**Blocking Queue :**

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

Java 5 comes with blocking queue implementations in the java.util.concurrent package. Yet, it can be useful to know the theory behind their implementation.

Notice how notifyAll() is only called from enqueue() and dequeue() if the queue size is equal to the size bounds (0 or limit). If the queue size is not equal to either bound when enqueue() or dequeue() is called, there can be no threads waiting to either enqueue or dequeue items.

**Priority Queue :**

java.lang.Object

java.util.AbstractCollection<E>

java.util.AbstractQueue<E>

java.util.PriorityQueue<E>

* Type Parameters:

E - the type of elements held in this collection

All Implemented Interfaces:

[Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html), [Iterable](https://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html" \o "interface in java.lang)<E>, [Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<E>, [Queue](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html)<E>

public class PriorityQueue<E>

extends [AbstractQueue](https://docs.oracle.com/javase/7/docs/api/java/util/AbstractQueue.html)<E>

implements [Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

An unbounded priority [queue](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) based on a priority heap. The elements of the priority queue are ordered according to their [natural ordering](https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html), or by a [Comparator](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in ClassCastException).

The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.

A priority queue is unbounded, but has an internal *capacity* governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

This class and its iterator implement all of the *optional* methods of the [Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html) and [Iterator](https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html) interfaces. The Iterator provided in method [iterator()](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html#iterator()) is *not* guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using Arrays.sort(pq.toArray**()).**

**Array Blocking Queue :**

Class ArrayBlockingQueue<E>

java.lang.Object

java.util.AbstractCollection<E>

java.util.AbstractQueue<E>

java.util.concurrent.ArrayBlockingQueue<E>

**Type Parameters:**

E - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Iterable<E>, Collection<E>, BlockingQueue<E>, Queue<E

public class **ArrayBlockingQueue<E>**

extends [AbstractQueue](https://docs.oracle.com/javase/7/docs/api/java/util/AbstractQueue.html)<E>

implements [BlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html)<E>, [Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

A bounded [blocking queue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) backed by an array. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

This is a classic "bounded buffer", in which a fixed-sized array holds elements inserted by producers and extracted by consumers. Once created, the capacity cannot be changed. Attempts to put an element into a full queue will result in the operation blocking; attempts to take an element from an empty queue will similarly block.

This class supports an optional fairness policy for ordering waiting producer and consumer threads. By default, this ordering is not guaranteed. However, a queue constructed with fairness set to true grants threads access in FIFO order. Fairness generally decreases throughput but reduces variability and avoids starvation.

This class and its iterator implement all of the optional methods of the [Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html) and [Iterator](https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html) interfaces.

This class is a member of the [Java Collections Framework](https://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html).

**Priority Blocking queue :**

PriorityBlockingQueue is an unbounded blocking queue that uses the same ordering rules as class [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/) and supplies blocking retrieval operations. Since it is unbounded, adding elements may sometimes fail due to resource exhaustion resulting in [OutOfMemoryError](https://www.geeksforgeeks.org/understanding-outofmemoryerror-exception-java/). This class does not permit null elements.

PriorityBlockingQueue class and its iterator implements all of the optional methods of the Collection and Iterator interfaces. The Iterator provided in method iterator() is not guaranteed to traverse the elements of the PriorityBlockingQueue in any particular order. For ordered traversal, use [Arrays.sort(pq.toArray())](https://www.geeksforgeeks.org/arrays-sort-in-java-with-examples/). Also, method drainTo() can be used to remove some or all elements in priority order and place them in another collection.

Operations on this class make no guarantees about the ordering of elements with equal priority. If an ordering is needed to be enforced, define custom classes or comparators that use a secondary key to break ties in primary priority values.

**Hashtable class** :

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

Points to remember

* A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* Java Hashtable class contains unique elements.
* Java Hashtable class doesn't allow null key or value.
* Java Hashtable class is synchronized.
* The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Hashtable class declaration

Let's see the declaration for java.util.Hashtable class.

public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable

You must override hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable.

.

**Concurrent HashMap :**

**ConcurrentHashMap:**It allows concurrent access to the map. Part of the map called Segment (internal data structure) is only getting locked while adding or updating the map. So ConcurrentHashMap allows concurrent threads to read the value without locking at all. This data structure was introduced to improve performance**.**

**Concurrency-Level:**Defines the number which is an estimated number of concurrently updating threads. The implementation performs internal sizing to try to accommodate this     many threads.

**Load-Factor:**It's a threshold, used to control resizing.

**Initial Capacity:**The implementation performs internal sizing to accommodate these many elements.

A ConcurrentHashMap is divided into number of segments, and the example which I am explaining here used default as 32 on initialization.

A ConcurrentHashMap has internal final class called Segment so we can say that ConcurrentHashMap is internally divided in segments of size 32, so at max 32 threads can work at a time. It means each thread can work on a each segment during high concurrency and atmost 32 threads can operate at max which simply maintains 32 locks to guard each bucket of the ConcurrentHashMap.

**The definition of Segment is as below:**

**/\*\* Inner Segment class plays a significant role \*\*/**

protected static final class Segment {

protected int count;

protected synchronized int getCount() {

return this.count;

}

protected synchronized void synch() {}

}

/\*\* Segment Array declaration \*\*/

public final Segment[] segments = new Segment[32];

As we all know that Map is a kind of data structure which stores data in key-value pair which is array of inner class Entry, see as below:

static class Entry implements Map.Entry {

protected final Object key;

protected volatile Object value;

protected final int hash;

protected final Entry next;

Entry(int hash, Object key, Object value, Entry next) {

this.value = value;

this.hash = hash;

this.key = key;

this.next = next;

}

// Code goes here like getter/setter

}