Collections Framework

- ***present in java.util package***

- List
  - Maintains insertion order
  - Allows duplicate values
  - Null values allowed
- Set
  - Doesn't allow duplicate values
  - Doesn't maintain insertion order.
  - One Null Value allowed. (Except for treeSet)
- Map
  - Stores data in key value format
  - Key is unique
  - One null key and multiple null values allowed

- ***Tree type data structure doesn't allow null values because it internally users compareTo() for comparing values which returns NPE on comparing will null whereas other data structures uses equals() for comparison which doesn't throw NPE.***

ListInterface

- **ArrayList**
  - Best for storing and accessing data.
  - Uses dynamic array.
  - Initial capacity = 10.
  - Size increases by n+(n/2)+1 on exceeding the size.
  - Non synchronised.

- **LinkedList**
  - Best for insert, update, delete.
  - Uses doubly linked list.
  - Doesn't have initial capacity.
  - Non synchronised.

- **Vector**
    - Legacy class.
    - Initial capacity = 10.
    - Size increases by 100% on exceeding the size.
    - Synchronised.
    - Slow

SetInterface

- **HashSet**
    - Best for searching operations
    - Contains unique elements
    - Allows null
    - Non synchronised
    - Default size = 16, load factor = 0.75
    - Doesn’t maintain insertion order

- **LinkedHashSet**
    - Same as HashSet but maintains insertion order

- **TreeSet**
    - Same as HashSet but stores elements in ascending order(default) and doesn’t allow null values

MapInterface

- **HashMap**
    - Unique keys.
    - One null key and multiple null values allowed.
    - Non synchronised.
    - Doesn’t maintain insertion order.
    - Default size = 16, Load factor = 0.75.
    - Size is doubled as soon as it reaches threshold.

- **LinkedHashMap**
    - Same as HashMap but maintains insertion order.

- **TreeMap**
    - TreeMap class is a red-black tree based implementation.
    - Same as HashMap but stores values in ascending order of keys and doesn’t allow null keys (multiple null values are allowed).

- **HashTable**
    - Legacy class.
    - Contains unique elements.
    - Doesn't allow null key or values.
    - Synchronised.
    - Initial capacity = 11, Load Factor = 0.75.

- ***HashSet internally uses HashMap. The values are stored as key in HashMap. This is done as the functionality of HashSet and HashMap is same in terms of uniqueness and hence the maintenance will be less.***

QueueInterface

- Stores elements in FIFO(First In First Out) manner.
- Methods are :-
    - **boolean add(object)** -> insert element and returns true on success.
    - **boolean offer(object)** -> insert element.
    - **Object remove()** -> removes head.
    - **Object poll()** -> removes head or returns null if queue is empty.
    - **Object element()** -> retrieves head but doesn't remove.
    - **Object peek()** -> retrieves head but doesn't remove or returns null if queue is empty.

- ***The difference between add(), remove(), element() and offer(), poll(),peek() is that the latter group of methods(p,o,p) returns a null value in case they are not able to add(in case of fixed size queue) or remove an element whereas the other methods throw NPE.***

- ***Null Value is not allowed in queue because if p,o,p methods() returns null, we wouldn't be able to tell the difference whether the queue is empty or null is an element present in the queue.***

DequeInterface

- Double ended queue
- Methods same as queue with few additional methods.
- add() and addLast(), poll() and pollLast() and similar types of methods have same behaviour.

Hashing Mechanism

- Process of converting object into integer form using hashCode(). This hash value is a memory reference in integer form.
-

Internal Working of HashMap

- **put()**
  - Calculate hashCode of key. A bucket is selected based on the generated hashCode. If key is null bucket-0 as hashCode of null is zero.
  - Place the Entry object in that bucket.
  - If an element is already present in that bucket the next element is appended to the linkedList.
  - If an element is having same key, the bucket is found using hashCode(). Then equals() method is used to check the equality of keys. When a match is found, the value is replaced with the new one.
- **get()**
  - Bucket is found using hashCode() on key. If element is present it returns the object else returns null.
  - If more than one elements are present in that bucket, equals() is used to match the keys.

Contract between equals() and hashCode()

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.

- ***Eg:-***
  - ***I have a class Fruit which is having only one data member, namely fruitName.***

```
class Fruit {
    public String fruitName;

    Fruit(String fruitName)
    {
```

```
            this.fruitName = fruitName;
        }
    }
```

- *I create two objects and add them in a hashMap*
  obj1 = new Fruit("Apple");
  obj2 = new Fruit("Apple");

- *The size of the map will be 2 as JVM considers both objects as different.*
- *Now if I override equals() method to check for the value rather than the memory location, which is the default functionality, both obj1 and obj2 will be equal and hence the size of the map should be 1 but since I haven't override the hashcode() method it would still generate different hashcode and therefore it would treat them as different objects. This is the violation of first contract.*
- *Therefore whenever equals() is overridden an equivalent implementation of hashCode() should also be provided and vice versa.*

Comparable Interface

- *present in java.lang package*

- Provides single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.
- Has only one method
  - **public int compareTo(Object obj)** which returns
    - positive integer, if the current object is greater than the specified object.
    - negative integer, if the current object is less than the specified object.
    - zero, if the current object is equal to the specified object.

How to use

- Implement Comparable interface for the custom class and write the logic in compareTo() method.
- call Collections.sort(list)

Comparator Interface

- ***present in java.util package***

- Provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.
- Has two methods
    - public int compare(Object obj1, Object obj2)
    - public boolean equals(Object obj)

How to use

- Define a class and implement Comparator interface. Eg:- **class AgeComparator implements Comparator<Student>**
- write the logic in compare(Object obj1, Object obj2) method.
- Call with comparator name. Eg:- **Collections.sort(list, new AgeComparator)**