java.util.concurrent

1. BlockingQueue and ArrayBlockingQueue

1. **Class Description**

BlockingQueue prevents threads from popping when the queue is full or popping when the queue is full. Then the queue access itself is safe. Among the BlockingQueue types are arrayblockingqueue, LinkedBlockingQueue, PriorityBlockingQueue, and SynchronousQueue.
ArrayBlockingQueue is a class that implements the BlockingQueue interface. Store an Object in an Array. Therefore, it is a fixed-size queue and cannot be resized after it is created. A sequential queue of blocked threads is created, optionally with a fairness policy. Instead, the exact order of the queuing is not guaranteed.
The implementation of ArrayBlockingQueue is often described as a producer-consumer problem. The producer creates a new object, tries to insert until the queue is full, If the producer tries to insert and the queue is full, it is blocked. The block is blocked until the consumer holds the object and seats in the queue. The consumer tries to fetch the object from the blocking queue. If the consumer tries to take in an empty queue, it is blocked until the producer inserts the object.
There are four different methods for inserting / removing ArrayBlockingQueue. It depends on how it is handled when it is blocked.

① Throws Exception: add (o), remove (o) - throws an exception.

② Special Value: offer (o), poll () - special value is returned. (Mostly true / false)

③ Blocks: put (o), take () - The method call blocks.

④ Times Out: offer (o, timeout, timeunit), poll (timeout, timeunit) - The method call blocks but does not exceed the given timeout. Returns the success of the operation as a special value. (Mostly true, false)
If you insert a null in the blockingqueue it will throw a NullPointerException.

2. **Code Example**

The attached code is an example code for BlockingQueue.
I created an ArrayBlockingQueue and solved the Producer-Consumer problem. You can see the process of inserting and subtracting "Item thread_id #".
You can see the result screen below.

```
<terminated> ex1 [Java Applicatio
Producer Thread Start!
Consumer Thread Start!
produce : Item0
comsume : Item0
produce : Item1
produce : Item2
produce : Item3
comsume : Item1
produce : Item4
produce : Item5
comsume : Item2
comsume : Item3
produce : Item6
produce : Item7
comsume : Item4
comsume : Item5
produce : Item8
comsume : Item6
produce : Item9
comsume : Item7
comsume : Item8
comsume : Item9
```

2. Semaphore

1. Class Description

The semaphore class is a class that counts the number of semaphores. Semaphores restrict access to critical sections. Types of functions include acquire () and release () functions. Initialize the number of semaphores by the number of permits.

Acquire a permit through acquire(), and return a permit through release(). There are as many initializations as you can get a permit. For example, if you initialize permits to N, then N threads can work at the same time. Do not allow more than N threads to be accessed at a time.

If you want to enter,

① Get permission

② Access critical section

③ Release.

If you do not get permission to reach the thread tolerance limit, wait until another thread releases it. The waiting thread is in the queue, which can be acquired by another thread in the queue when released by another thread. But there is no guarantee of fairness. Even if you are waiting at the first of the queues, you cannot get acquire for the first time. To be fair, the semaphore class has a boolean value in the constructor that this semaphore should be fair. Having fairness causes a performance / concurrency penalty, so you should only use it when you really need it.

There are other functions like this.

→ availablePermits(), acquireUninterruptibly(), drainPermits(), hasQueuedThreads(), getQueuedThreads(), tryAcquire(), etc.

2. Code Example

The attached code is an example code for Semaphore class.

I created a Semaphore class and gave the constructor the argument 2, which made the two threads accessible to the critical section. The thread accesses the critical section in one second.

You can see the result screen below.

```
<terminated> ex2 [Java Applica
Thread 0 : acquire
Thread 1 : acquire
Thread 0 : release
Thread 1 : release
Thread 2 : acquire
Thread 3 : acquire
Thread 2 : release
Thread 3 : release
Thread 7 : acquire
Thread 4 : acquire
Thread 7 : release
Thread 4 : release
Thread 6 : acquire
Thread 5 : acquire
Thread 6 : release
Thread 5 : release
Thread 8 : acquire
Thread 9 : acquire
Thread 8 : release
Thread 9 : release
```

3. ReadWriteLock

1. **Class Description**

The ReadWriteLock class provides an advanced way to handle thread locks. Multiple threads can simultaneously access resources for reading, but only one thread is allowed to write. A concurrency error does not occur when multiple threads access a resource for a read operation. However, errors occur when read and write operations occur at the same time, or when multiple write operations occur at the same time. When implementing ReadWriteLock, you need to create two locks, one for read access and one for write access.
The rules for locking are as follows.
→ ReadLock

① There is no thread putting a lock for writing.

② There is no thread requesting that lock.

③ Multiple threads can then lock for reading.

→ WriteLock

① No thread has a reading or writing lock.

② Then one thread can lock for writing at a time.

2. **Code Example**

The attached code is an example code for ReadWriteLock class.
I created a Reader thread and a Writer thread using the ReadWriteLock class. In the result screen below, the read operation is executed simultaneously, but the write operation is executed one at a time.
You can see the result screen below.

```
<terminated> ex3 [Java Application] C:₩P
Thread 1 is reading
Thread 0 is reading
Thread 1 done reading
Thread 0 done reading
Thread 1 is writing alone
Thread 1 done writing
Thread 0 is writing alone
Thread 0 done writing
Thread 2 is writing alone
Thread 2 done writing
Thread 2 is reading
Thread 2 done reading
Thread 3 is writing alone
Thread 3 done writing
Thread 3 is reading
Thread 4 is reading
Thread 3 done reading
Thread 4 done reading
Thread 4 is writing alone
Thread 4 done writing
```

4. AtomicInteger

1. Class Description

   The atomicinteger class is a class that can atomically read and write integer variables. It is in the java.util.concurrent.atomic package.
   The implementation method calls the AtomicInteger constructor first. Constructor Initialized to the number enclosed in parentheses.
   You can get an instance of AtomicInteger through the get () method.
   You can set an instance of AtomicInteger through the set () method.
   Compares the current value of the atomicInteger with the expected value through the compareAndSet () method, and stores the new value if the two values are equal.
   In addition, there are the following functions.

   - AddAndGet (): Adds the number in parentheses, returns, and subtracts a negative number.
   - GetAndAdd (): returns and adds the number.
   - IncrementAndGet (): Add 1 and return.
   - GetAndIncrement (): Add 1 after return.
   - DecrementAndGet (): Returns 1 after subtracting.
   - GetAndDecrement (): Subtract 1 after return.

2. **Code Example**

   The attached code is an example code for AtomicInteger class.
   Through the TestAtomicClass, each thread is performing a task of adding and subtracting its own id.
   You can see the result screen below.