java.util.concurrent - Java Concurrency Utilities

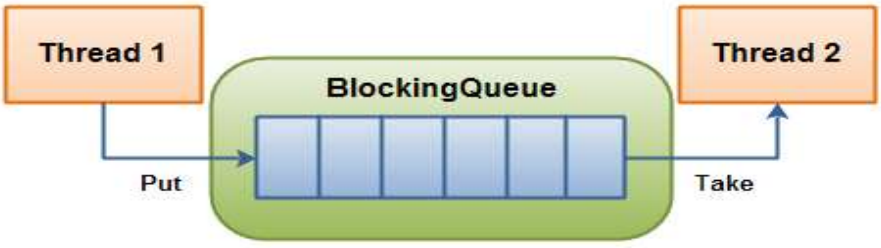
This package contains a set of classes that makes it easier to develop concurrent (multithreaded) applications in Java. Before this package was added, you would have to program your utility classes yourself.

BlockingQueue

The Java BlockingQueue interface in the java.util.concurrent package represents a queue which is thread safe to put into, and take instances from.

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 Implementations

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

- **ArrayBlockingQueue**
- **LinkedBlockingQueue**
- **PriorityBlockingQueue**
- **SynchronousQueue**

ArrayBlockingQueue

The ArrayBlockingQueue class implements the **BlockingQueue** 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();
```

LinkedBlockingQueue

The LinkedBlockingQueue class implements the **BlockingQueue** 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();
```

PriorityBlockingQueue

The PriorityBlockingQueue class implements the **BlockingQueue** 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 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();
```

SynchronousQueue

The SynchronousQueue class implements the **BlockingQueue** interface.

The SynchronousQueue is a queue that can only contain a single element internally. A thread inserting 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

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.

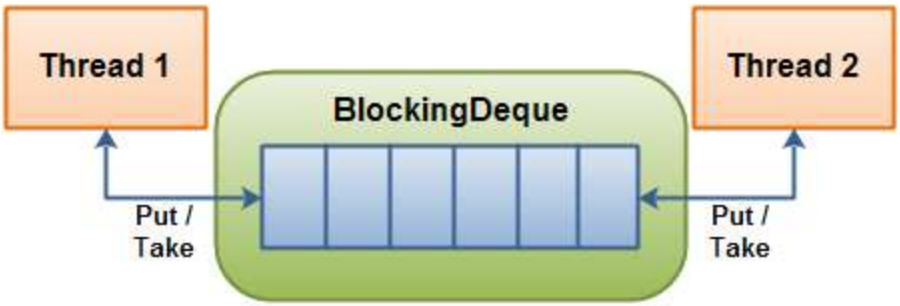
The BlockingDeque class is a Deque which blocks threads trying to insert or remove elements from the

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 producing 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 extends BlockingQueue

BlockingDeque Implementations

BlockingDeque is an interface.

Thejava.util.concurrent package has the following implementations of the BlockingDeque interface:

- **LinkedBlockingDeque**

LinkedBlockingDeque

The LinkedBlockingDeque class implements the **BlockingDeque** 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();
```

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. Thejava.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

theConcurrentHashMap is changed while being iterated. The Iterator is not designed to be used by more than one thread though.

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");
```

Q) What is fail-fast & fail-safe?

If Collection is modified structurally while one thread is iterating over it. This is because they work on clone of Collection instead of original collection and that's why they are called as fail-safe iterator.

Iterator written by ConcurrentHashMap keySet is fail-safe iterator and never throw ConcurrentModificationException in Java.

ConcurrentNavigableMap

The java.util.concurrent.ConcurrentNavigableMap inter face extends 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().

Implementation class :

`ConcurrentSkipListMap`

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.

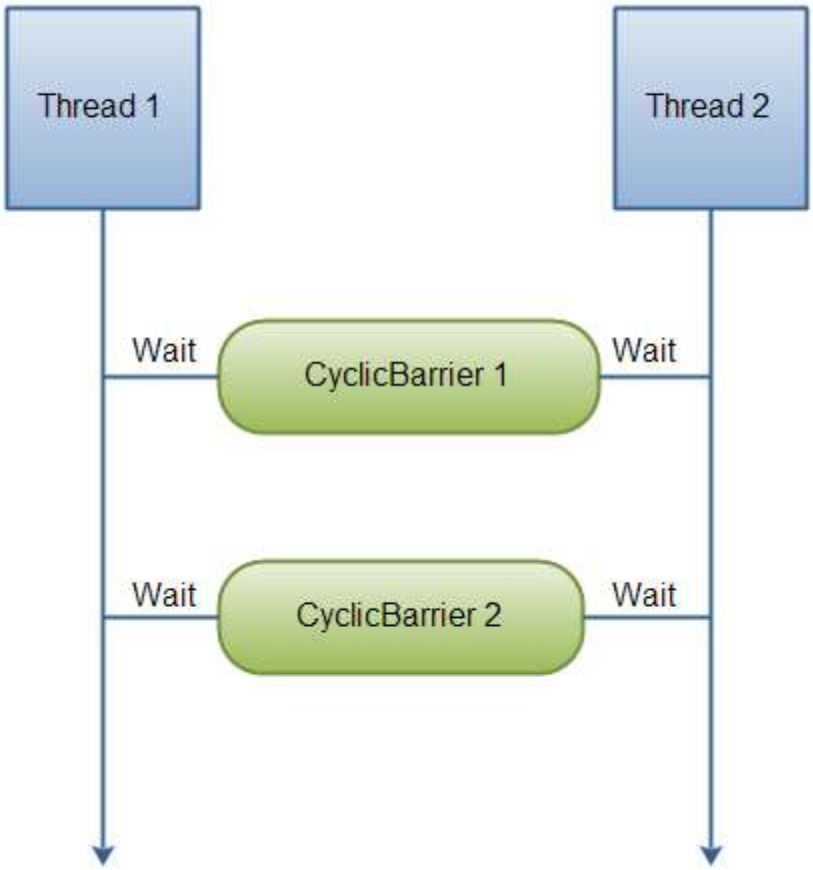
```
CountDownLatch latch = new CountDownLatch(3);

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();
        }
```

```
   }
}
```

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:

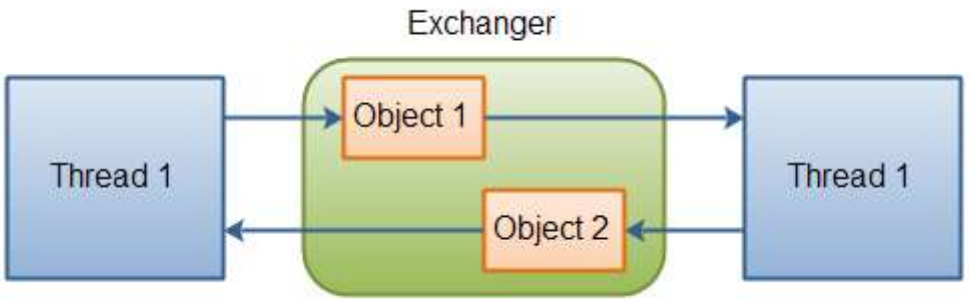
```
barrier.await(10, TimeUnit.SECONDS);
```

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.

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
```

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);
```

Refer :

https://www.mkyong.com/java/java-thread-mutex-and-semaphore-example/

Q) Difference between Semaphore & Mutex

Java multi threads example to show you how to use `Semaphore` and `Mutex` to limit the number of threads to access resources.

1. **`Semaphores`** – **Restrict the number of threads that can access a resource**. Example, limit max 10 connections to access a file simultaneously.
2. **`Mutex`** – **Only one thread to access a resource at once**. Example, when a client is accessing a file, no one else should have access the same file at the same time.