**Report**

# ex1.java

## BlockingQueue

BlockingQueue is a special kind of queue (like the queue in subway) that is most used in multi-threaded programming . we can get access to it through java.util.concurrent package ,this helps that threads will be used at the same time without messing up things . basically we can consider it as a box , with some space , when we wanna add sth into it and its full , we have to wait until sth comes out first , and opposite .it has 4 different methods for inserting and removing the elements , mostly used it probably put() and take() method (Blocks).

## ArrayBlockingQueue

ArrayBlockingQueue is a implementation of BlockingQueue , the difference between them is that in ArrayBlockingQueue data or elements will be save in a fixed size array , imagine a parking with 7 parking spaces , so fixed space is 7 parking spaces , can decrease or increase . how ever in BlockingQueue we could have unlimited amount of elements because nothing was stopping it . here there is first in , first out rule , which basically means that first elements that get out of array is the one that was in array the longest.

Maybe having some example can help us understand the usage of these a little better

an example of using BlockingQueue and ArrayBlockingQueue in actual code :

we have 2 thread for procucer and users here , in main class we just run them all . basically producer are putting element in queue and users are removing them (take)

package prob3;

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.ArrayBlockingQueue;

public class ex1 {

    public static *void* main(String[] *args*) {

        BlockingQueue<String> queue = new ArrayBlockingQueue<>(3); //cap = 3

        Producer producer = new Producer(queue);

        User User = new User(queue);

        producer.start();

        User.start();

    }

}

// THREADS------------------------------------------------------------------

class Producer extends Thread {

    private BlockingQueue<String> queue;

    public Producer(BlockingQueue<String> *queue*) {

        this.queue = *queue*;

    }

    public *void* run() {

        try {

            for (*int* i = 1; i <= 5; i++) {

                System.out.println("Provider:        trying to put item " + i);

                queue.put("Item " + i);  // waits if queue is full

                System.out.println("Provider:        put item " + i);

                Thread.sleep(1000); // simulate some delay

            }

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

}

class User extends Thread {

    private BlockingQueue<String> queue;

    public User(BlockingQueue<String> *queue*) {

        this.queue = *queue*;

    }

    public *void* run() {

        try {

            for (*int* i = 1; i <= 5; i++) {

                Thread.sleep(1000); // wait before taking

                String item = queue.take(); // waits if queue is empty

                System.out.println("User :           took " + item);

            }

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

}

# ex2.java

**ReadWriteLock** helps is java multi-threading to avoid the race condition or complex error which are cause of 2 threads working in the same thing , basically ReadWriteLock make sure if a thread is reading or writing an element , others will wait until the thread is finished and can access it after its unlocked. This example code can help to understand how it works , to say in a simple way , there are people reading the books and people who are editing , so the point is that while someone is writing sth other people cant unlock it to , read it to avoid the problems.

package prob3;

import java.util.concurrent.locks.ReentrantReadWriteLock;

import java.util.concurrent.locks.ReadWriteLock;

public class ex2 {

    public static *void* main(String[] *args*) {

        Library library = new Library();

        // these threads gonna read and write the "book"

        BookReader reader1 = new BookReader(library, "Reader A");

        BookReader reader2 = new BookReader(library, "Reader B");

        BookEditor writer = new BookEditor(library, "Writer X");

        reader1.start();

        reader2.start();

        writer.start();

    }

}

// SHARED CLASS -------------------------------------------------------------------

class Library {

    private final ReadWriteLock lock = new ReentrantReadWriteLock();

    private String book = "Original Book";

    public *void* read(String *readerName*) {

        lock.readLock().lock(); // yo lemme in to read

        try {

            System.out.println(*readerName* + ":  Reading the book -> " + book);

            Thread.sleep(1000); // chill time

        } catch (InterruptedException e) {

            e.printStackTrace();

        } finally {

            lock.readLock().unlock(); // done reading bruh

            System.out.println(*readerName* + ":  Done reading");

        }

    }

    public *void* write(String *writerName*) {

        lock.writeLock().lock(); // no one else allowed, I'm editing

        try {

            System.out.println(*writerName* + ":  Writing a new version of the book...");

            Thread.sleep(2000); // writer being dramatic

            book = " New Edition by " + *writerName*;

            System.out.println(*writerName* + ":  Done writing!");

        } catch (InterruptedException e) {

            e.printStackTrace();

        } finally {

            lock.writeLock().unlock(); // let the others in

            System.out.println(*writerName* + ":  Released the write lock");

        }

    }

}

// THREADS -------------------------------------------------------------------

class BookReader extends Thread {

    private final Library library;

    private final String readerName;

    public BookReader(Library *library*, String *readerName*) {

        this.library = *library*;

        this.readerName = *readerName*;

    }

    public *void* run() {

        for (*int* i = 0; i < 2; i++) {

            library.read(readerName);

            try {

                Thread.sleep(1500); // cooldown time

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

class BookEditor extends Thread {

    private final Library library;

    private final String writerName;

    public BookEditor(Library *library*, String *writerName*) {

        this.library = *library*;

        this.writerName = *writerName*;

    }

    public *void* run() {

        for (*int* i = 0; i < 2; i++) {

            library.write(writerName);

            try {

                Thread.sleep(3000); // chill before writing again

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

# ex3.java

**AtomicInteger** is mostly used for synchronization of a value , it helps the multi-threaded programs cause that way , there will be no race conditioning , in other words AtomicInteger is prevent the race conditioning in multi-thread programming in java .

package prob3;

import java.util.concurrent.atomic.AtomicInteger;

public class ex3 {

    public static *void* main(String[] *args*) {

        SharedLikesCounter counter = new SharedLikesCounter();

        // Create 2 users clicking like and unlike

        new UserA(counter).start();

        new UserB(counter).start();

    }

}

// The Shared Counter using AtomicInteger\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

class SharedLikesCounter {

    AtomicInteger likes = new AtomicInteger(50); // start with 50 likes

    public *void* addLike(String *name*) {

*int* before = likes.getAndAdd(1); // add 1, return old value

        System.out.println(*name* + " added a like. Before: " + before + " After: " + likes.get());

    }

    public *void* removeLike(String *name*) {

*int* after = likes.addAndGet(-1); // remove 1, return new value

        System.out.println(*name* + " removed a like. New total: " + after);

    }

}

// One user that keeps liking the post\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

class UserA extends Thread {

    SharedLikesCounter counter;

    public UserA(SharedLikesCounter *counter*) {

        this.counter = *counter*;

    }

    public *void* run() {

        for (*int* i = 0; i < 3; i++) {

            counter.addLike("UserA");

            try {

                Thread.sleep(600);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

// Another user who unlikes it\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

class UserB extends Thread {

    SharedLikesCounter counter;

    public UserB(SharedLikesCounter *counter*) {

        this.counter = *counter*;

    }

    public *void* run() {

        for (*int* i = 0; i < 2; i++) {

            counter.removeLike("UserB");

            try {

                Thread.sleep(900);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

# ex4.java

**CyclicBarrier** is used for multi-threading when we need all of thread to be finished to step into next level , what it does , It waits until all threads are finished and then let the programm continue debugging , basically like a red light ! (waits for all cars to arrive and then greenlight hahaha)

package prob3;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

public class ex4 {

    public static *void* main(String[] *args*) {

        // Barrier waits for 3 players

        CyclicBarrier barrier = new CyclicBarrier(3, () *->* {

            System.out.println("All players arrived! Let’s start the game!");

        });

        // Create 3 players

        new Player("Player1", barrier).start();

        new Player("Player2", barrier).start();

        new Player("Player3", barrier).start();

    }

}

class Player extends Thread {

    private String name;

    private CyclicBarrier barrier;

    public Player(String *name*, CyclicBarrier *barrier*) {

        this.name = *name*;

        this.barrier = *barrier*;

    }

    public *void* run() {

        try {

            System.out.println(name + " is getting ready...");

            Thread.sleep((*int*)(Math.random() \* 3000));

            System.out.println(name + " is ready and waiting...");

            barrier.await();

            Thread.sleep((*int*)( 1000));

            System.out.println(name + " starts the game!");

        } catch (InterruptedException | BrokenBarrierException e) {

            e.printStackTrace();

        }

    }

}

# ex5.java

**ExecutorService** is like a manager for threads in java multi-thread programming , so basically, we give him the task and the number of thread that should work on it and it manage all the things.

        ExecutorService pool = Executors.newFixedThreadPool(4);

**Callable** is just like runnable in java the only difference is that it returns a value , for example we wanna count the number of prime numbers in a range , in this case Callable return the integer , which is the number of prime numbers in that range.

**Future** is like an element or a ticket for getting the result of computing later , means a thread is working for a result , can will return some result later , future waits for that and get the result later and give them to us.

 Future<Integer> future = pool.submit(new PrimeCounter(chunkStart, chunkEnd));

            results.add(future);

This can be very useful in some cases like in the code blow .

package prob3;

import java.util.concurrent.\*;

import java.util.\*;

public class ex5 {

    public static *void* main(String[] *args*) throws Exception {

        // Create a thread pool with 4 threads

        ExecutorService pool = Executors.newFixedThreadPool(4);

*int* start = 1;

*int* end = 200000;

*int* chunk = 50000;

        List<Future<Integer>> results = new ArrayList<>();

        // Split the work into 4 tasks

        for (*int* i = start; i < end; i += chunk) {

*int* chunkStart = i;

*int* chunkEnd = Math.min(i + chunk, end);

            Future<Integer> future = pool.submit(new PrimeCounter(chunkStart, chunkEnd));

            results.add(future);

        }

        // Collect the results

*int* totalPrimes = 0;

        for (Future<Integer> result : results) {

            totalPrimes += result.get(); // Wait for each task

        }

        System.out.println("Total prime numbers between 1 and 200000: " + totalPrimes);

        pool.shutdown(); // Stop the thread pool

    }

}

// The Callable task that counts primes in a range

class PrimeCounter implements Callable<Integer> {

    private *int* start, end;

    public PrimeCounter(*int* *start*, *int* *end*) {

        this.start = *start*;

        this.end = *end*;

    }

    @*Override*

    public Integer call() {

*int* count = 0;

        for (*int* i = start; i <= end; i++) {

            if (isPrime(i)) count++;

        }

        System.out.println(Thread.currentThread().getName() + " counted " + count + " primes from " + start + " to " + end);

        return count;

    }

    private *boolean* isPrime(*int* *n*) {

        if (*n* <= 1) return false;

        for (*int* i = 2; i <= Math.sqrt(*n*); i++) {

            if (*n* % i == 0) return false;

        }

        return true;

    }

}