Java Collections Framework

→ At the top level of collections framework is the 'Collections' class.

Side Note:-
To format integers while printing use %02d %03d etc
[illegible] along with String.format

example:- int n = 5;
String.format("%02d", n);
will return formatted string i.e, 05

Collection's List Interface

Normally we can create arraylist & linkedlist as below:-

ArrayList <String> name = new ArrayList<String>();

↓ it doesnot take primitive datatypes like int, double etc. instead Provide Integer, Double etc i.e; wrapper classes or userdefined classes

↓ (It can also be left empty above JDK 8)

LinkedList<Integer> ll = new LinkedList<>();

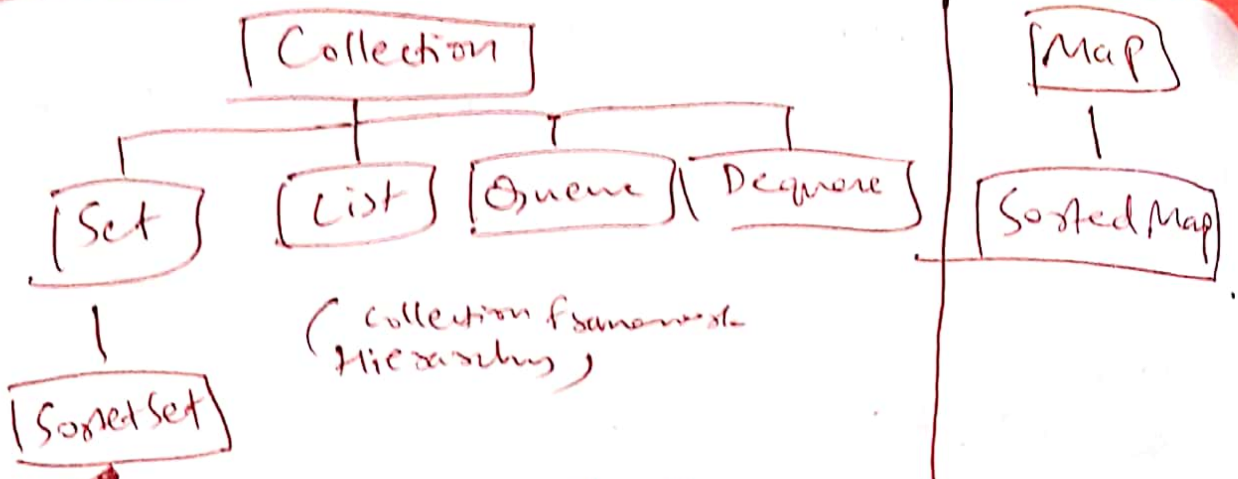
But using List interface we can make it more generic like:-

List<String> name = new ArrayList<>();
List<String> ll = new LinkedList<>();

And using 'Collection' we can make it even more generic

```
Collection<String> name = new LinkedList<>();
  "        "       "   =  "  HashSet<>();
  "        "       "   =  "  ArrayList<>();
  "        "       "   =  "  LinkedHashSet<>();
                                        etc.
```

Scanned with CamScanner



* The above all are Interfaces.

* A Map is not a true collection.

Binary Search Method

Collection Class also provides binary search method.

~~[illegible] to use~~

Side note :-
We can implement 'Comparable' interface to override default compareTo() method of String class. CompareTo returns an int. String1.compareTo(String2);

* compareToIgnoreCase() is another method.

Collections. binarySearch(___ , ___ , null)

Collection on which to be searched on.

target item or variable name

uses inbuilt comparison operation.

returns index of found element else -1.

Side Note :-
System.out.println -> also prints a new line
System.out.print -> normal print without new line.

Side Note:-

For sorting arrays in java, use Arrays.sort(arrayname);
For sorting collections " " , use Collections.sort(collectionname);

For changing collections in reverse order, use
Collections.reverse(collectionsname);

For shuffling elements in pseudo random order.

Collections.shuffle(collections name);

For min and max elements

Collections.min(collections name); returns min
Collections.max(collections name); and max item.
Internally these use compareTo method

For swapping two elements
Collections.swap(collectionname, index1, index2);

Collections Copy Method

It takes 2 parameters → destination & source list.
1st 2nd
1st param must be a generic type i.e.,
a collection or iterable or a list
2nd param must be a list
Collections.copy(newList, oldList)

newList has to be initialized before this to work and not just declared.

Comparator Interface

Unlike Comparable we dont implement comparator interface instead an object of type Comparator can be created using compare() method.
We can also create a new class and implement comparator interface.

Example ~~of~~ using anonymous class i.e. creating object

```
static final Comparator<String> NAME = new
                          Comparator<String>(){

   @Override
   public int compare(String a, String b){

   }

};
```

To use comparator example,
Collections.sort(collectionname, Comparatorname);
Now this sort function doesn't use inbuilt functionality but our compator's compare method alongside normal inbuilt functionality.

We can also split the declaration and initialization of comparator for example for above it will be

```
static final Comparator<String> NAME;
static{
   NAME = new Comparator<String>() {
   @override
   ...
   };
};
```

~~[illegible]~~ using a static initialization block

Map Interface

→ Key value pairs.
→ [illegible] Cannot have duplicate keys
→ Each key can only map to a single value.
Hash Map, LinkedHashMap, TreeMap

* Map<String, String> name = new HashMap<>();
* name.put("Java", "OOPS Java lang");
name.put("Python", "Another lang");

* name.get("Java");
get() fn uses key to get value.
if you use same key twice using put method it overwrites the old value. In case there was an old value the put() method returns that value while also overwriting it with new value. Else if there was no old value put method returns null and just adds the value.

* name.containsKey("Java");
containsKey() methods returns boolean value depending on the existence of key.

* name.keySet();
will return a set of all existing keys.
To loop through all key value pairs we can do,

```
for(String key: name.keySet()) {
    sout(key + ":" + name.get(key));
}
```

Note:- There is no ordering for key value pairs in HashMap

* name.remove("Java");

removes that key value pair.

name.remove("Java", "Hahaha");

* remove method can also remove using both key and value if it matches it will remove it else not. In the above case it won't cuz the value is wrong.

remove method returns true or false.

* name.replace("Java", "A beautiful lang");

* replace() method will replace it's value if the key exists otherwise it won't and will return null.

replace() can also taken 3 params, i.e.,

key, old value, new value

if key & old value exist & match then it replaces.

Side note:-
To use delimeters in strings,

String[] a = StringB.split(" ")

split() method provides delimeter option

The above will split the words and store in array

Sets & HashSet

* Set has no defined ordering
* Set cannot contain duplicates

Basic Methods
add() remove() clear() size() isEmpty() contains()

* HashSet is like a HashMap internally but it only stores the keys and values are some dummy objects.

```
Set<String> name = new HashSet<>();
```

Union of sets

we can get union of two sets by creating a new set and using the addAll() method

for example :-

```
Set<HeavenlyBody> moons = new HashSet<>();
for (HeavenlyBody planet: planets) {
    moons.addAll(planet.getSatellites());
}
```

obj.getClass()
obj.getName()

equals() and hashCode()

It is recommended that whenever we use sets & maps to override these two methods because in case of equals() the default java implementation works on referential equality. That is, if both point to the same object they are equal else not. This can allow us to have duplicate keys with different values if they are different objects. Hence it is recommended to override equals().

Hashing

When storing objects in a hashed collections such as hashsets & hashmap, think of the collection having a number of buckets to store the objects in. The hashcode determines which bucket the object is gonna go to.

"instanceof" keyword

```
Set<Integer> squares = new HashSet<>();
Set <Integer> cubes = new HashSet<>();
```

Add 100 squares and cubes in them.

```
Set<Integer> union = new HashSet<>(squares);
Set <Integer> intersection = new HashSet<>(squares);
intersection.retainAll(cubes)
```

↓

retain all methods ~~[illegible]~~ retains only those elements which are present already and also in cubes and removes everything else.

Math.sqrt(); → square root both return double
Math.cbrt(); → cube root

Set Interface Bulk operations

S1.containsAll(S2) – returns true if S2 is a subset of S1
S1.addAll(S2) – transforms S1 into union of S1 and S2.
S1.retainAll(S2) – transforms S1 into intersection of S1 & S2
S1.removeAll(S2) – transforms S1 into the set difference of (asymmetric) S1 & S2.

Side note :-
Arrays. ~~[illegible]~~ asList(arrayname); will return a list of the array elements

Side Note :-
enum interface
Used for grouping constants generally example :-

```
Public enum WeekDays {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY
}
```

Later we can create an enum variable like
WeekDays wd;

Sorted Collections

HashMap
HashSet $\Big\} \rightarrow$ Unsorted, chaotic

LinkedHashMap
LinkedHashSet $\Big\} \rightarrow$ Ordered. TreeMap is also ordered.

LinkedHashMap can be ultimately extended to use a TreeMap.

HashMap's getOrDefault (key, value); method. will get the item if it exists and if it doesn't exist it returns the given ~~[illegible]~~ value in the method.

→ When returning a Map, u can return mapName;
or
return Collections.unmodifiableMap(mapName);
so that it cannot be modified.

→ mapName.entrySet() method is used to create a set out of the same elements contained in the hashMap.

→ mapName.keySet() method returns a set of keys.

→ mapName.clear() empties Map.