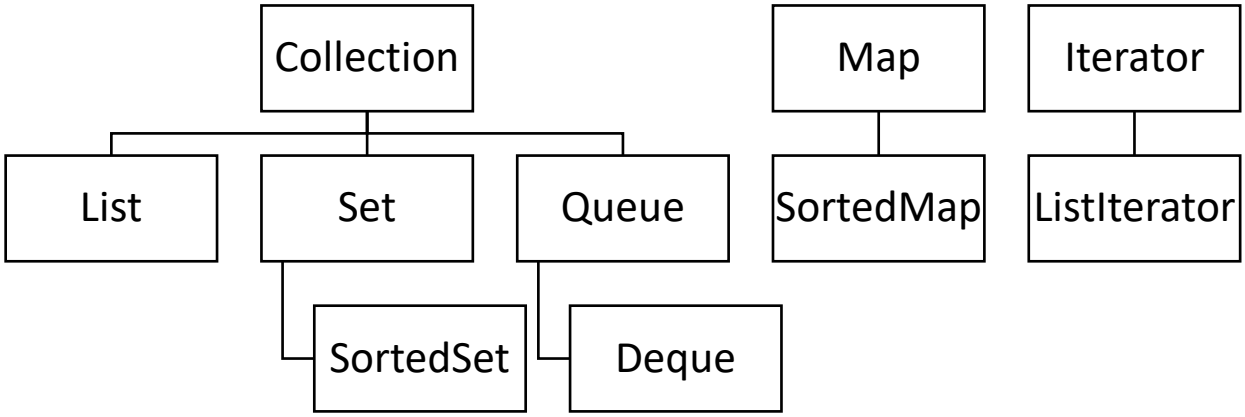
DATA STRUCTURES

. how to store data in a memory so that the computer may work effectively
. operations can be addition, deletion, updation, sorting, searching, etc.

TYPES OF DATA STRUCTURES

1**.** ARRAYS
2**.** STACKS
3**.** LINKED LISTS
4**.** QUEUES
5**.** GRAPHS
6**.** TREES
7**.** MAPS

*****JAVA has provided an API or collection framework to use the data structures. These collection frameworks contains the implementations of all the data structures.***



JAVA COLLECTION FRAMEWORK
COLLECTION INTERFACE

. The Collection Interface is the root interface of the Collection Framework hierarchy.
. JAVA does not provide direct implementation of Collection Interface but provides implementation of its subinterfaces like List, Set and Queue.

COLLECTION INTERFACE

i. LIST
ii. SET
  a. SortedSet
iii. QUEUE
  a. Deque

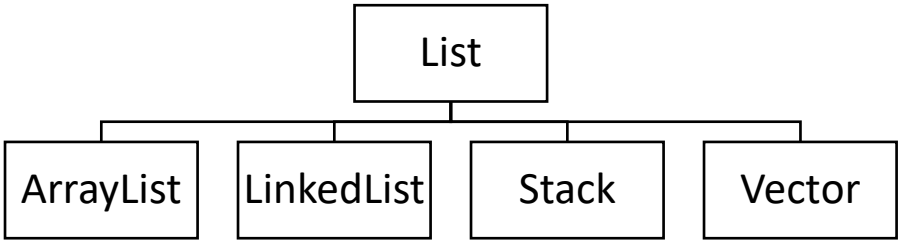
METHODS OF COLLECTION

. add() - inserts the specified element to the collection
. size() - returns the size of 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

. List Interface is an ordered collection that allows us to store and access the elements sequentially
. It extends the Collection Interface
. List Interface can contain duplicate elements
. Since List is an interface, so objects cannot be created from it**.** To use the functionalities of the List Interface, the following classes can be used:

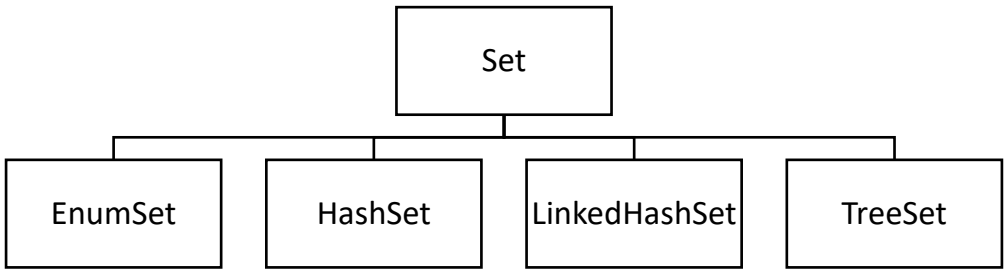
i. ArrayList
ii. LinkedList
iii. Vector
iv. Stack



SET INTERFACE

. Set Interface provides the features of the mathematical set in JAVA.
. It extends the Collection Interface
. Sets cannot contain duplicate elements
. Since Set is an interface, so objects cannot be created from it**.** To use the functionalities of the Set Interface, the following classes can be used:

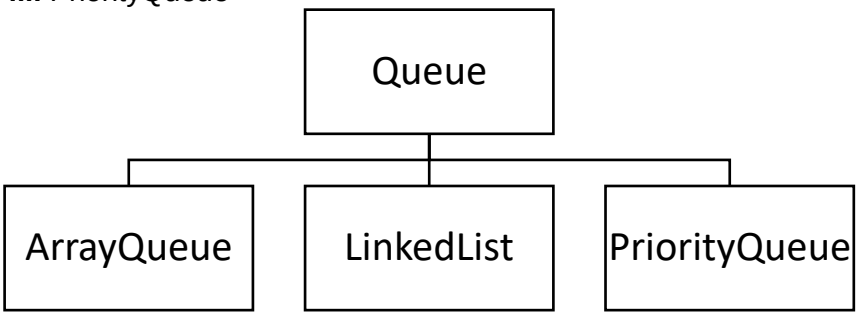
i. HashSet
ii. LinkedHashSet
iii. EnumSet
iv. TreeSet



QUEUE INTERFACE

. Queue is a Data Structure in which data is stored by following the order of FIFO (First In First Out)
. Queue Interface provides the functionality of queue data structure
. It extends the Collection Interface
. Since Queue is an interface, so objects cannot be created from it**.** To use the functionalities of the Queue Interface, the following classes can be used:

i. Array Deque
ii. LinkedList
iii. PriorityQueue

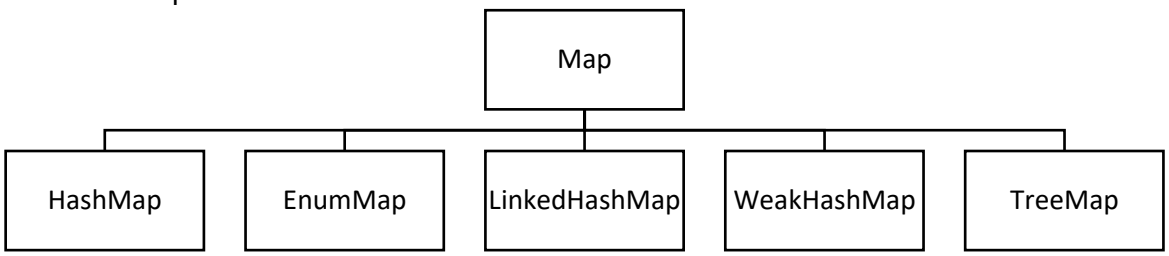


*****LinkedList class is present in two interfaces: List Interface and Queue Interface because LinkedList is a class and a class can implement multiple interfaces and LinkedList is implementing two interfaces because it needs the functionalities of both List Interface and Queue Interface***

MAP INTERFACE

. Map Interface does not implement the Collection Interface but comes under the Collection Framework
. Map Interface provides the functionality of the map data structure.
. Since Map is an interface, so objects cannot be created from it**.** To use the functionalities of the Map Interface, the following classes can be used:

i. HashMap <- IMP
ii. EnumMap
iii. LinkedHashMap
iv. WeakHashMap
v. TreeMap <- IMP



WHY MAP INTERFACE DOES NOT IMPLEMENT COLLECTION INTERFACE?

because Map contains key-value pair mapping which means for what value of a key is there a mapping or the mapping between a key and its value**.** Maps contains values in pairs and the working of pairing and hashing is different in this.

ARRAYLISTS

. Most-used Data Structure
. ArrayList class is an implementation of the List Interface that allows us to create reziable arrays.
. Present in 'java.util' package
. Array List (objects of ArrayList class) are also known as Dynamic Arrays.

ARRAYLIST vs ARRAY

. In Array, we have to declare the size of an array before we can use it and once its size is declared, it is hard to change it
. In Array List, which are the objects of the ArrayList class, its capacity is automatically adjusted when an element is added or removed from it.

SYNTAX:

ArrayList<Data_Type> arrayListName = new ArrayList<>(); //Creating an ArrayList

<Data_Type> --> Generics

*****In ArrayList, we cannot use int as Data_Type because it doesnot extends the object class and in Generics we have to define only the children of object class. So a wrapper class is made in JAVA which contains all the datatypes(int, bool, double, float, long, etc). The wrapper class of 'int' is 'Integer' and 'double' is 'Double'. This wrapper class can be used in Generics.***
*****We should use List<Data_Type> because List is at the top level and if there is a need to convert any two classes of List then it will be easily possible as they both would be List at the top-level.***

SYNTAX:

List<Data_Type> ListName = new classType<>(); //creating any list class object with list at top-level

WHY WAS GENERICS INTRODUCED IN JAVA?

Before Generics, the user was not able to define which data type a particular ArrayList will hold**.** So if the user enters elements of different data types then a problem can arise**.** The problem is when the code will be executed then the user would be expecting a particular output but since the ArrayList contains multiple data types the code will terminate and give an exception**.** This was a problem in big applications where the application would be expecting an integer but if the ArrayList contained a string then the code will terminate immediately**.** So to remove this, Generics was introduced in Java from version 1.5

BENEFITS OF GENERICS

Generics already defines the data type of the ArrayList and if the user enters a non-matching data type then the code will return a compilation error

METHODS OF ARRAYLIST

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

BENEFITS OF ARRAYLISTS

Many times we don't know what will be the size of a particular list**.** At that time ArrayList is used as we can increase or decrease the size of ArrayList as per need.

LINKEDLIST

. Implements list interface.
. Linear Data Structures
. Elements not stored in contiguous locations
. Every element is a seperate object with a data part and an address part.
. Linked using pointers and addresses

LINKEDLIST vs ARRAYLIST

In ArrayList, the data is stored in a continous manner, i.e., all the elements are stored at the same place and in a sequential manner in the memory**.**
In LinkedList, the data is stored in a non-continous manner, i.e., all elements are stored at different places in the memory and each element has a different address**.**

FINDING ELEMENTS IN LINKEDLIST

Since all the elements in ArrayList are stored in a continous manner, we can find any element using its index as the compiler knows the address of the starting element and can find the address of any element given its index.
But since all the elements in LinkedList are stored in a non-continous manner, it is difficult to find any element as each element has a different address in the memory**.** So to find any element easily we store the address of the next LinkedList along with the data**.** Then to find an element we will go to the first element and find the address of the next memory location where data is stored**.** Similarly we can traverse all the locations untill we find the element or we get the address as NULL.
*****Since JAVA does not has the concept of address, so we don't store the address of the next memory location in the element. Instead we store the reference of the next memory location in the element.***

IMPORTANT POINTS ABOUT LINKEDLISTS

. The place where the data and the reference are stored is called Node**.**
. Each Node contains the data and the reference to the next Node**.** A Node is stored in a class.
. The first element of a LinkedList is called Head.
. To traverse a LinkedList, we just need the Head.
*****TIME COMPLEXITY OF INSERTION/DELETION IN ARRAYLIST IS O(n).***

ADVANTAGES OF LINKEDLISTS

. Insertion of new element between any two elements is very easy as compared to ArrayList
. Deletion of any element is very easy.
. Dynamicity and Ease of Insertion/Deletion

DISADVANTAGES OF LINKEDLISTS

. Traversal of elements or Finding elements is difficult as we need to traverse all the nodes until we find the element.
. Nodes cannot be accessed directly

LINKEDLIST IN JAVA

. To store elements in LinkedList, a Doubly LinkedList is used
. Doubly LinkedList provides linear data structure and inherits an abstract class and implement List and Deque Interfaces.
. LinkedList class consists of various constructors and methods like other collections

SYNTAX:

```
List<Data_Type> linkedListName = new LinkedList<>();      //Creating a LinkedList
```

*****Has same methods as List Interfaces and ArrayList. If we use ArrayList instead of LinkedList, then also all the methods will run as they all are under the List Interface***

SYNTAX FOR TRAVERSAL IN LINKEDLIST:

```
Node temp = head;
while(temp.next != null)
{
        temp = temp.next
}
```

When the list is empty, and if we try to traverse the list then the program will throw an null pointer exception**.** This will happen because initially our head is null and so our temp is null and when we use temp.next in while loop it means we are finding the next of null (null.next)**.** So it throws an null pointer exception**.** To avoid this we first need to check if our list is empty or not**.** We can do this by checking if the head is null or not**.** If the head is null, it means our list is empty and we cannot traverse the list without adding any element to the list.

```
if(head == null)
{
        //list empty
        //cannot traverse the list without entering the elements
        return;
}
                OR
if(isEmpty())
{
        //list empty
        //cannot traverse the list without entering the elements
        return;
}
boolean isEmpty()
{
        if(head == null)
        {
                return true;
        }
        else
        {
                return false;
        }
}
```

This can be shortened further by:

```
boolean isEmpty()
{
        return head == null;
}
```

VECTOR

. Vector class is an implementation of List Interface

. It allows us to create resizable-arrays similar to ArrayList class

VECTOR vs ARRAYLIST

Vector class synchronizes each individual operation i.e., it automatically applies lock to the operation which is performed by the user but in ArrayList the methods are not synchronized.

To make a time and space efficient program, we use ArrayList over Vectors.

*****ArrayList should be used in place of Vector because vectors are thread safe and are less efficient.***

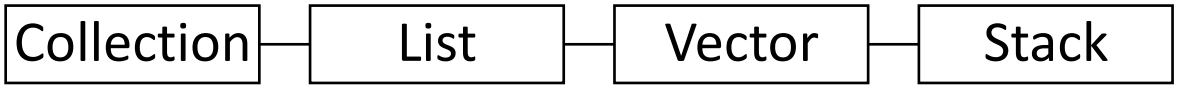
WHY ARE METHODS SYNCHRONIZED IN VECTORS

to make the operations thread safe, i.e., all the functions in vectors are thread safe.

This means if there is a block of code and we have a single core CPU then it will perform any operation step-by-step**.** This would take both time and more no of steps**.** On the other hand, if we have multi core CPU then they will divide the operation among themselves and do the work faster and in lesser no of steps**.** Every processor contains some threads and in these threads, process takes place parallely**.** Let there be two threads and each thread is doing the same function on a list, then it is possible that there can be a clash in both of the function**.** In this case, synchronized keyword comes to light**.** The synchronized keyword adds a lock to the function and only the thread that has the synchronized keyword will be able to perform the function**.** After the completion of the function, the lock is released and it will shift to the thread which would be doing the similar function.

In ArrayList, the size of list is increased by 1.5 times, whenever space is required
In Vectors, the size of list is increased by 2 times, whenever space is required

STACK
. Stack is a Data Structure in which we push or pop elements as per our need
. It stores and access data in LIFO(Last In First Out)/FILO(First In Last Out) manner
. Stack provides functionality of Stack Data Structure
. Stack class extends Vector class
. Can be implemented using Arrays and LinkedList



METHODS OF STACK
. push(E e) . pop() . peek()
. search(E e) . empty()

Stack is used in those questions where we first need to go a final point and then return to the start point using the same path. Stack is also used in Reccursion

QUEUE
. Stack is a Data Structure in which we push or pop elements as per our need
. It stores and access data in FIFO(First In First Out)/LILO(Last In Last Out) manner
. Stack provides functionality of Queue Data Structure
. Can be implemented using Arrays and LinkedList

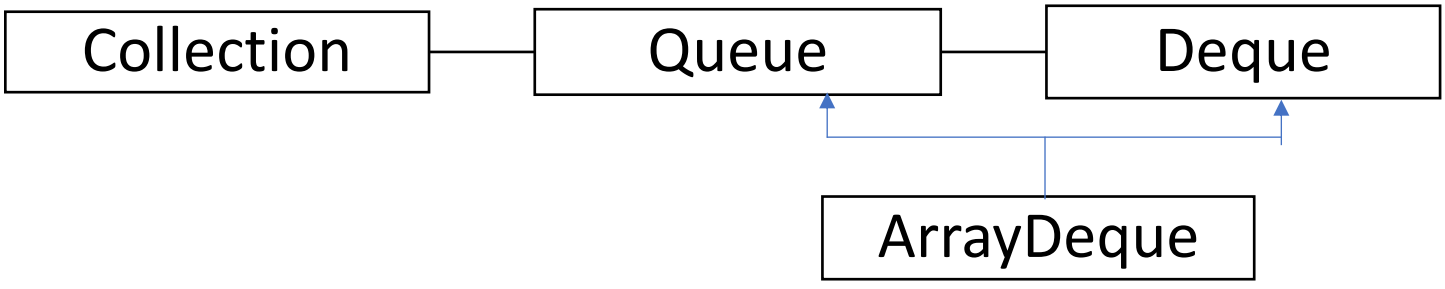
METHODS OF QUEUE

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

. In Queue, we use two pointers: Head and Rear
. Initially both Head and Rear are Null
. pop() is performed on Head and push() is performed on Rear
. pop() is called dequeue() and push() is called enqueue()
. Whenever the Head and Rear are at the same position, it means that the Queue is empty

ADDING AND REMOVING ELEMENTS FROM QUEUE
Initially both Head and Rear are at the start position and the queue is empty. When we add an element the element goes to the Rear position of the Queue and the Rear pointer shifts by 1 position. When an element is to be removed, it is removed from the Head position and then the Head shifts by 1 position. If at any place the Head and Rear pointer come at same position, this means that the Queue is Empty. If the Head is still at the Start Position and the Rear pointer is at the End, this means that the Queue is Full. This is in the case of Linear Queue.

ARRAY DEQUE
. Array Deque means 'Array Doubly Ended Queue'
. Doubly Ended Queue is a type of Data Structure similar to Queue
. Special kind of growable array
. In this, we can insert and remove the elements from both Head and Tail
. We can implement both Queue and Stack using this



DEQUE

. In regular Queue, elements are added from Tail and removed from Head.
. In Deque, elements can be added or removed from both Head and Tail

METHODS IN DEQUE

OPERATION	METHOD	MEHTOD 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

. ArrayDeque is recommended to implement LIFO (Last-In-First-Out) Stack Class
. ArrayDeque is faster than Stack Class
. ArrayDeque provides following methods to implement a stack
  i. **push()** - adds an element to the top of stack
  ii. **peek()** - returns the element from the top of the stack
  iii. **pop()** - returns and removes an element from the top of stack

WHY IS ARRAYDEQUE FASTER THAN STACK?

because Stack Class is implemented from Vector Class and all the functions in a Vector Class are thread safe and due to this overhead occurs on some functions to acquire locks and ArrayDeque is not thread safe and doesnot contains overheads**.** So the implementation of stack using ArrayDeque is faster than using Stack Class.

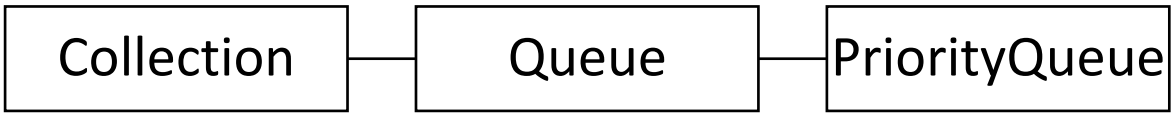
ARRAYDEQUE AS A QUEUE

. ArrayDeque provides following methods to implement a queue

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

PRIORITY QUEUE

. It 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 same priority occur then they are served according to their order in the queue
. In OS, there are some processes that have some higher priority and so those processes are removed from the task queue first
. In queue, FIFO rule is implemented whereas in priority queue elements are removed on the basis of priority
. Priority of an element in a queue can be defined in any way
. In priority queue, all the elements must be comparable so that we can find which element has higher priority



METHODS OF PRIORITY QUEUE

. add() . remove() . element()

*****Whenever an element is inserted into the priority queue, it is inserted in an unsorted manner but when the elements are removed from the same queue, then they are remvoed in the sorted manner***

WHERE TO USE PRIORITY QUEUE

. when we are given a lot of numbers and we need to find the 5 biggest numbers among them, then we can first sort all the elements and then find the numbers but it will have the time complexity of O(nlogn)**.** If we do the same using Priority Queue then the time taken will be O(nlogk), where k is the total no of elements we want to find
. when we are given some LinkedList which has all elements sorted in it and we want to make a combined LinkedList with all the elements sorted.

PRIORITY QUEUE OPERATION

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 A M	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 P
insert	E		7	P E M A P L E	A E E L M P P
remove max		P	6	E E M A P L	A E E L M P

SLIDING WINDOW MAXIMUM

Given an array nums there is a sliding window of size k which is moving from the very left of the array to the very right**.** We can only see the k numbers in the window**.** Each time the sliding window moves to right by one position, return the max sliding window

SET

. Most powerful Data Structure
. In this all operation take a constant time whereas all other data structures take linear time
. doesnot allows duplicate elements

METHODS OF SET

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

HOW ARE OPERATIONS IN SETS PERFORMED IN CONSTANT TIME?

Whenever we store elements in an Array, the elements are stored in contiguous memory locations at definite index**.** In HashSet, the elements are not stored in a list/contiguous memory**.** We can assume that the elements are stored in a bag and when we put any element again in the bag then it checks and doesnot puts it in the bag**.** The checking of elements is done by Hash Functions**.** We give some input to the Hash Function and it returns some output corresponding to the particular input**.** The Hash Function gives the same output for the same input no matter how many times a particular input is given**.** The Hash Output corresponds to the location of the input in the memory**.** In other words we can say that the Hash Output helps to find the memory location of the particular input**.** All the operation done by the Hash Function is done in a constant time**.** So we can say that ADD function in Sets takes constant time as we have the direct memory location**.**
Similarly, REMOVE operation also takes a constant time as we have the direct memory location of the element which is to be deleted.
As we didn't had the memory location of elements in Array or LinkedList so the time taken to remove an element was more in Array and LinkedList.

LINKEDHASHSET

. In HashSet, it is not necessary that it will give the elements in the same order as it is given in input
. To get the elements in the same order as given in input, we use LinkedHashSet
. It has the same properties as Set
. It uses a LinkedList to store its elements
. It cannot do the operations in constant time as it uses the operations of LinkedList

TREESET

. It is used when we want the elements in a sorted order

OPERATIONS OF SET

OPERATION	METHODS TO USE	DESCRIPTION
Union	x.addAll(y)	to get the union of two sets 'x' and 'y'
Intersection	x.retainAll(y)	to get the intersection of two sets 'x' and 'y'
SubSet	y.containsAll(x)	to check if 'x' is a subset of 'y'

MAP INTERFACE

. Elements of Map are stored in key/value pairs
. Keys are unique values associated with individual Values
. Map cannot contain duplicate key and each key is associated with a single value
. Values can be accessed and modified only by using the keys associated with them

METHODS OF MAP

METHODS	DESCRIPTION
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 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 a 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 key K with the new specified value V
replace(k, oldValue, newValue)	Replaces the value of the key K with the newValue 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 the key K associated with value V
keySet()	returns the set of all keys present in a map
values()	returns the set of all values present in a map
entrySet()	returns a set of all key/value mapping present in a map

*****All operations in HashMap are completed in constant time***

WORKING OF HASHMAP

Let us assume we have some keys in form of String and each key has some value corresponding to it. We need to find the Hash Function for the keys. So we can find the ASCII values of each key and let this ASCII value be the value that will be returned by the Hash Function. Now we will find the remainder the result of Hash Function will give when divided by the length of Array because the result of Hash Function will be a bigger number and the size of our Array is limited. There is a possibility that two or more keys can have the same result of Hash Function. Due to this once the index is taken by a particular key in the Array, the other key will not be able to take it and it will not be present in the Array. This will result in a collision. This has occured because our Hash Function is not optimized. To takle collision, chaining is used.

CHAINING

In this, every index of the Array contains a LinkedList. So every element of the Array is stored in a LinkedList of the index of the Array. So if collision takes place at any point, then the element will be stored in the same index as the elements are stored in LinkedList and any new key introduced to the same index will be added to the end of LinkedList. There is a disadvantage of Chaining that it might be possible that multiple elements are being added to the same index but there are some spaces still left where those elements could be added. So to tackle this, we need to make more optimized Hash Functions. It could also be avoided by using Arrays of greater size.

HASH FUNCTION FOR STRINGS - *s.charAt(i)*31^(n-i) then find its remainder by dividing by size of Array*

INTERNAL WORKING OF HASHMAP

. HashMap uses an Array table to store its key-value pair
. Each element of Array holds the head of a LinkedList to avoid collision
. The hash of every key is calculated and elements are placed in Array using this Hash Function
. The default size of Array is kept 16 and the load factor at 0.75

*****Load Factor is a certain percentage which determines when the elements being added to the Array are to be stopped and when the size of the Array is to be increased. Once the Load Factor is reached, the size of Array is Doubled and Rehashing is performed.***

HASHCODE AND EQUALS

In JAVA there is an Object Class which is at the Top or Parent of all the classes and all the objects made of any Class, inherits the Object Class**.** There are some methods inside the Object Class which are inherited by all the objects and classes**.** These methods are:

. getClass()	**.** hashCode()	**.** wait()
. clone()	**.** equals()	**.** finalize()
. notifyall()	**.** toString()	**.** notify()

HashCode and Equals are one of these functions.

SYNTAX FOR HASHCODE

Object class defined hashCode() method like this:

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

SYNTAX FOR EQUALS

Object class defined equals() like this:

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

*****If we override the equals() method of an object then we have to override the hashCode() for the same object and the variables used to define the equals() method must be used to define the hashCode() method***

CONTRACT BETWEEN HASHCODE AND EQUALS

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

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

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

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

COMPARABLE INTERFACE

. It is an interface which defines the properties of the class, i.e., how this class will be sorted now, what will be its natural ordering now
. It is concrete or fixed
. Imposes a total ordering on the objects of each class that implements it
. This ordering is referred to as 'class's natural ordering'
. compareTo() method is referred to as its natural comparison method
. Lists of objects that implement this can be sorted automatically by Collections.sort and Arrays.sort
. Objects that implement this can be used as keys in a sorted map or as element in sorted set without the need to specify a comparator

SYNTAX

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

SYNTAX OF PRINTING A LIST OF STUDENTS USING LAMBDA:

```
listName.forEach(System.out::println);
```

*****While sorting in decreasing order, if the integer returning from the compareTo() method is positive then it means that the current object is smaller, and if it returns negative then it means that the current object is greater and if it returns 0 then it means that both the object are equal and reverse for increasing order.***

COMPARATOR INTERFACE

. When we don't want any natural ordering for a class
. It is flexible
. Comparision function which imposes total ordering on some collection of objects
. Can be passed to sort method (like Collections.sort or Arrays.sort) to allow precise control over the sort order
. Used to control the order of certain data structures (sorted sets or sorted maps)
. Provide an ordering for collections of objects that don't have a natural ordering

SYNTAX

```
public interface Comparator<T>
{
        int compare(T obj1, T obj2);
}
```

ADVANTAGES:

. Creation of many Comparators is possible
. Sorting can be done in many ways by just passing the comparator of the sorting we want to perform whereas in Comparable, we have to change the natural ordering of the class.

SYNTAX TO SORT LIST USING COMPARATOR AND LAMBDA:

```
Collections.sort(listName,Comparator.comparing(class::functionName).
                        thenComparing(class::functionName2).reversed());
```

****.thenComparing is used for further comparision if both values are equal in first comparision***
.reversed() is used to reverse the order