Problem 3 report

Antoine DESRUET 50221600

1 A

BlockingQueue

`BlockingQueue` is an interface for a queue that is thread safe. That means that the implementation of this queue allow one thread at a time to acces the queue data.

ArrayBlockingQueue

`ArrayBlockingQueue` is one implementation of the `BlockingQueue` interface that use an array. That means that the data structure use to store the data is an array. The array has a predifined memory size, it means that it's not dynamic. It implements the `BlockingQueue` methods in order to use the array as a queue, also those methods are thread safe.

Code

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

class Cooker extends Thread {
    BlockingQueue<String> _orders;

    Cooker(BlockingQueue<String> o) {
        _orders = o;
    }

    @Override
    public void run() {
        String pizza;
        while (true) {
            try {
                pizza = _orders.take();
                System.out.printf("Cooker: Making %s pizza\n", pizza);
                sleep(1000);
            } catch (Exception e) {
            }
        }
    }
}

class Waiter extends Thread {
    BlockingQueue<String> _orders;
    Random rand = new Random();
```

```
    Waiter(BlockingQueue<String> o) {
        _orders = o;
    }

    @Override
    public void run() {
        String currentPizza;
        String[] pizzas = { "Peperoni", "Sweat potato", "Bulgogi",
"Chicken" };
        while (true) {
            try {
                currentPizza = pizzas[rand.nextInt(pizzas.length)];
                _orders.put(currentPizza);
                System.out.printf("Waiter: Adding %s pizza to queue\n",
currentPizza);
                sleep(500);
            } catch (Exception e) {
            }
        }
    }

}

public class ex1 {
    public static void main(String[] args) {
        BlockingQueue<String> queue = new ArrayBlockingQueue<String>(100);
        Cooker c = new Cooker(queue);
        Waiter w = new Waiter(queue);

        c.start();
        w.start();
        try {
            c.join();
            w.join();
        } catch (Exception e) {
        }
    }
}
```

Execution explaination

One thread push string to the `BlockingQueue` and sleep, the other thread pop one string and sleep.

2 A

ReadWriteLock

`ReadWriteLock` is a specific type of lock. It can allow multiple threads to acces the data to read at the same time. But it only allow one thread at a time to acces the data to modify it.

Code

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

class Consumer extends Thread {
    ReadWriteLock _lock;
    List<Integer> _list;

    Consumer(ReadWriteLock lock, List<Integer> l) {
        _lock = lock;
        _list = l;
    }

    @Override
    public void run() {
        int n = 0;
        while (true) {
            _lock.readLock().lock();
            if (_list.size() == 0) {
                _lock.readLock().unlock();
                continue;
            }
            n = _list.get(0);
            _lock.readLock().unlock();

            System.out.printf("Thread number %d: first number == %d\n",
getId(), n);
            try {
                sleep(500);
            } catch (Exception e) {
            }
        }
    }
}

class Sender extends Thread {
    ReadWriteLock _lock;
    List<Integer> _list;
    Random rand = new Random();

    Sender(ReadWriteLock lock, List<Integer> l) {
        _lock = lock;
        _list = l;
    }

    @Override
    public void run() {
        int n;
        while (true) {
            n = rand.nextInt();
            _lock.writeLock().lock();
```

```
            System.out.printf("Thread number %d: adding %d to list\n",
this.getId(), n);
            _list.add(0, n);
            _lock.writeLock().unlock();
            try {
                sleep(500);
            } catch (Exception e) {
            }
        }
    }
}

public class ex2 {
    static private final int N_CONSUMER = 10;
    static private final int N_SENDER = 10;

    public static void main(String[] args) {
        List<Integer> l = new ArrayList<Integer>();
        ReadWriteLock lock = new ReentrantReadWriteLock();
        Consumer[] consumer_threads = new Consumer[N_CONSUMER];
        Sender[] sender_threads = new Sender[N_SENDER];

        for (int i = 0; i < N_CONSUMER; ++i) {
            consumer_threads[i] = new Consumer(lock, l);
            consumer_threads[i].start();
        }
        for (int i = 0; i < N_SENDER; ++i) {
            sender_threads[i] = new Sender(lock, l);
            sender_threads[i].start();
        }
        for (int i = 0; i < N_CONSUMER; ++i) {
            try {
                consumer_threads[i].join();
            } catch (Exception e) {
            }
        }
        for (int i = 0; i < N_SENDER; ++i) {
            try {
                sender_threads[i].join();
            } catch (Exception e) {
            }
        }
    }
}
```

Execution explaination

A set of 10 threads try to populate a list with random numbers. A set of 10 thread print the first number of the list.

3 A

AtomicInteger

`AtomicInteger` is an `atomic` variable. It allows to do atomic operations on `int` type variable. Atomic variables allows to do operations on variables on different thread at the same time without concurency problems.

Code

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

class T1 extends Thread {
    AtomicInteger _i;

    T1(AtomicInteger i) {
        _i = i;
    }

    @Override
    public void run() {
        while (true) {
            if (_i.get() > 20) {
                _i.set(0);
                System.out.println("Reseting the integer to 0");
                try {
                    sleep(1000);
                } catch (Exception e) {
                }
            }

        }
    }
}

class T2 extends Thread {
    AtomicInteger _i;

    T2(AtomicInteger i) {
        _i = i;
    }

    @Override
    public void run() {
        while (true) {
            System.out.printf("Thread %d: after increment %d\n", getId(),
_i.addAndGet(1));
            try {
                sleep(500);
            } catch (Exception e) {
            }
        }
    }
}
```

```
class T3 extends Thread {
    AtomicInteger _i;

    T3(AtomicInteger i) {
        _i = i;
    }

    @Override
    public void run() {
        while (true) {
            System.out.printf("Thread %d: before increment %d\n", getId(),
_i.getAndAdd(1));
            try {
                sleep(500);
            } catch (Exception e) {
            }
        }
    }
}

public class ex3 {
    public static void main(String[] args) {
        AtomicInteger AInteger = new AtomicInteger(0);
        T1 t1 = new T1(AInteger);
        T2 t2 = new T2(AInteger);
        T3 t3 = new T3(AInteger);

        t1.start();
        t2.start();
        t3.start();

        try {
            t1.join();
            t2.join();
            t3.join();

        } catch (Exception e) {
        }
    }
}
```

Execution explaination

Creating 3 threads. Thread number 1 check if the number is over 20, if it's true it reset the number to 0 and then sleep for 1s. Thread number 2 increment the number and then print it. Thread number 3 print the number and then increment it.

4 A

CyclicBarrier

`CyclicBarrier` is used to syncronize threads. It kind of works like a break point in the source code. Once a thread reach this point it will wait until the others reach this point too.

Code

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

class T1 extends Thread {
    CyclicBarrier _barrier;

    T1(CyclicBarrier b) {
        _barrier = b;
    }

    @Override
    public void run() {
        int sleep_time = 100;
        while (true) {
            try {
                System.out.printf("Thread %d: will sleep for %dms\n",
getId(), sleep_time);
                sleep(sleep_time);
                _barrier.await();
            } catch (Exception e) {
            }
        }
    }
}

class T2 extends Thread {
    CyclicBarrier _barrier;

    T2(CyclicBarrier b) {
        _barrier = b;
    }

    @Override
    public void run() {
        int sleep_time = 300;
        while (true) {
            try {
                System.out.printf("Thread %d: will sleep for %dms\n",
getId(), sleep_time);
                sleep(sleep_time);
                _barrier.await();
            } catch (Exception e) {
            }
        }
    }
}

class T3 extends Thread {
```

```
    CyclicBarrier _barrier;

    T3(CyclicBarrier b) {
        _barrier = b;
    }

    @Override
    public void run() {
        int sleep_time = 500;
        while (true) {
            try {
                System.out.printf("Thread %d: will sleep for %dms\n",
getId(), sleep_time);
                sleep(sleep_time);
                _barrier.await();
            } catch (Exception e) {
            }
        }
    }
}

class T4 extends Thread {
    CyclicBarrier _barrier;

    T4(CyclicBarrier b) {
        _barrier = b;
    }

    @Override
    public void run() {
        int sleep_time = 700;
        while (true) {
            try {
                System.out.printf("Thread %d: will sleep for %dms\n",
getId(), sleep_time);
                sleep(sleep_time);
                _barrier.await();
            } catch (Exception e) {
            }
        }
    }
}

public class ex4 {
    public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(4, new Runnable() {
            public void run() {
                System.out.println("Threads synchronized");
            }
        });
        Thread[] threads = { new T1(barrier), new T2(barrier), new
T3(barrier), new T4(barrier) };
        for (int i = 0; i < threads.length; ++i) {
            threads[i].start();
```

```
        }
        for (int i = 0; i < threads.length; ++i) {
            try {
            threads[i].join();
            } catch (Exception e) {}
        }
    }
}
```

Execution explaination

4 Threads are created, each one has a different sleep duration. Then they wait to be synchronized with the cyclic barrier.