(i)
BlockingQueue and ArrayBlockingQueue explanation

The BlockingQueue interface extends the Queue interface to provide additional methods to support blocking operations. It is also a packaged interface that represents a thread-safe queue designed for concurrent work involving multiple threads.

BlockingQueue allows threads to block when trying to add an element if the queue is full or retrieve an element if the queue is empty. These blocking behaviors are used to synchronize threads and manage resource contention in a concurrent environment.

The main methods provided to the BlockingQueue interface are put(), take(), offer(), and poll().

put() adds the specified element to the queue and waits for space to become available if needed.

take() retrieves and removes the head of the queue, waiting for an element to become available if necessary.

offer() should wait until the specified timeout to add the specified element to the queue and make space available if needed.

poll() retrieves and removes the head of the queue, waiting until a specified timeout if necessary for elements to become available.

ArrayBlockingQueue is a class that makes it possible to wait until the condition is met for either the insertion operation or the subtraction operation. Allows multiple threads to safely insert and retrieve elements from a queue without causing data inconsistencies. Queues follow the principle of first-in, first-out order. The element that has been in the queue the longest is removed first when it is dequeued.

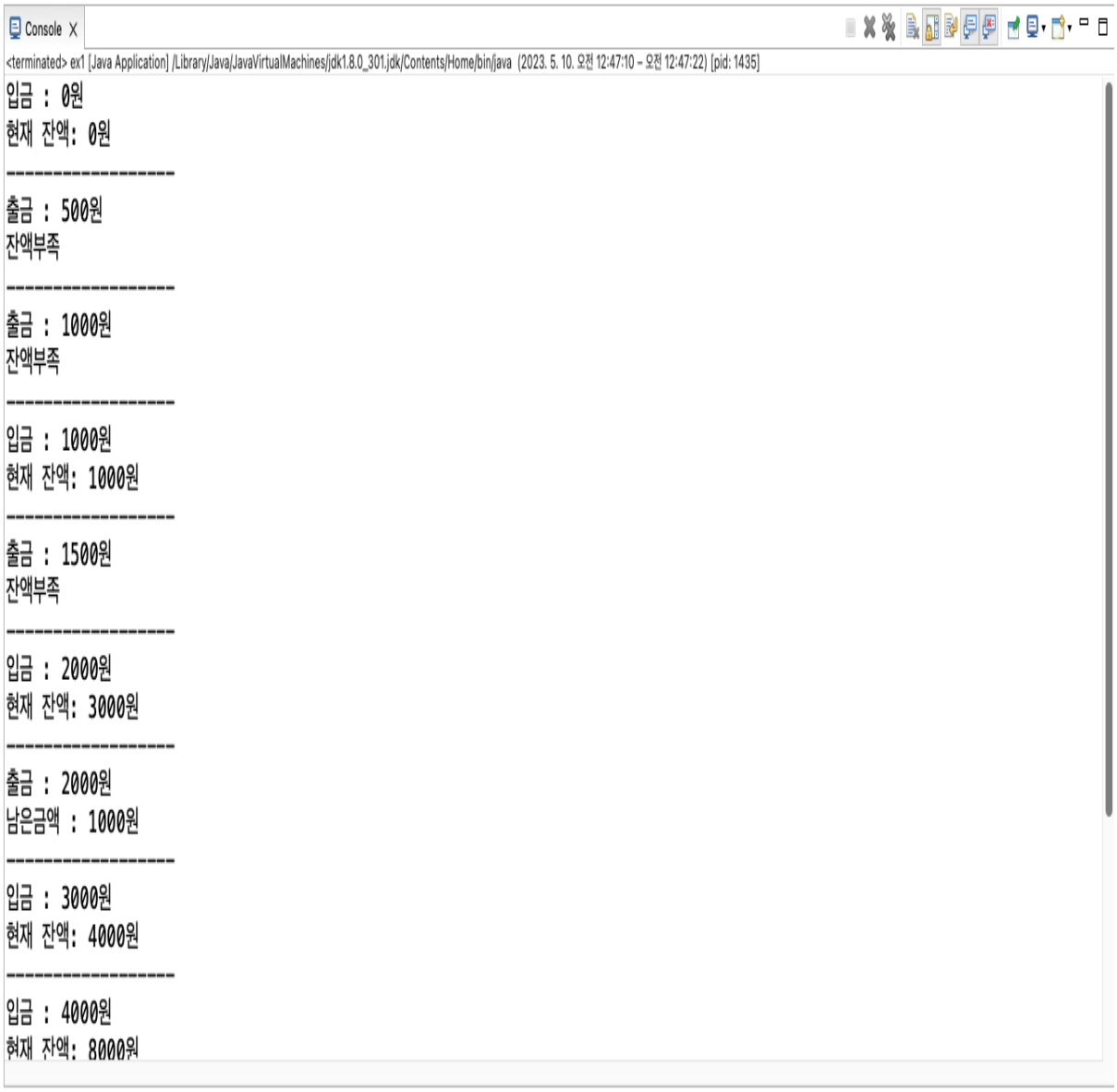
Supports an optional fairness policy for ordering queued producer and consumer threads.

(i)
BlockingQueue and ArrayBlockingQueue Source code

```
package Prob3;

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

public class ex1 {
	public static void main(String[] args) {
		BlockingQueue<Integer> account = new ArrayBlockingQueue<>(1);
		account.add(0);

		Thread deposit = new Thread(new Deposit(account));
		Thread withdraw = new Thread(new Withdraw(account));1

		deposit.start();
		withdraw.start();

	}
}


class Deposit implements Runnable{
	private BlockingQueue<Integer> account;

	Deposit(BlockingQueue<Integer> account){
		this.account = account;
	}

	@Override
	public void run() {
		for(int i=0;i<=5;i++) {
			try {
				int currentBalance = account.take();
				int newBalance = currentBalance + i * 1000;
				System.out.println("입금 : " + (i * 1000) + "원");
				System.out.println("현재 잔액: " +newBalance + "원");
		System.out.println("------------------");
				account.put(newBalance);
				Thread.sleep(2000);
			}catch(InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}


class Withdraw implements Runnable{
	private BlockingQueue<Integer> account;

	Withdraw(BlockingQueue<Integer> account){
		this.account = account;
	}

	@Override
	public void run() {
```

```
                for(int i = 1;i<=5;i++) {
                        try {
                                int currentBalance = account.take();
                                int amountToWithdraw = i * 500;
                                System.out.println("출금 : " + amountToWithdraw + "원");
                                if(currentBalance >= amountToWithdraw) {
                                        int newBalance = currentBalance - amountToWithdraw;
                                        System.out.println("남은금액 : " + newBalance + "원");
                                        System.out.println("------------------");
                                        account.put(newBalance);
                                } else {
                                        System.out.println("잔액부족");
                                        System.out.println("------------------");
                                        account.put(currentBalance);
                                }
                                Thread.sleep(2000);
                        }catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

(i)
BlockingQueue and ArrayBlockingQueue Result


Console X
<terminated> ex1 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_301.jdk/Contents/Home/bin/java (2023. 5. 10. 오전 12:47:10 – 오전 12:47:22) [pid: 1435]
입금 : 0원
현재 잔액: 0원

출금 : 500원
잔액부족

출금 : 1000원
잔액부족

입금 : 1000원
현재 잔액: 1000원

출금 : 1500원
잔액부족

입금 : 2000원
현재 잔액: 3000원

출금 : 2000원
남은금액 : 1000원

입금 : 3000원
현재 잔액: 4000원

입금 : 4000원
현재 잔액: 8000원

(ii)
ReadWriteLock
explanation

ReadWriteLock manages concurrent access to shared resources. Allows multiple threads to read a resource concurrently, while restricting write access to only one thread at a time.

There are two related locks: read locks and write locks. A read lock allows multiple threads to access and read a shared resource concurrently. A write lock can only be acquired by one thread for writing. Once the write lock is acquired, no other thread can acquire the read or write lock until the write lock is released.

There are Lock readLock() and Lock wirteLock() as ReadWriteLock access methods to the read-write lock.

Lock readLock() returns the read lock associated with ReadWriteLock.

Lock wirteLock() returns the write lock associated with ReadWriteLock.

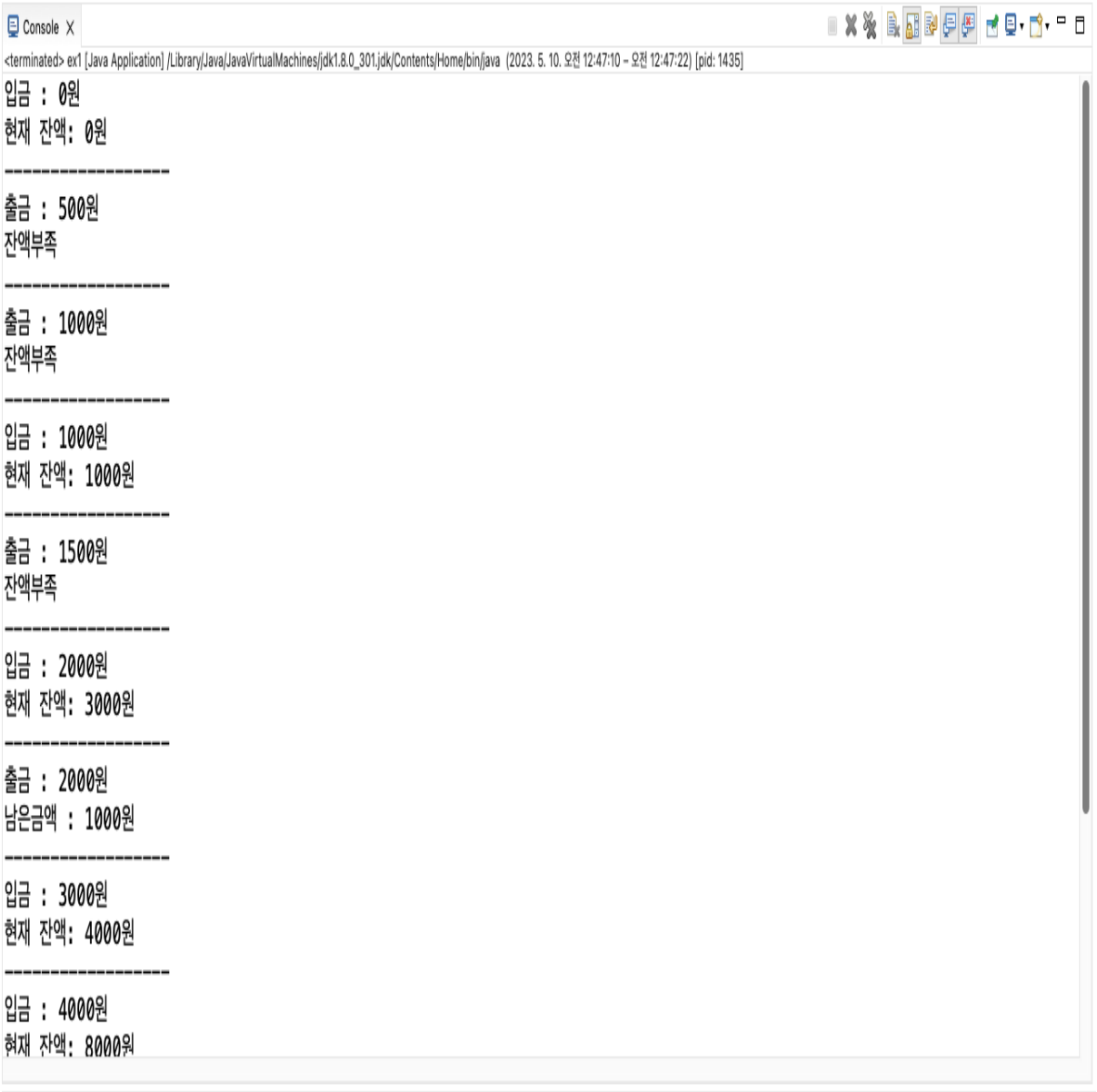
An implementation of the ReadWriteLock interface is provided by the java.util.concurrent.locks.ReentrantReadWriteLock class. A single thread can acquire a read lock or write lock multiple times. It also provides a fairness policy that can be configured to control the order in which threads are granted access to locks.

(ii)
ReadWriteLock
Source code

```
package Prob3_2;

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

public class ex2 {
	public static void main(String[] args) {
		BankAccount account = new BankAccount();

		Thread deposit = new Thread(new Deposit(account));
		Thread withdraw = new Thread(new Withdraw(account));

		deposit.start();
		withdraw.start();
	}
}

class BankAccount {
	private int balance = 0;
	private ReadWriteLock lock = new ReentrantReadWriteLock();

	public void deposit(int amount) {
		lock.writeLock().lock();
		try {
			balance += amount;
			System.out.println("입금: " + amount + "원");
			System.out.println("현재 잔액: " + balance + "원");
			System.out.println("------------------");
		} finally {
			lock.writeLock().unlock();
		}
	}

	public void withdraw(int amount) {
		lock.writeLock().lock();
		try {
			int currentBalance = getBalance();
			System.out.println("출금: " + amount + "원");
			if (currentBalance >= amount) {
				balance -= amount;
				System.out.println("남은 금액: " + (currentBalance - amount) + "원");
				System.out.println("------------------");
			} else {
				System.out.println("잔액부족");
				System.out.println("------------------");
			}
		} finally {
			lock.writeLock().unlock();
		}
	}

	public int getBalance() {
		lock.readLock().lock();
		try {
			return balance;
```

```
            } finally {
                lock.readLock().unlock();
            }
        }
    }

    class Deposit implements Runnable {
        private final BankAccount account;

        Deposit(BankAccount account) {
            this.account = account;
        }

        @Override
        public void run() {
            for (int i = 0; i <= 10; i++) {
                try {
                    account.deposit(i * 1000);
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    class Withdraw implements Runnable {
        private final BankAccount account;

        Withdraw(BankAccount account) {
            this.account = account;
        }

        @Override
        public void run() {
            for (int i = 1; i <= 10; i++) {
                try {
                    account.withdraw(i * 500);
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

(ii)
ReadWriteLock
Result


Console
<terminated> ex1 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_301.jdk/Contents/Home/bin/java (2023. 5. 10. 오전 12:47:10 – 오전 12:47:22) [pid: 1435]
입금 : 0원
현재 잔액: 0원

출금 : 500원
잔액부족

출금 : 1000원
잔액부족

입금 : 1000원
현재 잔액: 1000원

출금 : 1500원
잔액부족

입금 : 2000원
현재 잔액: 3000원

출금 : 2000원
남은금액 : 1000원

입금 : 3000원
현재 잔액: 4000원

입금 : 4000원
현재 잔액: 8000원

(iii)
AtomicInteger
explanation

AtomicIneger provides an atomically updatable integer value. Ensures thr ead safety during concurrent operations. This is useful when you need to perform atomic operations on integer values without using synchroni zation constructs such as locks or synchronized blocks.

Atomic operations are executed to avoid conditions that can occur whe n multiple threads access shared data concurrently. AtomicIntger provide s methods for performing common arithmetic and comparison operation s on integer values.

AtomicInteger(), AtomicInteger(int initialValue), int get(), void set(int new Value), int getAndIntcrement(), int getAndDecrement(), int getAndAdd(), i nt addAndGet(), int incrementAndGet(), int There is decrementAndGet().

int get() returns the current value.

void set(int newValue) sets the value to the given value.

int addAndGet() atomically adds the given value to the current value a nd returns the new value.

int getAndAdd() atomically adds the given value to the current value an d returns the previous value.

(iii)
AtomicInteger
Source code

```
package Prob3_3;

import java.util.concurrent.atomic.AtomicInteger;

public class ex3 {
	public static void main(String[] args) {
		Counter counter = new Counter();

		Thread increment = new Thread(new Increment(counter));
		Thread decrement = new Thread(new Decrement(counter));

		increment.start();
		decrement.start();

		try {
			increment.join();
			decrement.join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println("-----------");
		System.out.println("최종 카운터 값: " + counter.getValue());
	}
}

class Counter {
	private AtomicInteger value = new AtomicInteger(0);

	public void increment() {
		value.addAndGet(1);
	}

	public void decrement() {
		value.getAndAdd(-1);
	}

	public void setValue(int newValue) {
		value.set(newValue);
	}

	public int getValue() {
		return value.get();
	}
}

class Increment implements Runnable {
	private final Counter counter;

	Increment(Counter counter) {
		this.counter = counter;
	}

	@Override
	public void run() {
		for (int i = 0; i < 10; i++) {
			counter.increment();
```

```
                System.out.println("증가: " + counter.getValue());
            }
        }
}

class Decrement implements Runnable {
        private final Counter counter;

        Decrement(Counter counter) {
            this.counter = counter;
        }

        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                counter.decrement();
                System.out.println("감소: " + counter.getValue());
            }
        }
}
```

(iii)
AtomicInteger
result


Console
<terminated> ex3 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_301.jdk/Contents/Home/bin/java (2023. 5. 10. 오전 12:48:57 – 오전 12:48:58) [pid: 1450]
증가: 0
감소: 0
증가: 1
감소: 0
감소: 0
증가: 1
증가: 0
증가: 1
증가: 2
증가: 3
증가: 4
증가: 5
증가: 6
감소: -1
감소: 5
감소: 4
감소: 3
감소: 2
감소: 1
감소: 0

최종 카운터 값: 0

(iiii)
CyclicBarrier
explanation

Using Cyclic, you can wait at a desired point inside threads that are running simultaneously, and release the wait when all threads enter the waiting state. If N threads are running, N is given as an argument value when creating CyclicBarrier. When CyclicBarrier's await() is called within each thread and the number reaches N times, the wait state of all N threads is released. This is useful when multiple threads are doing work in parallel and you need to wait for all threads to complete before proceeding to the next step.
Methods of the CyclicBarrier class include int await() and int await(long timeout, TimeUnit unit).

The int await() method waits until all parties have reached the barrier and then proceeds. This method may throw InterruptedException or BrokenBarrierException.

The int await(long timeout, TimeUnit unit) method waits until all parties reach the barrier or the specified time has elapsed before proceeding. This method can also throw InterruptedException, BrokenBarrierException, and TimeoutException.

(iiii)
CyclicBarrier
Source code

```
package Prob3_4;

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

public class ex4 {
	public static void main(String[] args) {
		int numberOfThreads = 5;
		CyclicBarrier barrier = new CyclicBarrier(numberOfThreads, () -> System.out.println("All threads have
arrived"));

		for (int i = 0; i < numberOfThreads; i++) {
			new Thread(new Worker(barrier, i)).start();
		}
	}
}

class Worker implements Runnable {
	private final CyclicBarrier barrier;
	private final int id;

	Worker(CyclicBarrier barrier, int id) {
		this.barrier = barrier;
		this.id = id;
	}

	@Override
	public void run() {
		try {
			System.out.println("Thread " + id + " start!");
			Thread.sleep((long) (Math.random() * 3000));

			System.out.println("Thread " + id + " finish, Arrive at the barrier");
			barrier.await();

			System.out.println("Thread " + id + " -> Pass");
		} catch (InterruptedException | BrokenBarrierException e) {
			e.printStackTrace();
		}
	}
}
```

(iiii)
CyclicBarrier
result


Console X
<terminated> ex4 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_301.jdk/Contents/Home/bin/java (2023. 5. 10. 오전 12:49:59 – 오전 12:50:02) [pid: 1458]
Thread 0 start!
Thread 2 start!
Thread 1 start!
Thread 3 start!
Thread 4 start!
Thread 0 finish, Arrive at the barrier
Thread 1 finish, Arrive at the barrier
Thread 4 finish, Arrive at the barrier
Thread 2 finish, Arrive at the barrier
Thread 3 finish, Arrive at the barrier
All threads have arrived
Thread 3 -> Pass
Thread 0 -> Pass
Thread 1 -> Pass
Thread 2 -> Pass
Thread 4 -> Pass