Queue?

Queue means ‘waiting line’, which is very similar to queues in real life: a queue of people standing in an airport’s checkin gate; a queue of cars waiting for green light in a road in the city; a queue of customers waiting to be served in a bank’s counter, etc.

In programming, queue is a data structure that holds elements prior to processing, similar to queues in real-life scenarios. Let’s consider a queue holds a list of waiting customers in a bank’s counter. Each customer is served one after another, follow the order they appear or registered.

The first customer comes is served first, and after him is the 2nd, the 3rd, and so on. When serving a customer is done, he or she lefts the counter (removed from the queue), and the next customer is picked to be served next. Other customers come later are added to the end of the queue. This processing is called First In First Out or FIFO.

During the processing, the queue can be dynamically changed, i.e. processed elements are removed from the queue, and new elements are added to the queue.

In the Java Collections Framework, Queue is the main interface, and there are four sub interfaces: Deque, BlockingDeque, BlockingQueue, and TransferQueue.

Except the Deque interface which is in the java.util package, all others are organized in the java.util.concurrent package, which is designed for multi-threading or concurrent programming.

Characteristics of Queue:

Basically, a queue has a head and a tail. New elements are added to the tail, and to-be-processed elements are picked from the head. The following picture illustrates a typical queue:

A black and white text

Description automatically generated with medium confidence

Elements in the queue are maintained by their insertion order. The Queue interface abstracts this kind of queue.

Another kind of queue is double ended queue, or deque. A deque has two heads, allowing elements to be added or removed from both ends. The following picture illustrates this kind of queue:

A number of numbers in a row

Description automatically generated with medium confidence

The Deque interface abstracts this kind of queue, and it is a sub interface of the Queue interface. And the LinkedList class is a well-known implementation. Some implementations accept null elements, some do not.

Queue does allow duplicate elements, because the primary characteristic of queue is maintaining elements by their insertion order. Duplicate elements in terms of equals contract are considered distinct in terms of queue, as there is no two elements having same ordering.

Additionally, the Java Collection Framework provides the BlockingQueue interface that abstracts queues which can be used in concurrent (multi-threading) context.

A blocking queue waits for the queue to become non-empty when retrieving an element, and waits for space become available in the queue when storing an element.

Similarly, the BlockingDeque interface is blocking queue for double ended queues.

Behaviors of Queue:

Due to Queue’s nature, the key operations that differentiate Queue from other collections are extraction and inspection at the head of the queue.

For deques, the extraction and inspection can be processed on both ends.

And because the Queue interface extends the Collection interface, all Queue implementations provide core operations of a collection like add(), contains(),remove(), clear(), isEmpty(), etc.

And keep in mind that, with queues, operations on the head are fastest (e.g. offer() and remove()), whereas operations on middle elements are slow (e.g.contains(obj) and remove(obj)).

Queue’s Interfaces:

Queue is the super interface of the queue branch in the Java Collection Framework. Under it, there are the following sub interfaces:

* **Deque**: abstracts a queue that has two heads. A deque allows adding or removing elements at both ends.
* **BlockingQueue**: abstracts a type of queues that waits for the queue to be non-empty when retrieving an element, and waits for space to become available in the queue when storing an element.
* **BlockingDeque**: is similar to BlockingQueue, but for double ended queues. It is sub interface of the BlockingQueue.

And since Java 7, the BlockingQueue interface has a new sub interface called TransferQueue, which is a specialized BlockingQueue, which waits for another thread to retrieve an element in the queue.

Major Queue’s Implementations:

The Java Collection framework provides many implementations, mostly for the BlockingQueue interface. Below I name few which are used commonly.

Queue implementations are grouped into two groups: general-purpose and concurrent implementations.

-General-purpose Queue implementations:

* + **LinkedList**: this class implements both List and Deque interface, thus having hybrid characteristics and behaviors of list and queue. Consider using a LinkedList when you want fast adding and fast removing elements at both ends, plus accessing elements by index.

* + **PriorityQueue**: this queue orders elements according to their natural ordering, or by a Comparator provided at construction time. Consider using a PriorityQueue when you want to take advantages of natural ordering and fast adding elements to the tail and fast removing elements at the head of the queue.

* + **ArrayDeque**: a simple implementation of the Deque interface. Consider using an ArrayDeque when you want to utilize features of a double ended queue without list-based ones (simpler than a LinkedList).

Concurrent Queue implementations:

* + **ArrayBlockingQueue**: this is a blocking queue backed by an array. Consider using an ArrayBlockingQu0eue when you want to use a simple blocking queue that has limited capacity (bounded).

* + **PriorityBlockingQueue**: Use this class when you want to take advantages of both PriorityQueue and BlockingQueue.

* + **DelayQueue**: a time-based scheduling blocking queue. Elements added to this queue must implement the Delayed interface. That means an element can only be taken from the head of the queue when its delay has expired.

# Mastering Queue Collection in Java

## Queue API Structure

Before giving you code examples about using Queue collections, I think it’s necessary to understand deeper about the API Structure of Queue collection, as queues have very different characteristics than other types of collection.

Because the Queue interface extends the Collection interface, al Queue implementations have basic operations of a collection:

* Single operations: add(e),contains(e), iterator(), clear(), isEmpty(), size() and toArray().
* Bulk operations: addAll(), containsAll(), removeAll() and retainAll().

Understanding Queue interface’s API Structure:

Basically, Queue provides three primary types of operations which differentiate a queue from others:

1. **Insert**: adds an element to the tail of the queue.
2. **Remove**: removes the element at the head of the queue.
3. **Examine**: returns, but does not remove, the element at the head of the queue. And for each type of operation, there are two versions:

▪ The first version throws an exception if the operation fails, e.g. could not add element when the queue is full. ▪ The second version returns a special value (either null or false, depending on the operation).

The following table summarizes the main operations of the Queue interface:

|  |  |  |
| --- | --- | --- |
| **Type of operation** | **Throws exception** | **Returns special value** |
| Insert | add(e) | offer(e) |
| Remove | remove() | poll() |
| Examine | element() | peek() |

Understanding Deque interface’s API Structure:

As you know, the Deque interface abstracts a double ended queue with two ends (first and last), so its API is structured around this characteristic.

A Deque implementation provides the xxxFirst() methods that operate on the first element, and the xxxLast() methods that operate on the last element. The following table summarizes the API structure of Deque:

|  |  |  |
| --- | --- | --- |
| **Type of operation** | **First element** | **Last element** |
| Insert | addFirst(e) offerFirst(e) | addLast(e) offerLast(e) |
| Remove | removeFirst() pollFirst() | removeLast() pollLast() |
| Examine | getFirst() peekFirst() | getLast() peekLast() |

Understanding BlockingQueue interface’s API Structure:

A blocking queue is designed to wait for the queue to become non-empty when retrieving an element (the put(e) method), and wait for space to become available in the queue when storing an element (the take() method).

In addition, a blocking queue provides specialized operations that can wait up to a specified duration when inserting and removing an element.

The following table summarizes the API structure of BlockingQueue interface:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type of operation** | **Throws exception** | **special value** | **blocks** | **times out** |
| Insert | add(e) | offer(e) | put(e) | offer(e, time, unit) |
| Remove | remove() | poll() | take() | poll(time, unit) |
| Examine | element() | peek() | N/A | N/A |

Understanding BlockingDeque interface’s API Structure:

Similarly, a BlockingDeque is a specialized BlockingQueue for double ended queue with two ends (head and tail). Its API is in scheme of xxxFirst() methods operating on the first element and xxxLast() methods operating on the last element.

The following table summarizes the API structure of BlockingDeque:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **First Element (head)** | |  |  |  |
|  | Throws exception | special value | blocks | times out |
| **Insert** | addFirst(e) | offerFirst(e) | putFirst(e) | offerFirst(e, time, unit) |
| **Remove** | removeFirst() | pollFirst() | takeFirst() | pollFirst(time, unit) |
| **Examine** | getFirst() | peekFirst() | N/A | N/A |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Last Element (tail)** |  |  |  |  |
|  | Throws exception | special value | blocks | times out |
| **Insert** | addLast(e) | offerLast(e) | putLast(e) | offerLast(e, time, unit) |
| **Remove** | removeLast() | pollLast() | takeLast() | pollLast(time, unit) |
| **Examine** | getLast() | peekLast() | N/A | N/A |

# Performing Operations on Queue Collection

Let’s go through various code examples to understand how to use Queue collections in daily coding. In the upcoming examples, I use different implementations like LinkedList, ArrayDeque, PriorityQueue,ArrayBlockingQueue, etc.

# 1. Creating a New Queue Instance

As a best practice, it’s recommended to use generic type and interface as reference type when creating a new collection. For queues, depending on the need of a particular type (queue, deque and blocking queue), use the corresponding interface as the reference type.

For example, the following statements create 3 different types of queues:

Queue<String> namesQueue = new LinkedList<>();

Deque<Integer> numbersDeque = new ArrayDeque<>();

BlockingQueue<String> waitingCustomers = new ArrayBlockingQueue<>(100);

Most Queue implementations do not have restriction on capacity (unbounded queues), except

the ArrayBlockingQueue, LinkedBlockingQueue and LinkedBlockingDeque classes. The following statement creates an ArrayBlockingQueue with fixed capacity of 200 elements:

BlockingQueue<String> waitingCustomers = new ArrayBlockingQueue<>(200);

Also remember that we can use the copy constructor to create a new Queue instance from another collection. For example:

List<String> listNames = Arrays.asList("Alice", "Bob", "Cole", "Dale", "Eric", "Frank"); Queue<String> queueNames = new LinkedList<>(listNames); System.out.println(queueNames); Output:

[Alice, Bob, Cole, Dale, Eric, Frank]

# 2. Adding New Elements to the Queue

To insert an element to the tail of the queue, we can use either the add() or offer() method. The following code adds two elements to a linked list:

Queue<String> queueNames = new LinkedList<>(); queueNames.add("Mary"); queueNames.add("John");

When using an unbounded queue (no capacity restriction), the add() and offer() methods do not show the difference. However, when using a bounded queue, the add() method will throw an exception if the queue is full, while the offer() method returns false. The following example illustrates this difference: Queue<Integer> queueNumbers = new ArrayBlockingQueue<>(3); queueNumbers.add(1); queueNumbers.add(2); queueNumbers.add(3);

queueNumbers.add(4);// this line throws exception

The last line throws java.lang.IllegalStateException: Queue full because we declare the queue with capacity of 3 elements. Hence adding the 4th element results in an exception.

However, we are safe when using the offer() method, as shown in the following code snippet:

Queue<Integer> queueNumbers = new ArrayBlockingQueue<>(3);

System.out.println(queueNumbers.offer(1));

System.out.println(queueNumbers.offer(2));

System.out.println(queueNumbers.offer(3));

System.out.println(queueNumbers.offer(4)); Output:

true true true false

The following code snippet adds an element to the head and an element to the tail of a double ended queue (notice the type of the interface is used):

Deque<String> queueNames = new ArrayDeque<>(); queueNames.offer("Katherine"); queueNames.offer("Bob"); queueNames.addFirst("Jim"); queueNames.addLast("Tom"); System.out.println(queueNames); Output:

[Jim, Katherine, Bob, Tom]

For blocking queue, use the put(e) or offer(e, time, unit) in case you want the current thread to wait until space becomes available in the queue. For example:

BlockingQueue<Integer> queueNumbers = new ArrayBlockingQueue<>(100); try {

queueNumbers.put(2000); } catch (InterruptedException ie) { ie.printStackTrace();

}

# 3. Removing the Head of the Queue

A Queue provides the remove() and poll() methods that allow us to pick the element at the head and remove it from the queue. And you should understand the difference between these two methods.

The remove() method returns the head element or throws an exception if the queue is empty. For example: Queue<String> queueCustomers = new LinkedList<>(); queueCustomers.offer("Jack");

String next = queueCustomers.remove();

System.out.println("Next customer is: "+ next); next = queueCustomers.remove(); // throws exception

Here, the queue has only one element, so the first call to remove() working fine. However the subsequent invocation results in java.util.NoSuchElementException because the queue becomes empty.

In contrast, the poll() method returns null if the queue is empty, as shown in the following example: Queue<String> queueCustomers = new LinkedList<>(); queueCustomers.offer("Jack");

System.out.println("next: " + queueCustomers.poll());

System.out.println("next: " + queueCustomers.poll()); // returns null Output:

next: Jack next: null

The following example removes the head element and tail element from a deque: Deque<String> queueCustomers = new ArrayDeque<>(); queueCustomers.offer("Bill"); queueCustomers.offer("Kim"); queueCustomers.offer("Lee"); queueCustomers.offer("Peter"); queueCustomers.offer("Sam");

System.out.println("Queue before: " + queueCustomers);

System.out.println("First comes: " + queueCustomers.pollFirst()); System.out.println("Last comes: " + queueCustomers.pollLast()); System.out.println("Queue after: " + queueCustomers); Output:

Queue before: [Bill, Kim, Lee, Peter, Sam]

First comes: Bill

Last comes: Sam

Queue after: [Kim, Lee, Peter]

For a blocking queue, use the take()or poll(time, unit) methods in case you want the current thread to wait until an element becomes available. For example:

BlockingQueue<String> queueCustomers = new ArrayBlockingQueue<>(100); queueCustomers.offer("Kathe"); try {

String nextCustomer = queueCustomers.take();

} catch (InterruptedException ie) { ie.printStackTrace();

}

# 4. Examining the Head of the Queue

In contrast to the remove() method, the examine methods element() and peek() return (but do not remove) the head of the queue. So consider using these methods in case you just want to check what is currently in the head element without modifying the queue.

Also, you need to know the difference between the element() and peek() methods:

The element() method throws an exception in case the queue is empty, whereas the peek() method returns null. For example:

Queue<String> queueCustomers = new PriorityQueue<>(); queueCustomers.offer("Jack");

System.out.println("who's next: " + queueCustomers.poll()); // this returns null in case the queue is empty

System.out.println("who's next: " + queueCustomers.peek());

// this throws exception in case the queue is empty

System.out.println("who's next: " + queueCustomers.element());

For a deque, use the getFirst() or peekFirst() methods to examine the first element, and getLast() or peekLast() to examine the last element. Here’s an example: Deque<Integer> queueNumbers = new ArrayDeque<>(); queueNumbers.add(10); queueNumbers.add(20); queueNumbers.add(30); queueNumbers.add(40);

System.out.println("first: " + queueNumbers.getFirst());

System.out.println("last: " + queueNumbers.peekLast()); **There’s no method for examining a blocking queue.**

# 5. Iterating over Elements in the Queue

We can use the enhanced for loop, iterator and forEach() method to iterate over elements in the queue. The following code snippet illustrates how to iterate a linked list using the enhanced for loop: Queue<String> queueNames = new LinkedList<>(); queueNames.add("Dale"); queueNames.add("Bob"); queueNames.add("Frank"); queueNames.add("Alice"); queueNames.add("Eric"); queueNames.add("Cole"); queueNames.add("John"); for (String name : queueNames) {

System.out.println(name);

} Output:

Dale

Bob

Frank

Alice

Eric

Cole

John

More simply, using Lambda expression with forEach() method in Java 8:

queueNames.forEach(name -> System.out.println(name));

The following example iterates over elements of a PriorityQueue which sorts elements by natural ordering: Queue<String> queueNames = new PriorityQueue<>(); queueNames.add("Dale"); queueNames.add("Bob"); queueNames.add("Frank"); queueNames.add("Alice"); queueNames.add("Eric"); queueNames.add("Cole"); queueNames.add("John");

queueNames.forEach(name -> System.out.println(name)); Output:

Alice

Bob

Cole

Dale

Eric

Frank John

As you can see in the output, the elements are sorted in the alphabetic order (natural ordering of Strings).

**NOTE:** Pay attention when using an iterator of a PriorityQueue, because it is not guaranteed to traverse the elements of the priority queue in any particular order.

# 6. Concurrent Queues

All implementations of BlockingQueue are thread-safe. The following implementations are not:

* LinkedList
* ArrayDeque
* PriorityQueue

When you want to use a synchronized linked list, use the following code:

List list = Collections.synchronizedList(new LinkedList<>());

And consider using the PriorityBlockingQueue instead of the PriorityQueue when you want to use a synchronized priority queue.

# Producer - Consumer Example Using Queue

Typically, queue is used to implement producer-consumer scenarios. The following kinds of program will need to use queue:

* **Chat applications:** Messages are put into a queue. When you are sending a message, you are the producer; and your friend who reads the message, is the consumer. Messages need to be kept in queue because of network latency. Imagine network connection dropped when you are trying to send a message. In this case, the message is still in the queue, awaiting the receiver to consume upon the connection becomes available.
* **Online help desk applications:** Imagine a company has 5 persons working as customer support staffs. They chat with clients through a help desk application. They can talk with maximum 5 clients at a time, so other clients will be queued up. When a staff finishes serving a client, the next client in the queue is served next.
* **Real-time processing applications such as screen recorder**. The logic behind this kind of application is there are two threads working concurrently: The producer thread captures screenshots constantly and puts the images into a queue; the consumer thread takes the images from the queue to process the video.
* **And much more.**

Above I name only few types of application in which we need to use queue. Remember using queue when you need to implement producer-consumer processing.

In Java, using a BlockingQueue implementation is a good choice, as its put(e) method let the producer thread waits for space to become available in the queue, and its take() method let the consumer thread waits for an element become available in the queue.

The following is a pseudo-code of a typical producer-consumer application:

class Producer implements Runnable { private final BlockingQueue queue; Producer(BlockingQueue q) { queue = q; } public void run() {

while (true) { queue.put(produce()); }

}

Object produce() { ... }

}

class Consumer implements Runnable { private final BlockingQueue queue; Consumer(BlockingQueue q) { queue = q; } public void run() {

while (true) { consume(queue.take()); }

}

void consume(Object x) { ... }

}

class Program { void main() {

BlockingQueue q = new SomeBlockingQueueImplementation();

Producer p = new Producer(q);

Consumer c1 = new Consumer(q);

Consumer c2 = new Consumer(q);

new Thread(p).start(); new Thread(c1).start(); new Thread(c2).start();

}

}

Here, the Producer class is a thread which constantly produces objects and put them into the queue. In practice, we should specify condition to exit the loop, such as closing/shutdown the program or a maximum number of objects reached.

The Consumer class is another thread which constantly takes objects from the queue to process. In practice, we should specify condition to stop this thread by checking the queue for a special object (null, false or special value), for example:

while (true) {

Integer number = queue.take(); if (number == -1) { break; }

consume(number);

}

And in this case, the producer is responsible to put this special object into the queue to indicate there’s no more elements to process.

For you reference, here’s a working example that gives you the idea. It’s a program that creates one producer thread and one consumer thread:

Producer:

import java.util.\*; import java.util.concurrent.\*;

public class Producer implements Runnable {

private BlockingQueue<Integer> queue; public Producer (BlockingQueue<Integer> queue) { this.queue = queue;

}

public void run() { try {

for (int i = 0; i < 10; i++) { queue.put(produce());

Thread.sleep(500);

}

queue.put(-1); // indicates end of producing System.out.println("Producer STOPPED.");

} catch (InterruptedException ie) { ie.printStackTrace();

}

}

private Integer produce() {

Integer number = new Integer((int) (Math.random() \* 100));

System.out.println("Producing number => " + number); return number;

}

}

Consumer:

import java.util.\*; import java.util.concurrent.\*; public class Consumer implements Runnable { private BlockingQueue<Integer> queue; public Consumer(BlockingQueue<Integer> queue) { this.queue = queue;

}

public void run() { try {

while (true) {

Integer number = queue.take(); if (number == -1) { break;

}

consume(number);

Thread.sleep(1000);

}

System.out.println("Consumer STOPPED.");

} catch (InterruptedException ie) { ie.printStackTrace();

} }

private void consume(Integer number) {

System.out.println("Consuming number <= " + number);

}

}

Test Program:

import java.util.\*; import java.util.concurrent.\*; public class ProducerConsumerTest {

public static void main(String[] args) {

BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(20);

Thread producer = new Thread(new Producer(queue));

Thread consumer = new Thread(new Consumer(queue)); producer.start(); consumer.start();

}

}

Compile and run this program would print the following output:

Producing number => 21

Consuming number <= 21

Producing number => 90

Producing number => 51

Consuming number <= 90

Producing number => 23

Producing number => 61

Consuming number <= 51

Producing number => 63

Producing number => 75

Consuming number <= 23

Producing number => 99

Consuming number <= 61

Producing number => 59

Producing number => 31 Producer STOPPED.

Consuming number <= 63

Consuming number <= 75

Consuming number <= 99

Consuming number <= 59

Consuming number <= 31 Consumer STOPPED.