Language Advance Concept : Collections

AGENDA

- → Java Collection Framework
- → Collection Interface
- → Interfaces that extends Collection Interface:
- → Map Interface
- → Comparable
- → Comparators

JAVA COLLECTION FRAMEWORK

→ Any Group of Individual objects which are represented as a single unit is known as a collection of objects

→ A framework is a set of classes & interfaces which provide a ready made architecture.

what is Interface ?

Eg:-

```
Interface XYZ {
    void function();
    int function();
}
```

```
class temp implementing XYZ{
    void function() {
    }
    int function() {
    }
}
```

Eg:

```
Interface Car {
    function start();
    function stop();
}
```

```
class Maruti implementing Car {
    function start() {
    }
    function stop() {
    }
}
```

Eg 3

arrayList → list.add()

Vector → vector.add()

★ Interfaces can also be extended.

Eg 3

```
Interface RBI {
    guidline1();
    guidline2();
}
```

```
Class MyBank implements RBI {
    guidline1() {
        ==
    }
    guidline2() {
        ==
    }
}
```

JCF (Resume)

→ Java Collection Framework (JCF) is a set of classes & Interfaces that implements commonly reusable collection data structure like List, Set, Queue, Map etc. The JCF is organized into interfaces & implementation of those interfaces. The interfaces define the functionality of the collection data structures, & the implementation provide concreate implementation of those interfaces.

Q Need of Java Collection Framework?

→ Before the Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors, or Hashtable

→ All of these collections had no common interface. Therefore, through the main aim of all the collections is the same, the implementations of all these collections was defined independently & had no correlation among them. Hence, it is very difficult for the users to remember all the different methods, syntax & constructors present in every collection class.

Eg :-

```
class CollectionDemo {
    Public static void main ( String [] args) {
        int arr[] = new int[]{ 1, 2, 3, 4};

        // vector & hashtable

        Vector <Integer> v = new Vector();
        Hashtable <Integer, String> h
                          = new Vector();

        v. addElement (1);
        v. addElement (2);

        h. put (1, " Hey ");
        h. put (2, " all ");

        SOPC arr[0]);
        SOPC v. elementAt(0));
        SOPC h. get (1));
    }
}
```

Different ways of insertion.

=> Advantages of JCF

① **Consistent API :-** The API has a basic set of interfaces like Collection, Set, List or Map, all the classes (Arraylist, LinkedList, Vector etc) that implement these interfaces have some common set of methods.

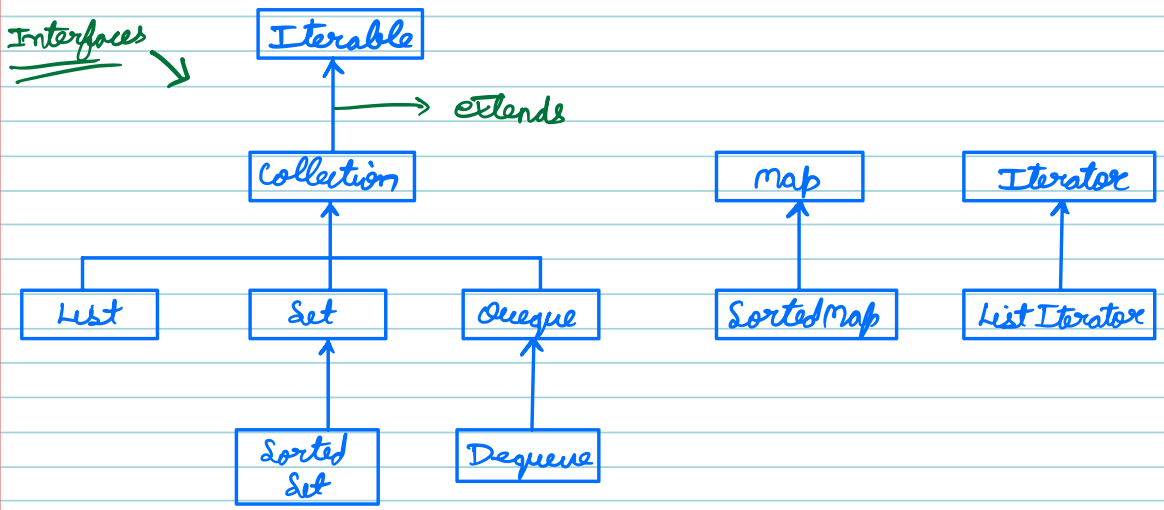
② **Reduces programming effort :-** A programmer doesn't have to worry about the design of the Collection but rather he can focus on its best use in his program. Therefore, the basic concept of OOP (i.e.) abstraction has been successfully implemented.

③ Increase program speed & quality :- Programmer need not think of best implementation of a specific data structure. He can simply use best implementation.

Q How JCF looks Life ?

* Java.Util.* → Contains all classes & objects that are categorized into Java Collection Framework.

Java Collection Framework



Iterable Interface :- ① This is root of collection Framework

② Purpose is to provide an iterator for the Collections.

③ Contains only one abstract method which is the iterator.

COLLECTION INTERFACE

→ The Collection interface is the root interface of the java collections Framework. It declares the core methods that all collections will have.

→ The Collection Interface is a part of the java.util package.

→ The JDK does not provide any direct implementation of this interface: it provides implementation of more specific sub-interface like Set & List. This interface is typically used to pass collection around & manipulate them where maximum generality is desired.

→ The Collection interface is not directly implemented by any class. However, it is implemented indirectly via its subtypes or subinterfaces like List, Queue & Set. For Example, the HashSet class implements the Set interface which is a subinterface of the Collection interface.

→ It implements the iterable interface.

⇒ Function of Collection Interface

↳ **add()** - insert 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.

⇒ Interfaces that extends Collection Interface

① List

↳ The interface is dedicated to the data of the list type in which we can store all the ordered collections of the objects. This also allows duplicate data to be present in it.

② Set

↳ A Set is an unordered Collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects & wish to store only the unique objects.

③ Sorted Set

↳ This interface is very similar to the Set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface & is used to handle the data which needs to be sorted.

④ Map

↳ A Map is a data structure that supports the Key - value pair for mapping the data. This interface doesn't support duplicate keys because the same key cannot have multiple mappings, however, it allows duplicate values in different keys. A map is useful if there is data & we wish to perform operation on the basis of the key.

⑤ Queue

↳ As the name suggests, a queue interface maintains the FIFO (First in First out) order similar to a real world queue line. The interface is dedicated to storing all the elements where the order of the elements matter. For example, whenever we try to book a ticket, the ticket are sold at the first come first serve basis. Therefore the person whose request arrives first into the queue gets the ticket.

⑥ Deque

↳ This is a very slight variation of the queue data structure. Deque, also known as a doubled-ended queue, is a data structure where we can add & remove the elements from both the ends of the queue. This interface extends the queue interface.

List Interface

Collection
↑
List

↳ Ordered collection of objects in which duplicate values can be stored.

↳ List preserves insertion order, therefore it allows positional access & insertion of elements.

↳ Some implementations of List interface.

① Array List :- Resizable - array implementation

② Vector :- Synchronized Resizable - array implementation

③ Stack :- Subclass of Vector that implements a standard last in - first out stack.

④ Linked - List :- Doubly - Linked list implementation of the List & Deque interfaces.

↳ Since List is an interface, hence object can't be created.

DECLATION

```
Public interface List<E> extends Collection<E>;
```

⇒ ArrayList

↳ Resizable array also known as Dynamic array. It provides interface to work with Dynamically sized list of elements, allowing efficient insertion, deletion & random access.

Here's how its Typically implemented.

1. **Backing Array**: The core of an ArrayList is an underlying array that holds the elements. This array is initially created with a default size, and elements are stored sequentially in it.
2. **Resizing**: As elements are added to the ArrayList, the backing array may become full. When this happens, a new larger array is created, and the existing elements are copied from the old array to the new one. This process is called resizing or resizing the array.
3. **Dynamic Sizing**: The resizing operation ensures that the ArrayList can dynamically grow or shrink in size as needed. This dynamic sizing is a key feature that differentiates it from a regular array.
4. **Index-Based Access**: ArrayList allows elements to be accessed by their index. This is achieved by directly accessing the corresponding element in the backing array using the index.
5. **Insertion and Deletion**: When an element is inserted at a specific index or removed from the ArrayList, the other elements may need to be shifted to accommodate the change. This can involve moving multiple elements within the array.
6. **Efficiency**: ArrayList provides efficient constant-time (O(1)) access to elements by index. However, insertion or deletion operations at the beginning or middle of the list may require shifting elements and take linear time (O(n)), where n is the number of elements.
7. **Automatic Resizing**: Java's ArrayList uses automatic resizing strategies to ensure that the array is appropriately resized when needed. The exact resizing strategy can vary across different implementations and versions of Java.

Ex:- `List <Integer> temp = new ArrayList <Integer> ();`

⇒ Vector

↳ A Vector provide us with dynamic arrays in Java. May be slower than standard array but helps where lot of Manipulation in the array is needed. Similar to ArrayList with primary Difference that Vector is Synchronized while ArrayList is not.

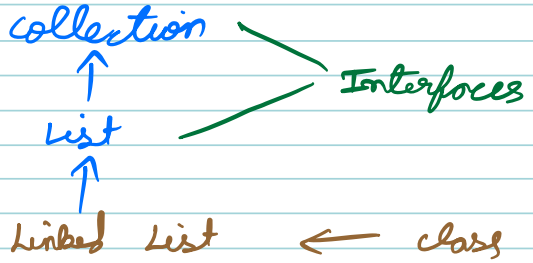
Eg:- `List < Integer > temp = new Vector <Integer> ();`

⇒ Stack

↳ Stack is a class implemented in collection framework & extends the Vector class Modal & implements the Stack data structure. Follows Last in first out principle. In addition to basic push & pop, the class provide three more functions of empty, search & peek.

Eg :- `List <Integer> temp = new Stack <Integer> ();`

⇒ Linked List



↳ Linear data structure where the elements are not stored in contiguous location & every element is seperate object with a data part & address part.

↳ The elements are linked using pointers & addresses.

↳ Each element is known as Node.

↳ Due to the dynamically & ease of insertions & deletion, they are preffered over the arrays.

Eg :- `List <Integer> temp = new Linked List <Integer> ();`

Set Interface

Collection
↑
Set

↳ Can store atmox 1 null

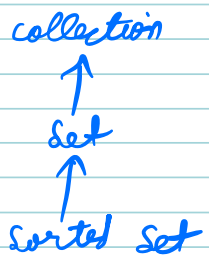
↳ Inherit methods from collection interface & adds a feature that restrict the insertion of the duplicate elements.

Declaration

```
Public Interface Set < > extends Collection < >;
```

⇒ HashSet ⇒ Widely Used Implementation of Set Interface.

SORTED SET INTERFACE



↳ Inherit methods from Set interface & adds a feature that stores all the elements in this interface in sorted manner.

Declaration

```
Public interface SortedSet < > extends Set < >;
```

⇒ Tree Set ⇒ Implementation of SortedSet

↳ TreeSet internally use Tree data structure to store.

MAP INTERFACE

↳ Present in java.util package & represents mapping b/w a Key & a value.

↳ Not a subtype of Collection interface

↑ This is a reason why it behaves differently.

↳ contains unique Key.

↳ 2 type of Map Interfaces.

① Map
② SortedMap

↳ implementation of Map & SortedMap

① HashMap
② LinkedHashMap
③ TreeMap

HashMap	LinkedHashMap	TreeMap
① Implements Map interface	① Implement Map interface with some additional method.	① Implement SortedMap.
② Random order of Key	② Maintains SortedOrder of insertion	② Maintains SortedOrder.

Queue Interface

↳ Present in java.util package & extends the Collection Interface.

↳ Hold the elements about to be processed in FIFO (First in First out) order.

↳ It is an ordered list of objects with its use limited to inserting elements at end of the list & deleting element from the start of the list.

Declaration

public interface Queue<> extends Collection;

↳ classes which implement Queue.

① Priority Queue.

② Linked List

Deque Interface

↳ Its subtype of Queue Interface.

↳ Deque :- Double ended Queue

↳ Can act like stack as well as Insertion & Deletion could be done from both ends.

Declaration

Public interface Deque<> extends Queue<>;

↳ ArrayDeque & LinkedList are the classes which implements Deque interface.

↳ Some frequently used methods :-

- ↳ addLast()
- ↳ addFirst()
- ↳ removeLast()
- ↳ removeFirst()
- ↳ getLast()
- ↳ getFirst()

COMPARABLE

Eg:-

```
class Person {
    String name;
    int age;
    Person (x, y) {
        name = x;
        age = y;
    }
}

main () {
    Person[] P = new Person[4];
    P[0] = new Peron ("A", 10);
    P[1] = new Peron ("B", 20);
    P[2] = new Peron ("C", 30);
    P[3] = new Peron ("D", 40);
    Array.sort (P);
}
```

Q will it sort with Arrays.sort(P)?

↳ No, as compiler don't know how to compare 2 person.

⇒ So it our job to define how to compare.

Solution:-

```
class Person implements Comparable<Person> {
    String name;
    int age;
    Person(x, y) {
        name = x;
        age = y;
    }
    Public int compareTo(Person other) {
        if (this.age < other.age) {
            return -1; // put current before other
        } else if (this.age > other.age) {
            return 1; // put other before me
        } else {
            return 0; // Same
        }
    }
}
```

Must to implement

Comparator Interface

↳ Helps when we don't have access to modify the class which is trying to sort.

Eg:-

```
class Person {
    String name;
    int age;
    Public Person( String name, int age){
        this.name = name;
        this.age = age;
    }
}

class AgeComparator implements Comparator<Person>{
    public int compare( Person person1,
                        Person person2){
        return Integer.compare( Person1.age,
                                Person2.age);
    }
}
```

This can be customized.

```
class Example {
    main () {
        List<People> people = new ArrayList<X);
        people.add( new People ( "Alice", 28));
        People.add( new People ( "Bob", 22));
        people.add( new People ( "charlie", 25));
        Collection.sort( People, new
                                 AgeComparator());
```

```
}
}
// Print people using loop;
```