BlockingQueue

Blocklanadigna navbat

BlockingQueue 1

- The *Java BlockingQueue* interface, java.util.concurrent.BlockingQueue, represents a queue which is thread safe to put elements into, and take elements out of from. In other words, multiple threads can be inserting and taking elements concurrently from a Java BlockingQueue, without any concurrency issues arising.
- BlockingQueue one of the most useful constructs *java.util.concurrent* to solve the concurrent producer-consumer problem.

- BlockingQueue java.util.concurrent package da joylashgan va producer-consumer muommosini hal qiladigan eng foydali class Interface lardan biri.
- Java da BlockingQueue bu interface. U java.util.concurrent.BlockingQueue package da joylashgan. BlockingQueue parallel dasturlashda elementlarni qo'yish va undan elementlarni olish uchun xavfsiz bo'lgan navbatni tagdim etadi. Boshqa so’zlar bilan aytsak, bir vaqtning o’zda bir nechta Thread navbatga element qo’shish va undan element olishi mumkin va bnda hech qanday parallellik bilan bog'liq muammolar bo’lmaydi.

BlockingQueue 2

- The term *blocking queue* comes from the fact that the Java BlockingQueue is capable of blocking the threads that try to insert or take elements from the queue. For instance, if a thread tries to take an element and there are none left in the queue, the thread can be blocked until there is an element to take. Whether or not the calling thread is blocked depends on what methods you call on the BlockingQueue.

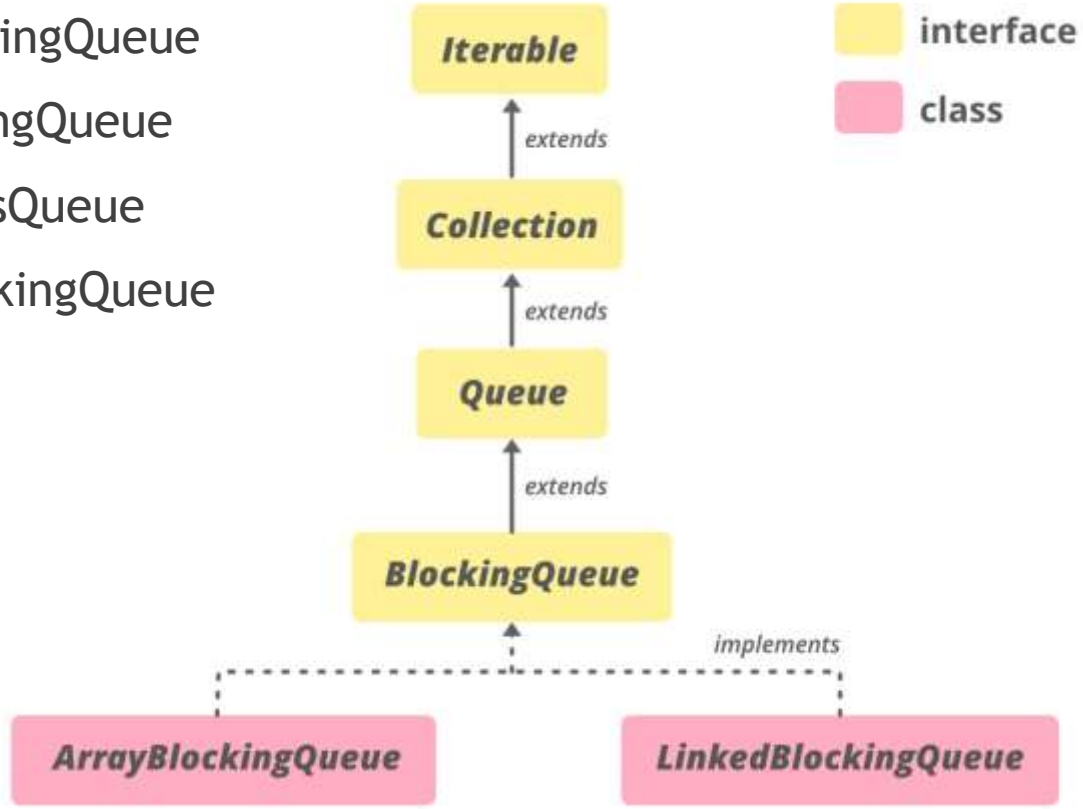
- Blocking queue deb atalishining sababi Javada BlockingQueue queue ga elemetn qo'shayotgan yoki undan element olayotgan Thread ni blocklash qobiliyatiga egadir. Malasan , agar Thread queue dan elementni olmoqchi bo'lsa va queue da element bo'lmasa shu Thread queue da element paydo bo'lmagunicha u block lanishi mumkin. Elementni olayotganda Thread block ga tushadimi yo'qmi bu BlockingQueue ninq qaysi methodini shqirishimizga bog'liq.

BlockingQueue Implementations classes

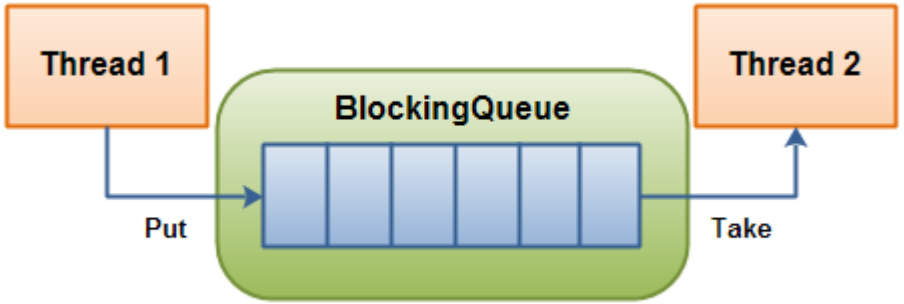
- Since BlockingQueue is an interface, you need to use one of its implementations to use it. The java.util.concurrent package has the following implementations of the BlockingQueue interface:
- BlockingQueue interface bo'lgani sababli siz undan implements olgan class lardan bittasini ishlatishingiz kerak. java.util.concurrent package da BlockingQueue ni implements qilgan quyidagi class lar taqdim etilgan:
  - ArrayBlockingQueue
  - DelayQueue
  - LinkedBlockingQueue
  - LinkedBlockingDeque
  - LinkedTransferQueue
  - PriorityBlockingQueue
  - SynchronousQueue

The Hierarchy of BlockingQueue

- LinkedBlockingQueue
- ArrayBlockingQueue
- SynchronousQueue
- PriorityBlockingQueue



BlockingQueue Usage 1



- A BlockingQueue with one thread putting into it, and another thread taking from it.
- .

BlockingQueue Usage 2

- The producing thread will keep producing new objects and insert them into the BlockingQueue, until the queue reaches some upper bound on what it can contain. It's limit, in other words. If the blocking queue reaches its upper limit, the producing thread is blocked while trying to insert the new object. It remains blocked until a consuming thread takes an object out of the queue.
- The consuming thread keeps taking objects out of the BlockingQueue to processes them. If the consuming thread tries to take an object out of an empty queue, the consuming thread is blocked until a producing thread puts an object into the queue.
- Queue to’lmagunichia Producer Thread ob’ektlarni ishlab chiqarib queue ga qo’shishni davom ettiraveradi. Agar queue to’lsa Producer Thread yangi o’bekt qo’shayotganda blockga tushadi. Consumer Thread queue dan element olmagunicha u block da bo’ladi.
- Cosumer Thread BlockingQueue dan ob’ektlarni ishlash uchun oladi. Agar bosh queue dan Consumer Thread ob’ektni olmoqchi bo’lsa, Consumer Thread queue da element paydo bo’lgunicha block ga tushadi.

BlockingQueue Methods

- The Java BlockingQueue interface has 4 different sets of methods for inserting, removing and examining the elements in the queue. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:
- BlockingQueue da element qo'shish, elemetni remove qilish, elementi olish uchun 4 xilsagi methodlar yaratgan.

	Throws Exception	**Special Value**	**Blocks**	**Times Out**
Insert	add(o)	offer(o)	put(o)	offer(o, timeout, timeunit)
Remove	remove(o)	poll()	take()	poll(timeout, timeunit)
Examine	element()	peek()		

BlockingQueue : 4 different sets

- **Throws Exception**:
  If the attempted operation is not possible immediately, an exception is thrown.
- **Special Value**:
  If the attempted operation is not possible immediately, a special value is returned (often true / false).
- **Blocks**:
  If the attempted operation is not possible immedidately, the method call blocks until it is.
- **Times Out**:
  If the attempted operation is not possible immedidately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

BlockingQueue Types

- We can distinguish two types of *BlockingQueue*:
  - unbounded queue – can grow almost indefinitely
  - bounded queue – with maximal capacity defined
- *BlockingQueue ni ikkita turga ajratish mumkin*
  - unbounded queue (chegaralanmagan navbat) – cheksiz tarzda o'sishi mumkin.
  - bounded queue (chegaralangan navbat) – maksimal hajim ko'rsatilgan bo'ladi.
-

Unbounded Queue

- Creating unbounded queues is simple (Chegaralanmanga navbat yaratish juda oson)

```
BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>();
```

- The Capacity of *blockingQueue* will be set to *Integer.MAX_VALUE*. All operations that add an element to the unbounded queue will never block, thus it could grow to a very large size.
- The most important thing when designing a producer-consumer program using unbounded BlockingQueue is that consumers should be able to consume messages as quickly as producers are adding messages to the queue. Otherwise, the memory could fill up and we would get an *OutOfMemory* exception.
- *BlockingQueue ning hajmi Integer.MAX_VALUE ning qiymatiga teng bo'ladi. Chegaralanmagna aueue ga element qo'shadigan barcha amallar (methodlar) blocklanmaydi, Shu sababdan bu juda katta hajmga chiqishi mumkin.*
- Producer-Consumer dasturini unbounded BlockingQueue bilan qilayotgandan consumer ning tezligi producer ning tezligiday bo'lishiga ahamiyat berishimiz kerak. Bo'lmasa Hotira to'lib qolib OutOfMemory istisnosi kelib chiqishi mumkin.

Bounded Queue 1

- The second type of queues is the bounded queue. We can create such queues by passing the capacity as an argument to a constructor:

```
BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>(10);
```

- Here we have a *blockingQueue* that has a capacity equal to 10. It means that when a producer tries to add an element to an already full queue, depending on a method that was used to add it (*offer()*, *add()* or *put()*), it will block until space for inserting object becomes available. Otherwise, the operations will fail.
- Ikkinchi turdagi queue bu bounded (chegaralangan) queue dir. Biz bunday queue ni hajmni constructor ga berib yubir orqali yaratishimiz mumkin.
- Bizda hajmi 10 ga teng bo'lgan blockigQueue yaratildi. Bu degani Producer allachagon to'lgan queue ga element qo'shmoqchi bo'lsa (offer,add,put) methodlari orqali, u qo'shish uchun joy paydo bo'lguncha blocklanadi. Aks holsa amal ishlamaydi.

Bounded Queue 2

- Using bounded queue is a good way to design concurrent programs because when we insert an element to an already full queue, that operations need to wait until consumers catch up and make some space available in the queue. It gives us throttling without any effort on our part.

- Paraller dasturlarda bounded queue dan foydalanish juda yaxshidir. Sababi to’lgan navbatga element qo’shilmoqchi bo’lganda qo’shmoqchi bo’lgan Thread, toki consumer queue dan element ni olib joy yaratmaguncha kutub turadi. Consumer queue dan elementni olib joy yaratsa producer yana elementlarni queue da qo’shib ketaveradi. Bu bizda ko’p kuch sarflamasdan ishlash imkonini beradi.

BlockingQueue API

- There are two types of methods in the *BlockingQueue* interface – methods responsible for adding elements to a queue and methods that retrieve those elements. Each method from those two groups behaves differently in case the queue is full/empty.
- *BlockingQueue interface da queue ga element qo'shadigan va queue dan element oladigan methodlar bor. Bu ikkita guruhdagi methodlar queue ni bo'sh/to'lgan ekanligiga qarab harhil ishlaydi.*

BlockingQueue Adding Elements

- *Element qo'shish:*
- *add()* – returns *true* if insertion was successful, otherwise throws an *IllegalStateException. (Elelemtn qo'shish mofaqiyatli bo'lsa true return qiladi bo'lmasa IllegalStateException tashaydi).*
- *put()* – inserts the specified element into a queue, waiting for a free slot if necessary. (Element ni queue ga qo'shadi, agar bosh joy bo'lmasa kutub turadi.)
- *offer()* – returns *true* if insertion was successful, otherwise *false*. *(Agar qo'shish mafaqiyatli bo'lsa true return qiladi, bo'lmasa false).*
- *offer(E e, long timeout, TimeUnit unit)* – tries to insert element into a queue and waits for an available slot within a specified timeout. (Navbatga element qo'shishga harata qiladi, agar joy bo'lmasa berilgan baqt maboynida kutib turadi)

BlockingQueue Retrieving Elements

- *take()* – waits for a head element of a queue and removes it. If the queue is empty, it blocks and waits for an element to become available. (Navbatni boshida turgna elementni olib tashlaydi. Agar Queue bosh bo'lsa element paydo bo'lguncha blockda turadi.)
- *poll(long timeout, TimeUnit unit)* – retrieves and removes the head of the queue, waiting up to the specified wait time if necessary for an element to become available. Returns *null* after a timeout. (Navbatdagi elemetni olib tashlaydi. Agar element bo'lmasa berilgan vaqt maboynida kutib turadi. Vaqt o'tguncha element payda bo'lmasa null return qiladi.)

BlockingQueue Example 1

```
public class NumbersProducer implements Runnable {
    private BlockingQueue<Integer> queue;

    public NumbersProducer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            int n = ThreadLocalRandom.current().nextInt(100);
            System.out.println("Producer: " + n);
                                queue.put(n);
        }

    }
}
```

- Producer Thread

BlockingQueue Example 2

- Consumer Thread

```
public class NumbersConsumer implements Runnable {
    private BlockingQueue<Integer> queue;

    public NumbersConsumer(BlockingQueue<Integer> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            Integer number = queue.take();
            System.out.println(" Consuming : " + number);
        }
    }
}
```

BlockingQueue Example 3

- Main Thread

```
public static void main(String[] args) {
        BlockingQueue<Integer> queue = new LinkedBlockingDeque<>();
        new Thread(new NumbersProducer(queue)).start();
        new Thread(new NumbersConsumer(queue)).start();
    }
```

BlockingQueue Links

- https://jenkov.com/tutorials/java-util-concurrent/blockingqueue.html
- https://www.baeldung.com/java-blocking-queue
- https://www.geeksforgeeks.org/blockingqueue-interface-in-java/
- https://www.journaldev.com/1034/java-blockingqueue-example

SynchronousQueue

SynchronousQueue 1

- **SynchronousQueue** allows us to exchange information between threads in a thread-safe manner.
- The *SynchronousQueue* only has **two supported operations: *take()* and *put()*, and both of them are blocking**.
- For example, when we want to add an element to the queue, we need to call the *put()* method. That method will block until some other thread calls the *take()* method, signaling that it is ready to take an element.
- **SynchronousQueue Thread lar o'rtasida xafsiz tarzda malumot almashish imkonini beradi.**
- SynchronousQueue ikkita amalni taqdim etadi. take() va put(), bular ikkalasi ham block lash hususiyatiga egadir.
- **Masalan: navbatga element qo'shmoqchi bo'lsak biz put() methodni chaqiramiz. Endi boshqa Thread take() methodni chaqurmagunicha put() method block ga tushadi, navbatdan elementni olish mumkin degan manoda.**

SynchronousQueue 2

- Although the *SynchronousQueue* has an interface of a queue, we should think about it as an exchange point for a single element between two threads, in which one thread is handing off an element, and another thread is taking that element.

- *Bundan tashqari SynchronousQueue* Queue interface dan implements olgan. Biz buni 2ta Thread o’rtasida ob’ekt almashadigan nuqta sifatida qarashimiz kerak, bunda bitta Thread elementni berib yuboradi ikkinchi Thread uni qabul qiladi.

SynchronousQueue 3

- The SynchronousQueue is a queue that can be used to exchange a single element with another thread. A thread inserting an element into the queue is blocked until another thread takes that element from the queue. Likewise, if a thread tries to take an element and no element is currently present, that thread is blocked until a thread insert an element into the queue.
- Calling this class a queue is a bit of an overstatement. It's more of a rendesvouz point.
- SynchronousQueue bu queue bo’lib u bitta elementi boshqa Thread bilan almashish uchun ishlatiladi. Queue ga element qo’shgan Thread boshqa bitta Thread queue dan elementni olgunucha block lanadi. Shu kabi, Agar Thread elemeni olmochi bo’lsa va queue da element bo’lmasa u blocklanadi toki boshqa Thread elementni add qilgunicha.
- Bu Class ni queue deb atash biroz ortiqcha gap. Bu ko'proq uchrashuv nuqtasi dir.

SynchronousQueue 4

- Exchanger: Is a rendezvous point where two threads can exchange objects
- SynchQueue: Is a queue! One thread puts and waits, until another thread pops.. there is no exchange business here
- Exchanger is a kind of bidirectional SyncQueue

SynchronousQueue Example 1

- Producer Thread

```
public class Producer extends Thread {
    private SynchronousQueue<Integer> synchronousQueue;

    public Producer(SynchronousQueue<Integer> s) {
        this.synchronousQueue = s;
    }

    @Override
    public void run() {
        Thread.sleep(2000);
        System.out.println("Producing: 1");
        synchronousQueue.put(1);

                        System.out.println("Producing: 2");
        synchronousQueue.put(2);
    }
}
```

SynchronousQueue Example 2

- Producer Thread

```
public class Consumer extends Thread {

    private SynchronousQueue<Integer> synchronousQueue;

    public Consumer(SynchronousQueue<Integer> s) {
        this.synchronousQueue = s;
    }

    @Override
    public void run() {
        System.out.println("Consuming: " + synchronousQueue.take());
        System.out.println("Consuming: " + synchronousQueue.take());
    }
}
```

SynchronousQueue Example 3

- Main

```
public class SynchronousQueueDemoMain {
    public static void main(String[] args) {
        SynchronousQueue<Integer> synchronousQueue = new SynchronousQueue<Integer>();

        Producer producer = new Producer(synchronousQueue);
        Consumer consumer = new Consumer(synchronousQueue);

        producer.start();
        consumer.start();
    }
}
```

SynchronousQueue Links

- https://jenkov.com/tutorials/java-util-concurrent/synchronousqueue.html
- https://www.geeksforgeeks.org/java-program-to-implement-synchronousqueue-api/
- https://www.baeldung.com/java-synchronous-queue
- https://jenkov.com/tutorials/java-util-concurrent/synchronousqueue.html