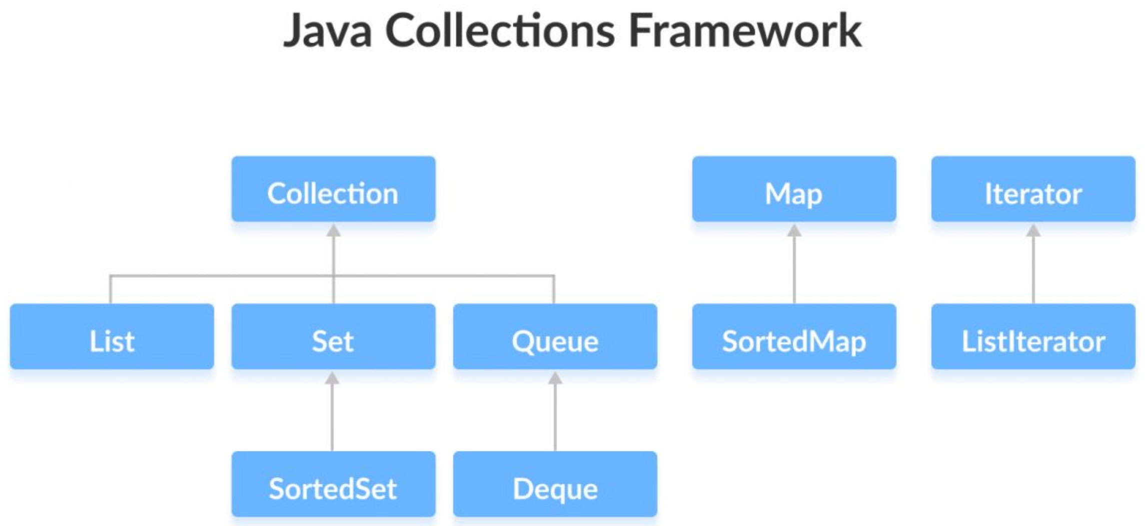
Java Collection Framework

Java Collection Framework Interfaces



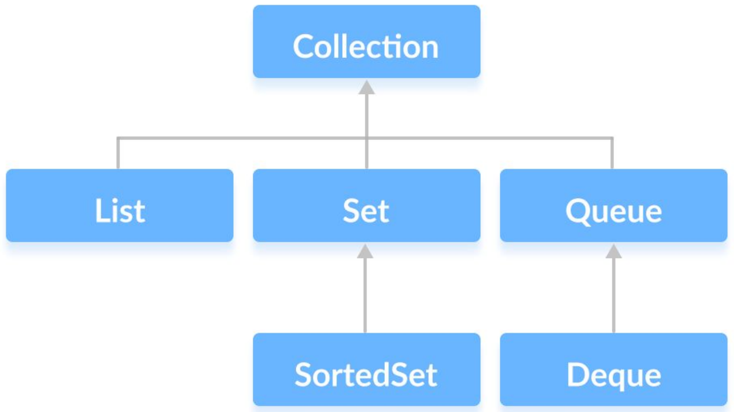
Java Collection Framework

1. Introduction to collection interfaces
2. List
3. Set
4. Queue
5. Map
6. Natural Ordering and Sorting
7. Comparators

Collection Interface

The Collection interface is the root interface of the collections framework hierarchy.

Java does not provide direct implementations of the Collection interface but provides implementations of its subinterfaces like **List, Set, and Queue.**



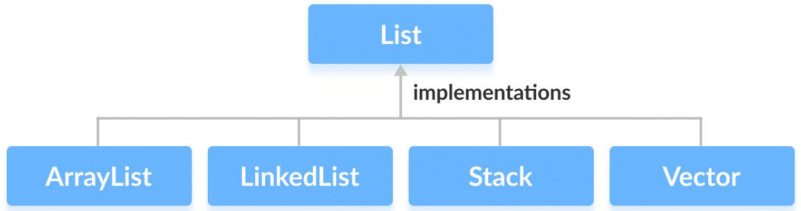
Methods of Collection

- add() - inserts the specified element to the collection
- size() - returns the size of the collection
- remove() - removes the specified element from the collection
- iterator() - returns an iterator to access elements of the collection
- addAll() - adds all the elements of a specified collection to the collection
- removeAll() - removes all the elements of the specified collection from the collection
- clear() - removes all the elements of the collection

List Interface

In Java, the List interface is an ordered collection that allows us to store and access elements sequentially. It extends the Collection interface.

Since List is an interface, we cannot create objects from it.In order to use functionalities of the List interface, we can use these classes:



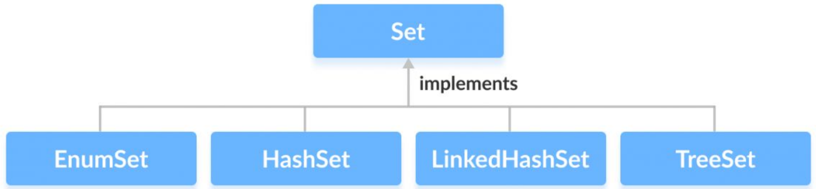
- ArrayList
- LinkedList
- Vector
- Stack

Set Interface

The Set interface of the Java Collections framework provides the features of the mathematical set in Java. It extends the Collection interface.Unlike the List interface, sets cannot contain duplicate elements.

In order to use functionalities of the Set interface, we can use these classes:

- HashSet
- LinkedHashSet
- EnumSet
- TreeSet

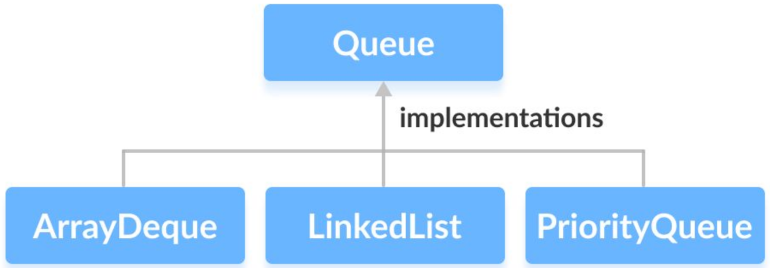


Queue Interface

The Queue interface of the Java collections framework provides the functionality of the queue data structure. It extends the Collection interface.

In order to use the functionalities of Queue, we need to use classes that implement it:

- ArrayDeque
- LinkedList
- PriorityQueue



Map Interface

The Map interface of the Java collections framework provides the functionality of the map data structure.

Since Map is an interface, we cannot create objects from it.

In order to use functionalities of the Map interface, we can use these classes:

- HashMap
- EnumMap
- LinkedHashMap
- WeakHashMap
- TreeMap

Java ArrayList & Generics

ArrayList vs Array

The ArrayList class is an implementation of the List interface that allows us to create resizable-arrays.

In Java, we need to declare the size of an array before we can use it. Once the size of an array is declared, it's hard to change it.

To handle this issue, we can use the ArrayList class. The ArrayList class present in the java.util package allows us to create resizable arrays.

Unlike arrays, array lists (objects of the ArrayList class) can automatically adjust its capacity when we add or remove elements from it. Hence, array lists are also known as dynamic arrays.

Creating an ArrayList

```
ArrayList<Type> arrayList= new ArrayList<>();
```

Here, Type indicates the type of an array list. For example,

```
// create Integer type arraylist
ArrayList<Integer> arrayList = new ArrayList<>();
// create String type arraylist
ArrayList<String> arrayList = new ArrayList<>();
```

Methods of ArrayList

- **add**(element)
- **addAll**(Collection)
- **get**(index)
- **set**(index, value)
- **remove**(index)
- **removeAll**()
- **clear**()
- **size**()
- **contains**()
- **isEmpty**()
- **toArray**()

Java LinkedList

LinkedList

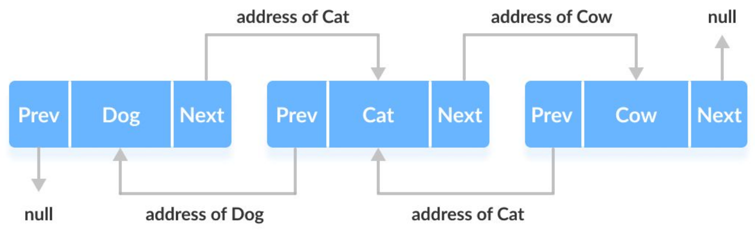
Linked List are linear data structures where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays.

It also has few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access.

LinkedList in Java

To store the elements in a linked list we use a doubly linked list which provides a linear data structure and also used to inherit an abstract class and implement list and deque interfaces. In Java, LinkedList class implements the list interface.

The LinkedList class also consists of various constructors and methods like other java collections.



LinkedList Implementation in Java

Java Vector & Stack

Vector

The Vector class is an implementation of the List interface that allows us to create resizable-arrays similar to the ArrayList class.

Java Vector vs. ArrayList

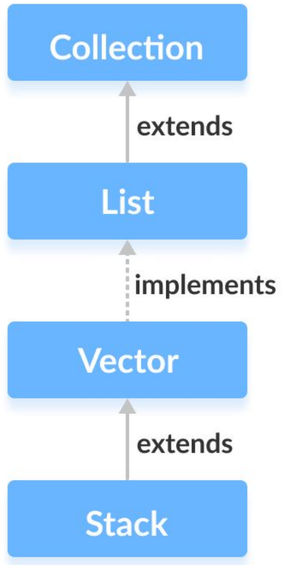
In Java, both ArrayList and Vector implements the List interface and provides the same functionalities. However, there exist some differences between them. The Vector class synchronizes each individual operation. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation. However, in array lists, methods are not synchronized.

Note: It is recommended to use ArrayList in place of Vector because vectors are not thread safe and are less efficient.

Stack

The Java collections framework has a class named Stack that provides the functionality of the stack data structure.

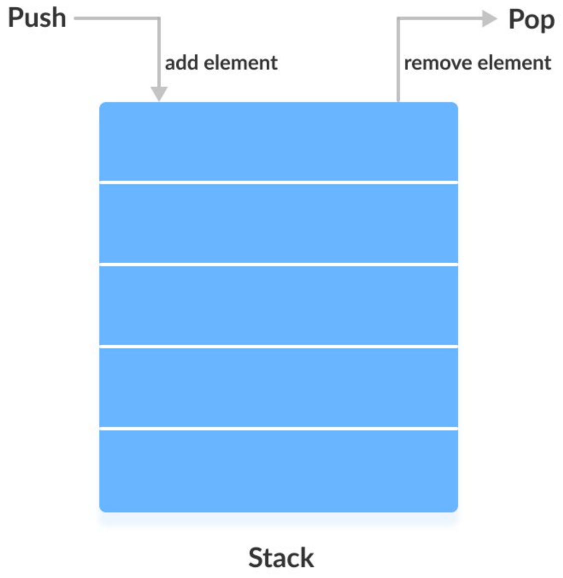
The Stack class extends the Vector class.



Stack Implementation

In stack, elements are stored and accessed in Last In First Out manner. That is, elements are added to the top of the stack and removed from the top of the stack.

Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.



Methods of Stack

- push(E e)
- pop()
- peek()
- search(E e)
- empty()

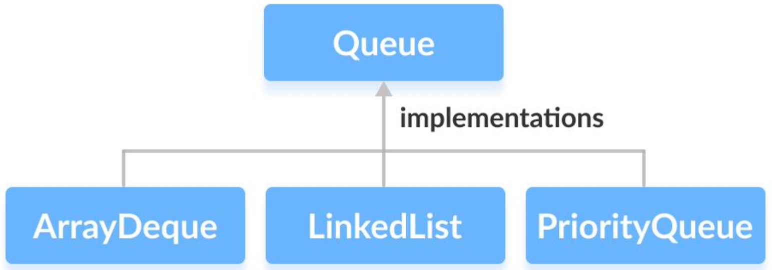
Java Queue

Queue Interface

The Queue interface of the Java collections framework provides the functionality of the queue data structure. It extends the Collection interface.

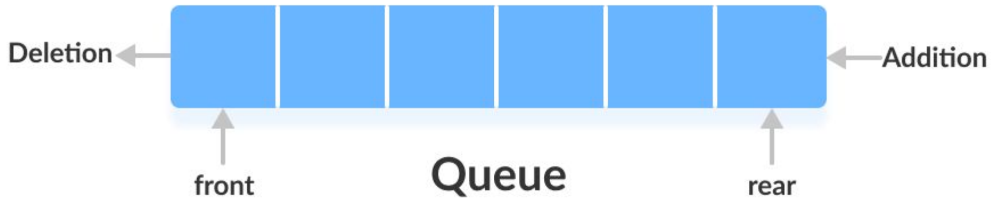
In order to use the functionalities of Queue, we need to use classes that implement it:

- ArrayDeque
- LinkedList
- PriorityQueue



Working of Queue Data Structure

In queues, elements are stored and accessed in First In, First Out manner. That is, elements are added from the behind and removed from the front.



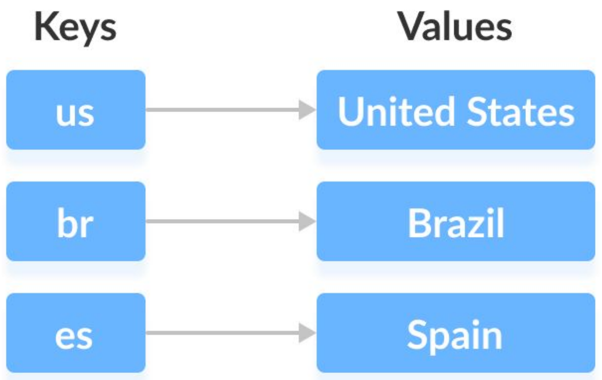
Methods of Queue

(throw Exception)	(return false / null)
add()	offer()
remove()	poll()
element()	peek()

Map Interface

In Java, elements of Map are stored in key/value pairs. Keys are unique values associated with individual Values. A map cannot contain duplicate keys. And, each key is associated with a single value.

We can access and modify values using the keys associated with them. In the above diagram, we have values: United States, Brazil, and Spain. And we have corresponding keys: us, br, and es. Now, we can access those values using their corresponding keys.

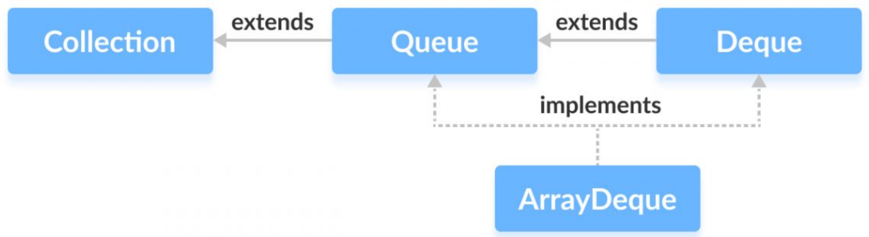


Java ArrayDeque

ArrayDeque

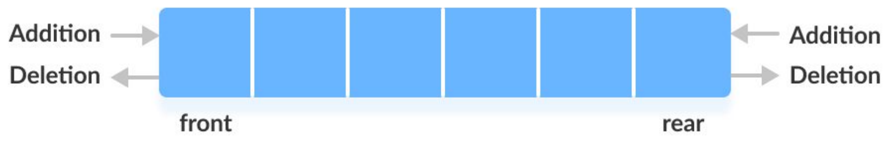
An **ArrayDeque** (also known as an "Array Double Ended Queue", pronounced as "ArrayDeck") is a special kind of a growable array that allows us to add or remove an element from both sides.

An ArrayDeque implementation can be used as a **Stack** (Last-In-First-Out) or a **Queue** (First-In-First-Out).



Deque

In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can insert and remove elements from both front and rear.



Operation	Method	Method throwing Exception
Insertion from Head	*offerFirst(e)*	*addFirst(e)*
Removal from Head	*pollFirst()*	*removeFirst()*
Retrieval from Head	*peekFirst()*	*getFirst()*
Insertion from Tail	*offerLast(e)*	*addLast(e)*
Removal from Tail	*pollLast()*	*removeLast()*
Retrieval from Tail	*peekLast()*	*getLast()*

ArrayDeque as a Stack

To implement a LIFO (Last-In-First-Out) stacks in Java, it is recommended to use a deque over the Stack class. The ArrayDeque class is likely to be faster than the Stack class. ArrayDeque provides the following methods that can be used for implementing a stack.

push() - adds an element to the top of the stack

peek() - returns an element from the top of the stack

pop() - returns and removes an element from the top of the stack

ArrayDeque as a Queue

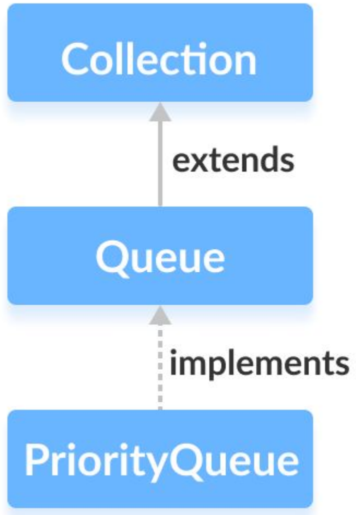
(throw Exception)	(return false / null)
add()	offer()
remove()	poll()
element()	peek()

Java PriorityQueue

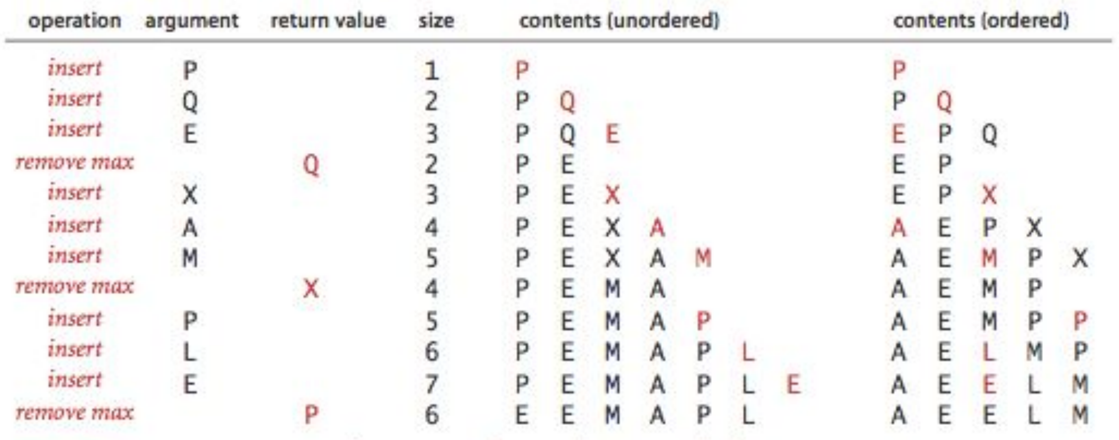
PriorityQueue

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.



PriorityQueue Operations



operation	argument	return value	size	contents (unordered)	contents (ordered)
insert	P		1	P	P
insert	Q		2	P Q	P Q
insert	E		3	P Q E	E P Q
remove max		Q	2	P E	E P
insert	X		3	P E X	E P X
insert	A		4	P E X A	A E P X
insert	M		5	P E X A M	A E M P X
remove max		X	4	P E M A	A E M P
insert	P		5	P E M A P	A E M P P
insert	L		6	P E M A P L	A E L M P
insert	E		7	P E M A P L E	A E E L M
remove max		P	6	E E M A P L	A E E L M

A sequence of operations on a priority queue

Methods of PriorityQueue

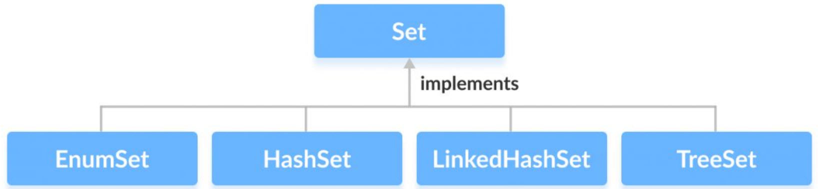
- add()
- remove()
- element()

Java Set

Set Interface

The Set interface of the Java Collections framework provides the features of the mathematical set in Java. It extends the Collection interface.Unlike the List interface, sets cannot contain duplicate elements.

In order to use functionalities of the Set interface, we can use these classes:



- HashSet
- LinkedHashSet
- EnumSet
- TreeSet

Methods of Set

- add(element)
- addAll(Collection)
- remove(element)
- removeAll()
- retainAll()
- clear()
- size()
- contains()
- containsAll()
- isEmpty()
- toArray()

Operations of Set

- **Union** - to get the union of two sets `x` and `y`, we can use **`x.addAll(y)`**
- **Intersection** - to get the intersection of two sets `x` and `y`, we can use **`x.retainAll(y)`**
- **Subset** - to check if `x` is a subset of `y`, we can use **`y.containsAll(x)`**

Implementation of Set

In order to use functionalities of the Set interface, we can use these classes:

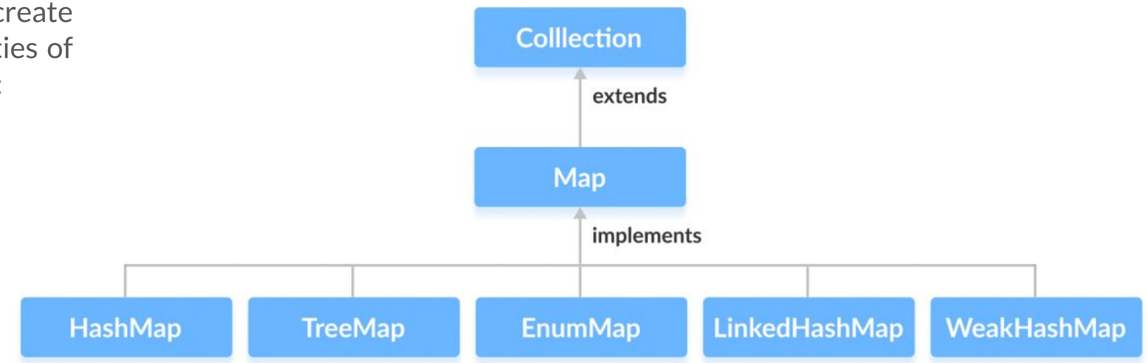
- HashSet
- TreeSet
- LinkedHashSet
- EnumSet

Java Map

Map Implementation

Since Map is an interface, we cannot create objects from it. In order to use functionalities of the Map interface, we can use these classes:

- HashMap
- EnumMap
- LinkedHashMap
- WeakHashMap
- TreeMap



Methods of Map

- **put**(K, V) - Inserts the association of a key K and a value V into the map. If the key is already present, the new value replaces the old value.
- **putAll**() - Inserts all the entries from the specified map to this map.
- **putIfAbsent**(K, V) - Inserts the association if the key K is not already associated with the value V.
- **get**(K) - Returns the value associated with the specified key K. If the key is not found, it returns null.
- **getOrDefault**(K, defaultValue) - Returns the value associated with the specified key K. If the key is not found, it returns the defaultValue.
- **containsKey**(K) - Checks if the specified key K is present in the map or not.
- **containsValue**(V) - Checks if the specified value V is present in the map or not.
- **replace**(K, V) - Replace the value of the key K with the new specified value V.

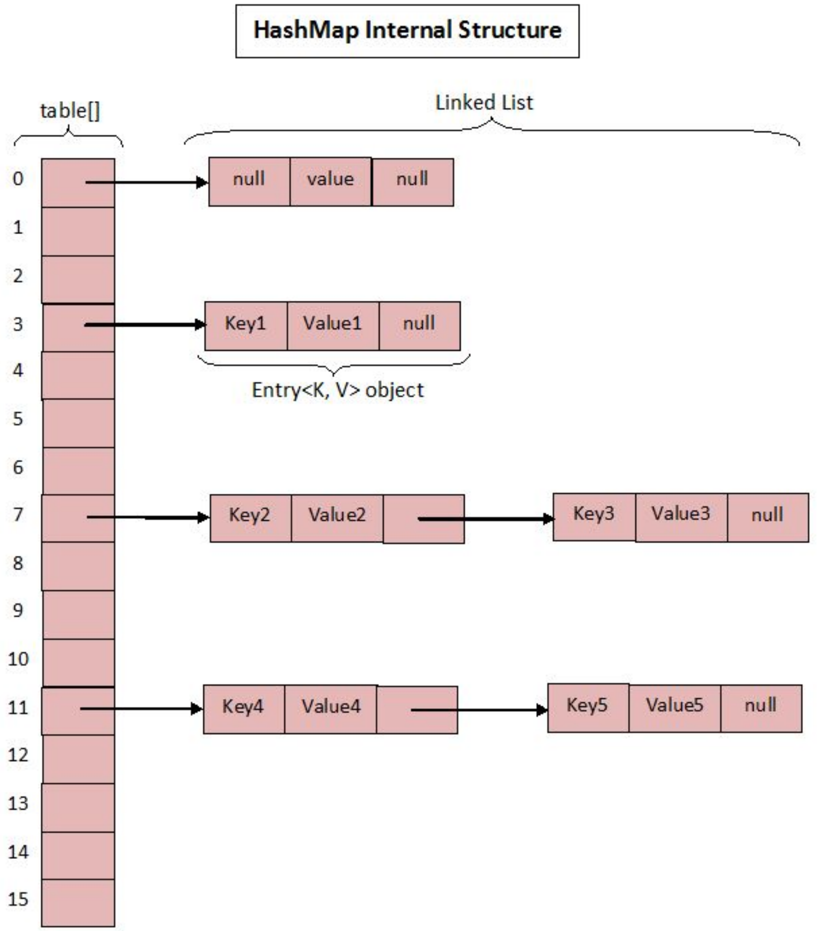
Methods of Map cntd.

- **replace**(K, oldValue, newValue) - Replaces the value of the key K with the new value new Value only if the key K is associated with the value oldValue.
- **remove**(K) - Removes the entry from the map represented by the key K.
- **remove**(K, V) - Removes the entry from the map that has key K associated with value V.
- **keySet**() - Returns a set of all the keys present in a map.
- **values**() - Returns a set of all the values present in a map.
- **entrySet**() - Returns a set of all the key/value mapping present in a map.

Internal Working of HashMap

HashMap uses an array table to store its key value pairs. Each element of the array holds the head of a linkedlist to avoid collision. The hash of every key is calculated and the elements are placed in the array using this hash function.

The default capacity is kept at 16 and the load factor at 0.75



Hashcode and Equals

hashCode() and equals() methods

hashCode() and equals() methods have been defined in **Object** class which is parent class for java objects. For this reason, all java objects inherit a default implementation of these methods.

Java hashCode()

Object class defined hashCode() method like this:

```
public int hashCode() {
        // TODO return the hashCode ;
}
```

Java equals()

Object class defined equals() method like this:

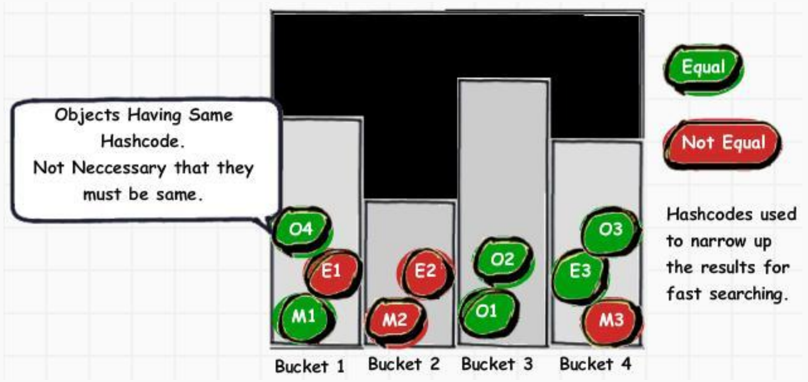
```
public boolean equals(Object obj) {
        return (this == obj);
}
```

The Contract

The contract between equals() and hashCode() is:

1) If two objects are equal, then they must have the same hash code.

2) If two objects have the same hash code, they may or may not be equal.


Objects Having Same Hashcode.
Not Neccessary that they must be same.
Equal
Not Equal
Hashcodes used to narrow up the results for fast searching.
O4
E1
M1
E2
M2
O2
O1
O3
E3
M3
Bucket 1
Bucket 2
Bucket 3
Bucket 4

Best Practices

1. Always use same attributes of an object to generate `hashCode()` and `equals()` both.
2. `equals()` must be *consistent* (if the objects are not modified, then it must keep returning the same value).
3. Whenever a.equals(b), then *a.hashCode()* must be same as *b.hashCode()*.
4. If you override one, then you should override the other.

Seven Best Books For Programming

1. Introduction to Algorithms: CLRS (Advanced)
2. Cracking the Coding Interview by Gayle Laakmann Mcdowell (Advanced)
3. Algorithms by Sedgwick (Moderate)
4. Data Structures and Algorithms made easy by Narasimha Karumanchi (Beginners)
5. Java for Dummies by Barry A. Burd (Beginners)
6. Operating system concepts by Abraham Silberschatz (General)
7. Computer Networking - A top down approach (General)

For Android / IOS / Web - refer to the official documentation

Comparable, Comparators and Sorting

Comparable Interface

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's **compareTo** method is referred to as its natural comparison method.

Lists (and arrays) of objects that implement this interface can be sorted automatically by **Collections.sort** (and **Arrays.sort**).

Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

```
public interface Comparable<T> {
   public int compareTo(T o);
}
```

Comparator Interface

A comparison function, which imposes a total ordering on some collection of objects.

Comparators can be passed to a sort method (such as **Collections.sort** or **Arrays.sort**) to allow precise control over the sort order.

Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that don't have a natural ordering.

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

Various Sorting Techniques

1. Comparable Objects
2. Comparator Objects
3. Anonymous Comparator Classes
4. Comparator.comparing() -> thenComparing() -> reverseOrder()