Collection Framework

A collection represents a group of object
Java collections provide classes & interfaces
for us to be able to write code interfaces
for us to be able to write code quickly &
effecientlly

Why do we need collections....?
We need collections for effecient storage &
better manipulation of data in Java.

For e.g. We use arrays to store integers
but what if we want to
→ Resize the array

→ Insert an element in between

→ Delete an element in array

→ Apply certain operations to change the array

- ArrayList → For variable size collections
- set → For distinct collection
- Stack → A LIFO data structure
  LIFO → Last in first out
- HashMap → For strong-key value pairs

ArrayList in Java

- The arrayList is a dynamic array found in java.util package
- The size of a normal array can't be changed, but ArrayList provides us the ability to inc. or dec. the size
- ArrayList is slower than the standard array, but it helps to manipulate data easily.

Syntax:

```
ArrayList<Integer> ll = new ArrayList<>();
// ArrayList object of Integer type.
```

Operations: (List.method(parameters))

→ add(Object): inserts an element at the end of ArrayList

→ add(Index, Object): inserts element at given index.

→ ~~remove()~~

→ remove(Index): is used to remove an element at a given index from the ArrayList.

Syntax

LL.remove(index)

→ contains(number): Is used to check if an arrayList contains a specified element or not. returns boolean.

→ addAll(List): The elements of ArrayList can be merged into another ArrayList.

- addAll(index, List): To add from a specific index.

→ ~~index~~
→ ~~IndexO~~
→ indexOf(number): prints the first occurence of a particular number. Returns -1 if the element is not present in the arrayList.

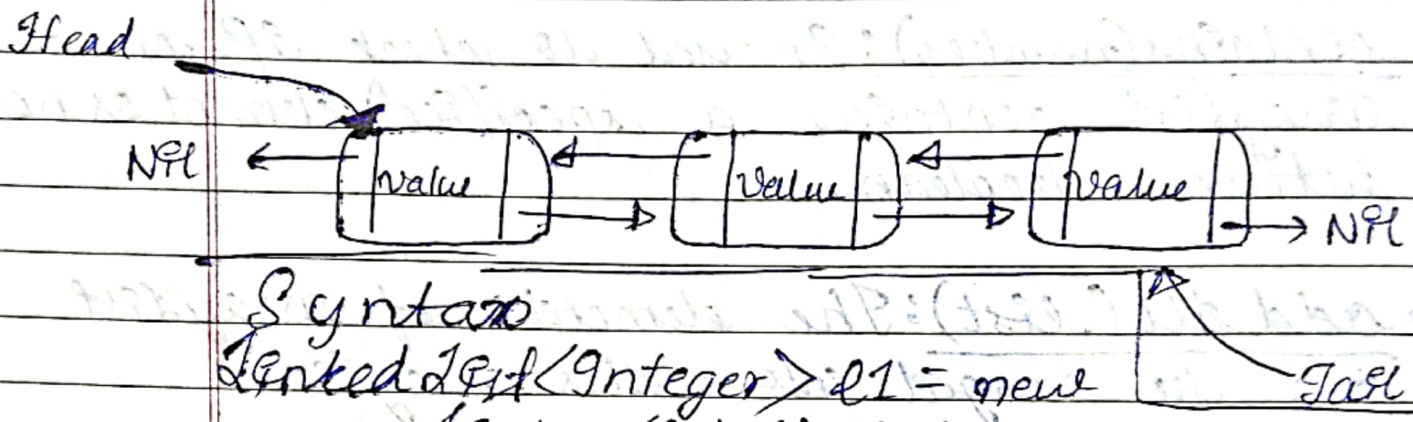
→ get(index): To print element at a particular index

→ lastIndexOf(number): To get the last index of a number.

Linked List in Java

- The LinkedList class in Java provides us with the doubly linked list data structure.

- Each element of the linked list is known as node.
- Each node points to the address of the next node & previous node.



Syntax

```
LinkedList<Integer> l1 = new
        LinkedList<>();
```

→ • Linked lists are preferred over the arraylist because the insertion & deletion can be done in a constant time. But, in arrays, if we want to add or delete an element in the list then, we need to shift all other elements.

- In a linked list, it is impossible to directly access an element because we need to traverse the whole linked list to get the desired element.

ARRAYLIST vs LINKED LIST

→ Although ArrayList & LinkedList implement the List interface & have the same methods, it is important to understand when to use which one.

→ The insertion & deletion can be done

in constant time in LinkedList, so it is best to use the LinkedList when you need to add or remove elements frequently.

→ Use ArrayList when you want to access the random elements frequently, as it can't be done in a LinkedList in const. time.

Methods: [list.method(parameters)]

It has all the methods of ArrayList with same syntax:

→ set(number): It is used to change an already existing element of a LinkedList.

→ addLast(number): To add an element at the end of LinkedList.

→ addFirst(number): To add an element at the start of LinkedList.

→

~~ARRAY~~

ArrayDeque in Java

ArrayDeque - Resizable ~~at~~ array +

Deque interface

→ It implements the Queue ~~interface~~ & Deque interface.

→ There are no capacity restrictions & provides us the facility to add or remove element from b.s. of queue.

→ Also known as Array Double Ended Queue.

→ It is fastes than Linked list & ~~Queue~~ stack.

Constructors of ArrayDeque class:

⇒ ArrayDeque(): Used to create an empty array deque that has the capacity to hold 16 elements

⇒ ArrayDeque(int numElements):

Used to create an empty array deque that has the capacity to hold the specified number of elements.

⇒ ArrayDeque(Collection<? extends E>c)

Used to create an array deque containing all the elements of the specified collections.

METHODS (ad.method(parameter))

Inserting

→ Insertion at front: add(), offerFirst(), addFirst()

→ Insertion at End : addLast(), offerLast()

Accessing elements

→ Accessing an element from the head :

getFirst(), peekFirst()

→ Accessing the last element:

getLast(), peekLast()

Removing an Element

→ Removing the first element:

removeFirst(), pollFirst(),

→ removeFirst() → throws exception if queue is empty
→ pollFirst() → returns null if queue is empty.

→ Removing the last element

removeLast(), pollLast()

Hashing in Java:

Hashing is the technique to convert the range of key-value pairs to a range of endices. In hashing, we use hash functions to map keys to remove some values.

E.g. Let arr = [11, 33, 22, 14]

hashIndex = (key % 10)

Keys	H(x) = Keys % 10	Hashtable
		0
11	11 % 10 = 01	1
33	33 % 10 = 3	3
22	22 % 10 = 2	2
14	14 % 10 = 4	4

Collision: The hash function may map 2 key values to a single index. Such a situation is known as collision.

Collision: The hash func^n may map 2 key values to a single index. Such a situation is known as a collision.

E.g. List = [22, 42, 64, 71]
H(x) = key % 10

Keys	H(x) = Keys % 10	Hash Table
		0
71	11 % 10 = 1	1
22	22 % 10 = 2	2
42	42 % 10 = 2	
		3
64	64 % 10 = 4	4

Here 22 & 42 are mapped to the index no. 2.
∴ we need to avoid collision.

→ Open Addressing → Chaining

HashSet in Java

→ Hashset uses a hashtable for storing the elements
→ It implements the set interface.
→ Duplicate values are not allowed.
→ Before storing any object, the hashset uses the hashCode() & equals() method to check any duplicate entry in hashtable
→ Allows null value
→ Best suited for search operations.

Constructors for Hashset:

Hashset():

This constructor is used to create a new empty Hashset that can store 16 elements & have load factor of 0.75.

~~Hashset(int int~~ Hashset(int initialCapacity):

This constructor is used to create a new empty Hashset which has capacity to store the specified number of elements & having load factor of 0.75.

Hashset(int initialCapacity, float loadFactor)

This constructor is used to create a Hashset with the specified capacity & load factor.

Hashset(Collection<? extends E>c):

Used to create a Hashset using the elements of collection c.

Methods (HashSet.method(parameter))

Inserting elements

1) ~~add()~~.

1) add(numbers):

→ The insertion order of the elements does not remain preserved in Hashset.

→ All duplicate elements are ignored because set contains only unique values

* Removing elements.

2) ~~remove()~~: remove(numbers)

→ Does not throw exception if specified element not present in the Hashset.

* Checking if it's empty or not

1) isEmpty():
Returns boolean.

* Removing all elements

1) clear():
removes all elements

* Printing size of Hashset

1) size()

Code:

```
import java.util.*;

public class CWH extends Thread{

    p s v m( ){

HashSet<Integer> myHash = new HashSet<>
                                (6, 0.5f);

myHash.add(6);
—"— —— "— (8);
—"— —— "—(3);
—"— —— "—(11);
—"— —— "—(11); // Element will be
                   ignored

sout ln(myHash);
```

Hash Map in Java

A Hashmap stores items in key/value pairs & you can access them by an index of another type (e.g. a Int or String).

→ One object is used as a key (ind-x) to another object (value).

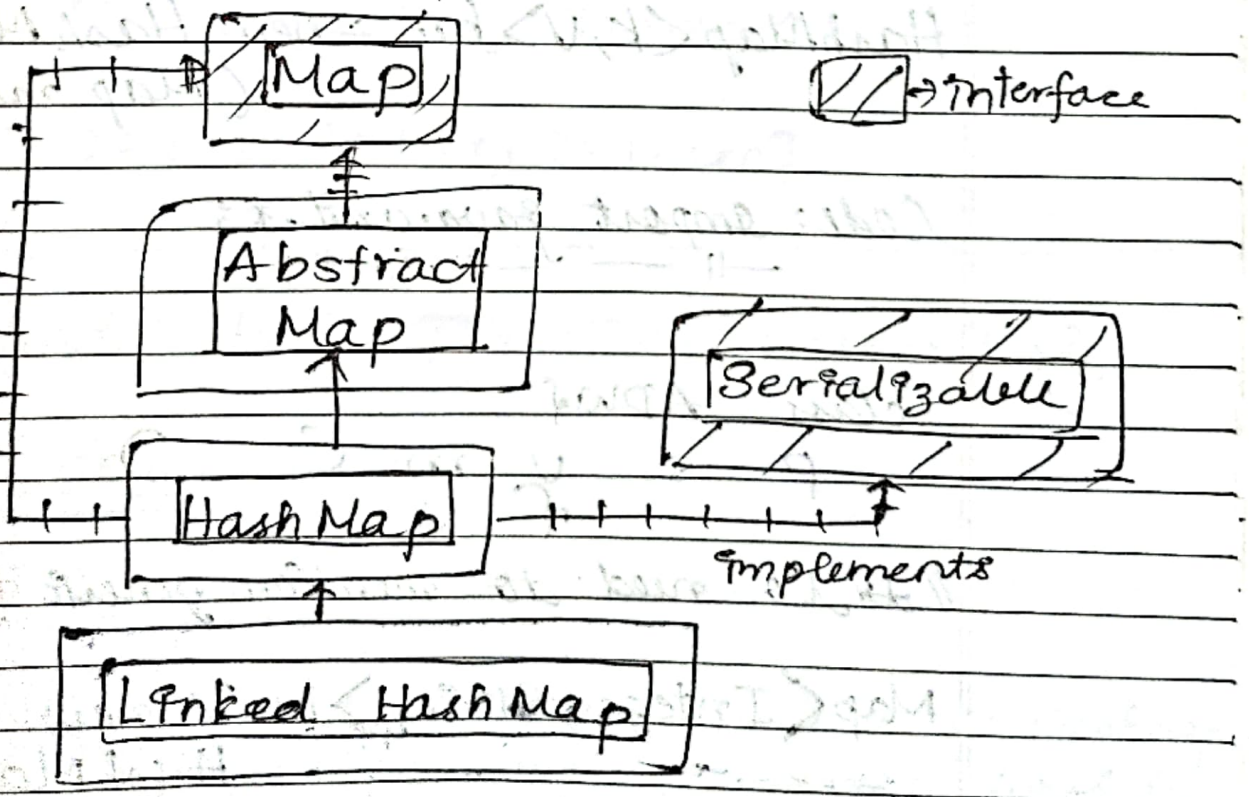
→ If you try to enter duplicate key it would replace element of corresponding key.

→ Hashmap is similar to Hashtable but it's unsynchronized

→ It allows us to store null keys as well, but there should be only one null key object & there can be any number of null values.

~~Syntax:~~

~~public class HashMap<K,V> ex~~



Constructors:

1) HashMap()
2) HashMap(int initialCapacity)
3) HashMap(int initialCapacity, float loadFact.)
4) HashMap(Map.map)

Syntax:

HashMap<K, V> hm = new HashMap<K, V>();

~~Initial~~

It's initial capacity = 16 loadFactor = 0.75

HashMap(Map map): It creates an instance of HashMap with the same mappings as the specified map.

HashMap<K, V> hm = new HashMap<K, V>(Map map);

Code: import java.util.*;
// import java.io.*;

class ADW{
p s v m ( ){
}
// It No need to mention generic type twice

Map<Integer, String> hm1 = new HashMap<>();

// Add elements

```
hm1.put(1, "one");
hm1.put(2, "two");
hm1.put(3, "three");

HashMap<Integer, String> hm2 =
    new HashMap<Integer, String>(hm1);
    sout("Map 1 : " + hm1);
    sout("Map 2 : " + hm2);
  }
}
```

o/p:

Map 1 : {1=one, 2 = two, 3 =three}

Map2 : {1=one, 2 = two, 3 =three}

Methods(HashMap.method(parameters))

1) Add Items

→ put() →

The insertion order is not retained in HashMap. Internally, for everyelement, a seperate hash is

generated & the elements are indexed based on the hash to make it efficient.

2) Changing elements :

→ put() :
Since the elements on the map are indexed using the keys, the value of the key can be changed by simply inserting the updated value of key.

3) Removing element

→ remove() :
It takes the key value & removes mapping for a key if present

4) Access an item

→ get() :

5) Size of HashMap

→ size()

6) Traversal of Hashmap

We can use the Iterator interface to traverse over any structure of the

Collection Framework. Since Iterators work with only one type of data we use ~~Entry~~ Entry <?,?> to resolve 2 seperate types into a compatible format. Using next() method we print the entries.

Code:

```
import java.util.*;

public class XYZ{
    p s v m( ){

HashMap<String, Integer> map=
                new HashMap<>();

map.put("Vishal", 10);
--11------("Sachin", 20);
--11------("Adwait", 30);

for(Map.Entry<String, Integer>e: map.entry
                                    set())

sout("Key: " + e.getKey()+
    "Value:" + e.getValue());
    }
}
```

O/P :

Key: vaibhav Value: 20
" : Adwait Value: 30
" : Sachin "

Key : Adwait Value : 30
" : Vishal " : 10
" : Sachin " : 20

NOTE: HashMap doesn't allow duplicate keys but allows duplicate values. That means a single key can contain more than 1 values

For e.g. code

```
import java.util.*;

public class Main {

    p s v m ( )
    {
HashMap<String, String> capitalCities =

        new HashMap<String, String>();

capitalCities.put("England", "London");
    " ------- ("Germany", "Berlin");
    " ------- ("Norway", "Oslo");
    " ------- ("USA", "Washington");
```

~~for(String i~~
sout("Countries");
~~for(String i;~~
for(String i : capitalCities.keySet()){

sout(i);

}

sout("Capitals");

for(String i ; capitalCities.values()){

sout(i);

}

}

O/P

USA

O/P

~~sout~~ Countries
USA
Norway
England
Germany

Capitals
Washington
Oslo
London
Berlin

Performance of HashMap depends on

→ Initial capacity
→ Load factor

→ Initial capacity: It is the no. of buckets hashmap can hold when instantiated. Initially 16 K-V pairs

→ Load factor: It is the percentage fill of buckets after which rehashing takes place. If is 0.75f by default, hence rehashing takes place after filling 75% of capacity.

→ Threshold: Product of load factor & initial capacity, initially (16 x 0.75 = 12) i.e. Rehashing takes place after inserting 12 elements

→ Rehashing: Process of doubling the capacity after reaching threshold.

Hashtable

→ A Hashtable is an array of a list. Each list is known as bucket. The pos° of bucket is identified by calling hashcode() method.

→ Contains unique elements

→ doesn't allow null key or value

→ Synchronized

→ Init. cap. = 11 load factor = 0.75