtracting values from the AtomicLong value atomically. These methods are:

- decrementAndGet()
- getAndDecrement()

The decrementAndGet() subtracts 1 from the AtomicLong value and returns its value after the subtraction. The getAndDecrement() also subtracts 1 from the AtomicLong value but returns the value the AtomicLong had before the subtraction.

25. AtomicReference

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 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 AtomicReference object.

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 typed AtomicReference.

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.