

## Report: Problem 3

소프트웨어학부 20204898 박소은

### (i) BlockingQueue

#### (i)-a. Explain about **BlockingQueue** and **ArrayBlockingQueue**

BlockingQueue is a thread-safe queue interface. This means that in a multi-threaded environment, attempting to insert an element into a full queue/attempting to take an element from an empty queue will block the thread. The blocked thread remains blocked until an element is taken/inserted.

BlockingQueue supports 4 methods:

1. Throw exception: It throws an exception if `add()`, `remove()` or `element()` is not possible.
2. Special value: It returns a special value if `offer()`, `poll()` or `peek()` is not possible.
3. Blocks: It blocks if `put()` or `take()` is not possible.
4. Times out: It waits for the given timeout if `offer()` or `poll()` is not possible.

	1. Throw Exception	2. Special Value	3. Blocks	4. Times Out
Insert	<code>add(o)</code>	<code>offer(o)</code>	<b><code>put(o)</code></b>	<code>offer(o, timeout, timeunit)</code>
Remove	<code>remove(o)</code>	<code>poll()</code>	<b><code>take()</code></b>	<code>poll(timeout, timeunit)</code>
Examine	<code>element()</code>	<code>peek()</code>		

ArrayBlockingQueue is a class which implemented the BlockingQueue interface. It is a limited sized array which stores elements in FIFO order.

#### (i)-b. Create an example of multithreaded JAVA code including **put()** and **take()** methods.

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class ex1 {
    public static void main(String[] args) {
        BlockingQueue queue = new ArrayBlockingQueue(2);

        ProducerThread producer = new ProducerThread(queue);
        producer.start();
    }
}
```

```

        ConsumerThread consumer = new ConsumerThread(queue);
        consumer.start();
    }

    static class ConsumerThread extends Thread {
        BlockingQueue queue;

        ConsumerThread(BlockingQueue queue) { this.queue = queue; }

        public void run() {
            try {
                Thread.sleep(500);
                System.out.println("Wn< ConsumerThread Starts >");

                for(int i=0; i<4; i++) {
                    System.out.println("Wn[TAKE] Current Queue: " + queue.toString());
                    System.out.println("[TAKE] Try to take an element");
                    System.out.println("[TAKE] Took an element: " + queue.take());
                    System.out.println("[TAKE] Queue remaining capacity: " +
queue.remainingCapacity());
                }
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }

    static class ProducerThread extends Thread {
        BlockingQueue queue;

        ProducerThread(BlockingQueue queue) { this.queue = queue; }

        public void run() {
            System.out.println("< ProducerThread Starts >");
            try {
                System.out.println("Wn[PUT] Queue remaining capacity: "+ queue.remainingCapacity());
                System.out.println("[PUT] Try to insert 1");
                queue.put(1);
                System.out.println("[PUT] Current Queue: " + queue.toString());

                System.out.println("Wn[PUT] Queue remaining capacity: "+ queue.remainingCapacity());
                System.out.println("[PUT] Try to insert 2");
                queue.put(2);
                System.out.println("[PUT] Current Queue: " + queue.toString());

                System.out.println("Wn[PUT] Queue remaining capacity: "+ queue.remainingCapacity());
                System.out.println("[PUT] Try to insert 3");
                queue.put(3);
                System.out.println("[PUT] Current Queue: " + queue.toString());

            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

```

- Example execution result

```
ex1 x
"C:\Program Files\Java\jdk-11.0.17\bin\java.
< ProducerThread Starts >

[PUT] Queue remaining capacity: 2
[PUT] Try to insert 1
[PUT] Current Queue: [1]

[PUT] Queue remaining capacity: 1
[PUT] Try to insert 2
[PUT] Current Queue: [1, 2]

[PUT] Queue remaining capacity: 0
[PUT] Try to insert 3

< ConsumerThread Starts >

[TAKE] Current Queue: [1, 2]
[TAKE] Try to take an element
[PUT] Current Queue: [2, 3]
[TAKE] Took an element: 1
[TAKE] Queue remaining capacity: 0

[TAKE] Current Queue: [2, 3]
[TAKE] Try to take an element
[TAKE] Took an element: 2
[TAKE] Queue remaining capacity: 1

[TAKE] Current Queue: [3]
[TAKE] Try to take an element
[TAKE] Took an element: 3
[TAKE] Queue remaining capacity: 2

[TAKE] Current Queue: []
[TAKE] Try to take an element
|
```

#### *Queue capacity: 2*

1. A producer thread wants to put 3 elements in the queue.
2. When it tries to put third element, it is blocked.
3. After 500ms, Consumer thread starts.
4. When consumer thread took first element, the producer thread wakes up, and put third element.
5. The consumer thread continues taking the elements out. When it tries to take an element out of the queue, it is blocked.

## (ii) ReadWriteLock

(ii)-a. Explain about ReadWriteLock.

The ReadWriteLock interface allows multiple threads to read shared resources, while writing can only be done by one thread at a time.

- Reading: If the ReadWriteLock is not locked for writing, multiple threads can lock for reading.
- Writing: If no threads are reading or writing (if the ReadWriteLock is not locked), one thread can lock for writing.

Therefore, there are two types of locks: readLock and writeLock.

- readLock
  - ReadWriteLock.readLock().lock(): Lock the readLock. Before unlocking the readLock, other threads cannot write. Multiple threads can execute this method.
  - ReadWriteLock.readLock().unlock(): After executing this method, if there are no readLock/writeLock left, a thread can write.
- writeLock
  - ReadWriteLock.writeLock().lock(): After executing this method, other threads cannot write.
  - ReadWriteLock.writeLock().unlock(): After executing this method, other threads can write.

ReentrantReadWriteLock class implemented the ReadWriteLock interface.

(ii)-b. Create an example of multithreaded JAVA code including **lock()**, **unlock()**, **readLock()**, **writeLock()** methods.

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ex2 {
    public static void main(String[] args) {
        ReadWriteLock lock = new ReentrantReadWriteLock();

        ReadThread readThread1 = new ReadThread(lock, 1);
        readThread1.start();
    }
}
```

```

ReadThread readThread2 = new ReadThread(lock, 2);
readThread2.start();

WriteThread writeThread3 = new WriteThread(lock, 3);
writeThread3.start();

WriteThread writeThread4 = new WriteThread(lock, 4);
writeThread4.start();
}

static class WriteThread extends Thread {
    ReadWriteLock lock;
    int id;

    WriteThread(ReadWriteLock lock, int id) {
        this.lock = lock;
        this.id = id;
    }

    @Override
    public void run() {
        // writeLock().lock()
        System.out.println("Thread #" + id + ": Try to lock writeLock()");
        lock.writeLock().lock();
        System.out.println("Thread #" + id + ": Locked writeLock()");

        // writeLock().unlock()
        System.out.println("Thread #" + id + ": Try to unlock writeLock()");
        lock.writeLock().unlock();
    }
}

static class ReadThread extends Thread {
    ReadWriteLock lock;
    int id;

    ReadThread(ReadWriteLock lock, int id) {
        this.lock = lock;
        this.id = id;
    }

    @Override
    public void run() {
        // readLock().lock()
        System.out.println("Thread #" + id + ": Try to lock readLock()");
        lock.readLock().lock();
        System.out.println("Thread #" + id + ": Locked readLock()");

        // Thread sleeps
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        // readLock().unlock()
        System.out.println("Thread #" + id + ": Try to unlock readLock()");
        lock.readLock().unlock();
    }
}

```

```
}  
}  
}
```

- Example execution

4 threads try to lock: thread1, 2 lock readLock, thread3, 4 lock writeLock.

ReadLocks sleep for 5s after locking the readLock, therefore readThreads can lock at the same time.

If the writeThread lock first, the readThreads and the other writeThread have to wait until the writeThread unlocks. On the other hand, if the readThread lock first, the other readThread can also lock the readLock, and the writeLocks have to wait until the readThreads unlock the lock.

- Execution result 1

```
Thread #2: Try to lock readLock()  
Thread #4: Try to lock writeLock()  
Thread #3: Try to lock writeLock()  
Thread #1: Try to lock readLock()  
Thread #3: Locked writeLock()  
Thread #3: Try to unlock writeLock()  
Thread #1: Locked readLock()  
Thread #2: Locked readLock()  
Thread #1: Try to unlock readLock()  
Thread #2: Try to unlock readLock()  
Thread #4: Locked writeLock()  
Thread #4: Try to unlock writeLock()  
  
Process finished with exit code 0
```

1. writeThread(Thread3) locked the writeLock first, therefore the other writeThread(Thread4) and readThreads waits until Thread3 unlock the lock.

2. After Thread3 unlock, readThreads(Thread1, Thread2) locked. The remaining writeThread(Thread4) waits until the readThreads unlock.

- Execution result 2

```
Thread #2: Try to lock readLock()
Thread #3: Try to lock writeLock()
Thread #4: Try to lock writeLock()
Thread #1: Try to lock readLock()
Thread #2: Locked readLock()
Thread #1: Locked readLock()
Thread #2: Try to unlock readLock()
Thread #1: Try to unlock readLock()
Thread #4: Locked writeLock()
Thread #4: Try to unlock writeLock()
Thread #3: Locked writeLock()
Thread #3: Try to unlock writeLock()

Process finished with exit code 0
```

1. This time, readThreads locked first.
2. As the writeThreads cannot lock simultaneously, after one writeThread is finished, the next thread locks.

### (iii) AtomicInteger

(iii)-a. Explain about **AtomicInteger**.

You can use an integer variable which provides concurrency in multi-threaded environment. Multiple threads can read and write the AtomicInteger object. Originally, all threads must be synchronized to do atomic operations, but the AtomicInteger class can safely increase and decrease (AtomicInteger) variable value without synchronization.

- `get()`: Get the AtomicInteger (integer) value.
- `incrementAndGet()`: Add 1 to the AtomicInteger value and get the value. Same with ``++i``.
- `getAndIncrement()`: Get the AtomicInteger value and add 1 to the value. Same with ``i++``.
- `decrementAndGet()`: Subtract 1 to the AtomicInteger value and get the value. Same with ``--i``.
- `getAndDecrement()`: Get the AtomicInteger value and subtract 1 to the value. Same with ``i--``.
- `addAndGet(num)`: Add 'num' to the AtomicInteger value and get the value.
- `getAndAdd(num)`: Get the AtomicInteger value and add 'num' to the value.
- `set(num)`: Set 'num' in the AtomicInteger value.
- `getAndSet(num)`: Get the AtomicInteger value and set 'num'.

(iii)-b. Create an example of multithreaded JAVA code including **get()**, **set()**, **getAndAdd()**, **addAndGet()** methods.

```
import java.util.concurrent.atomic.AtomicInteger;

public class ex3 {
    public static void main(String[] args) {
        AtomicInteger atomic = new AtomicInteger(10);
        for (int i=1; i<=5; i++) {
            new AtomicIntegerThread(i, atomic);
        }
    }

    static class AtomicIntegerThread extends Thread {
        AtomicInteger atomic;
        int id;

        AtomicIntegerThread(int id, AtomicInteger atomic) {
            this.id = id;
        }
    }
}
```



```

        this.atomic = atomic;
        this.start();
    }

    @Override
    public void run() {
        try {
            sleep((int)(Math.random() * 1000));
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        atomic.set(this.id);

        System.out.println("[Thread " + this.id + "] get(): " + atomic.get() +
            ", getAndAdd(" + this.id + "): " + atomic.getAndAdd(this.id));
        System.out.println("[Thread " + this.id + "] get(): " + atomic.get() +
            ", addAndGet(" + this.id + "): " + atomic.addAndGet(this.id));
    }
}

```

- Example execution result

```

"C:\Program Files\Java\jdk-11.0.17\bin\jav
[Thread 1] get(): 1, getAndAdd(1): 1
[Thread 5] get(): 5, getAndAdd(5): 5
[Thread 4] get(): 4, getAndAdd(4): 4
[Thread 1] get(): 2, addAndGet(1): 3
[Thread 5] get(): 3, addAndGet(5): 8
[Thread 4] get(): 8, addAndGet(4): 12
[Thread 3] get(): 3, getAndAdd(3): 3
[Thread 3] get(): 6, addAndGet(3): 9
[Thread 2] get(): 2, getAndAdd(2): 2
[Thread 2] get(): 4, addAndGet(2): 6

Process finished with exit code 0

```

As soon as `run()` is executed, the thread sets the atomic variable value to its 'Thread ID':

```
atomic.set(this.id)
```

And then the thread runs `getAndAdd(this.id)` and `addAndGet(this.id)`

When '`getAndAdd()`' is executed, the return value is same with '`get()`'s return value.

On the other hand, when '`addAndGet()`' is executed, its return value is '`get()`' + Thread.id.

We can see that '`addAndGet(num)`' gets the value after adding '`num`' to the AtomicInteger value, and `getAndAdd(num)` adds '`num`' to the value after getting the AtomicInteger value.

## (iv) CyclicBarrier

(iv)-a. Explain about **CyclicBarrier**

Threads have to wait until the number of waiting threads reaches specific value. After reaching the number, the threads are released and continue.

- `CyclicBarrier(int parties)`: Constructor of `CyclicBarrier`. The number of waiting threads should be 'parties' to be released. Before reaching the number, threads have to wait.
- `CyclicBarrier(int parties, Runnable barrierAction)`: Constructor of `CyclicBarrier`. If the number of waiting threads reaches 'parties', `barrierAction` is executed. The last thread executes `barrierAction`. If there is no need to run `barrierAction`, you can give null value.
- `CyclicBarrier.await()`: The thread has to wait until the number of waiting threads reaches 'parties'.
- `CyclicBarrier.await(int time, TimeUnit.SECONDS)`: You can limit the timeout of the waiting thread.
- `CyclicBarrier.getParties()`: It return 'parties'.

In other cases, if another thread interrupts or `reset()` method is called, threads can stop waiting.

(iv)-b. Create an example of multithreaded JAVA code including **await()** method.

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class ex4 {
    public static void main(String[] args) {
        BarrierAction action = new BarrierAction();
        CyclicBarrier barrier = new CyclicBarrier(3, action);

        for(int i=1; i<=6; i++) {
            new CyclicBarrierThread("Thread "+i, barrier);
        }
    }

    static class BarrierAction implements Runnable {
        @Override
        public void run() {
            System.out.println(" ! BarrierAction is executed ! ");
        }
    }

    static class CyclicBarrierThread extends Thread {
        String name;
```

```

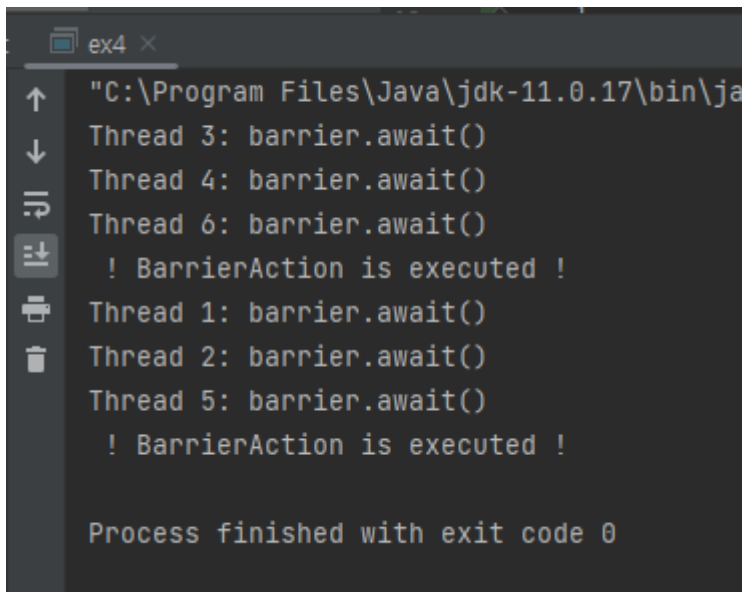
CyclicBarrier barrier;

CyclicBarrierThread(String name, CyclicBarrier barrier) {
    this.name = name;
    this.barrier = barrier;
    this.start();
}

@Override
public void run() {
    try {
        sleep((int)(Math.random() * 10000));
        System.out.println(this.name + ": barrier.await()");
        barrier.await();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } catch (BrokenBarrierException e) {
        throw new RuntimeException(e);
    }
}
}
}

```

- Example execution result



```

ex4 x
"C:\Program Files\Java\jdk-11.0.17\bin\java.exe"
Thread 3: barrier.await()
Thread 4: barrier.await()
Thread 6: barrier.await()
! BarrierAction is executed !
Thread 1: barrier.await()
Thread 2: barrier.await()
Thread 5: barrier.await()
! BarrierAction is executed !

Process finished with exit code 0

```

In the example code, CyclicBarrier needs 3 threads to await(), and if the number of waiting threads is 3, BarrierAction is executed.

There are 6 threads: Everytime 3 threads wait, the BarrierAction is executed.