Java Collections Framework Part1

Java Collections Framework

Monday, 7 August 2023 8:16 PM

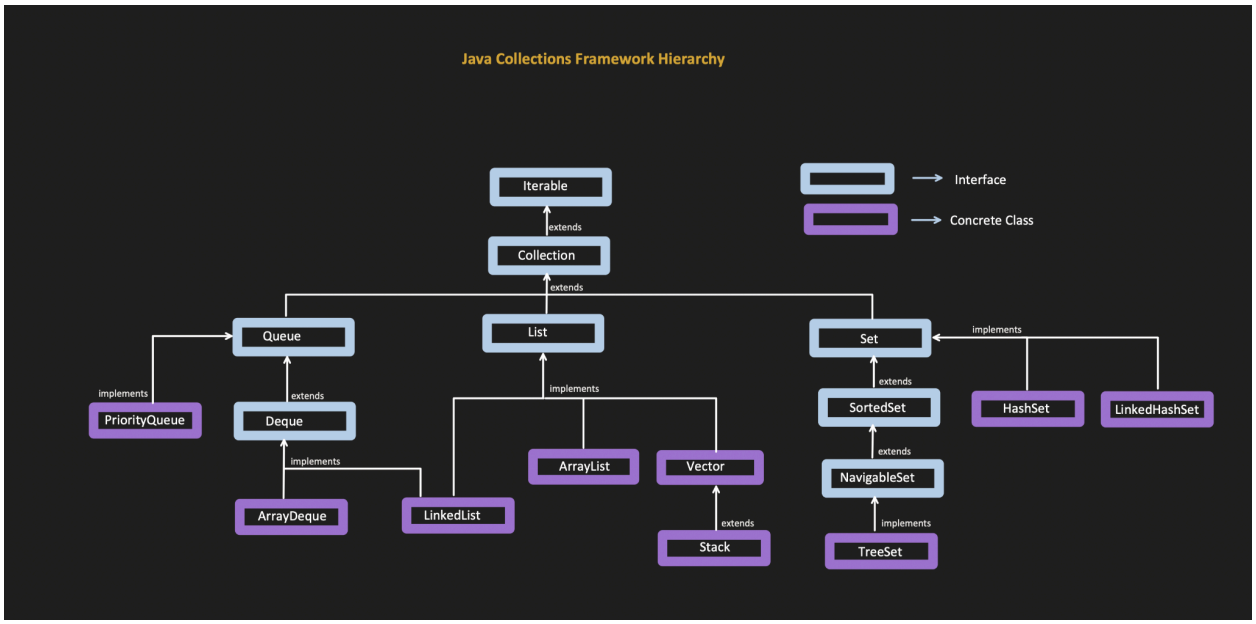
What is Java Collections Framework?

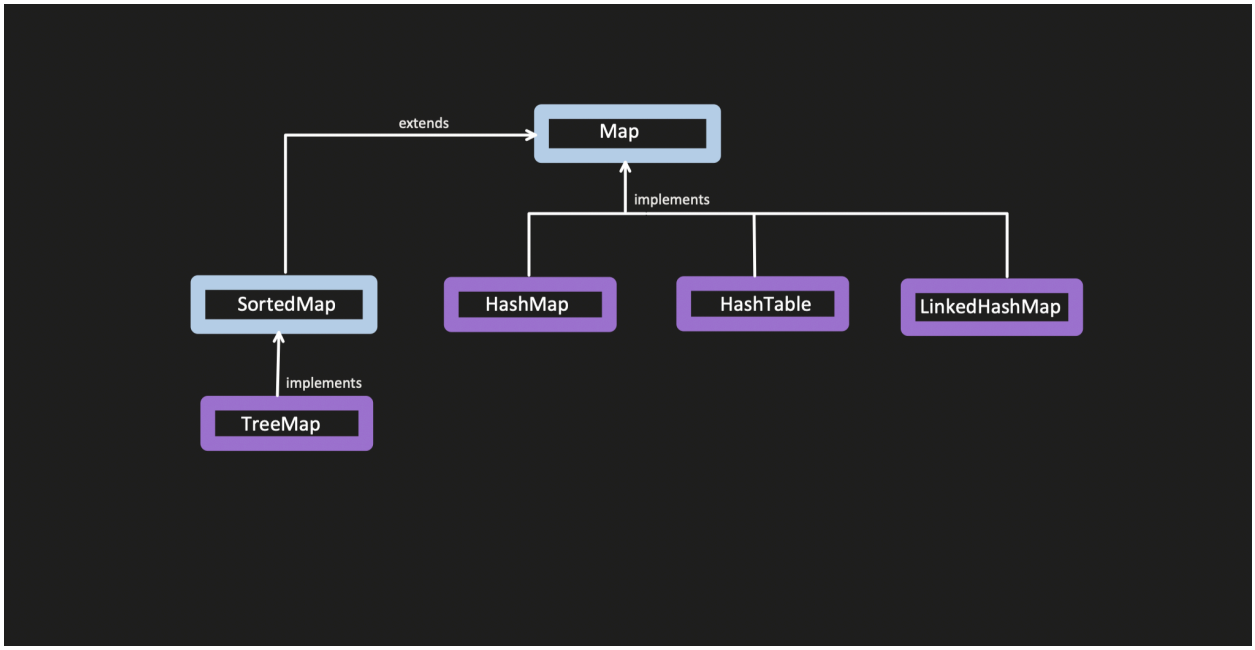
- Added in Java version 1.2
- Collections is nothing but a group of Objects.
- Present in ***java.util*** package.
- Framework provide us the architecture to manage these "group of objects" i.e. add, update, delete, search etc.

Why we need Java Collections Framework?

- Prior to JCF, we have Array, Vector, Hash tables.
- But problem with that is, there is no common interface, so its difficult to remember the methods for each.

```java
public class Main {
    public static void main(String[] args) {

        int arr[] = new int[4];
        //inset an element in an array
        arr[0] = 1;
        //get front element
        int val = arr[0];

        Vector<Integer> vector = new Vector();
        //insert an element in vector
        vector.add(1);
        //get element
        vector.get(0);
    }
}
```





Iterable: **Used to TRAVERSE the collection.**

Below are the methods which are frequently used.....

<table>
<tr><th>S.NO</th><th>METHODS</th><th>Available in</th><th>USAGE</th></tr>
<tr><td>1.</td><td>iterator()</td><td>Java1.5</td><td>It returns the Iterator object, which provides below methods to iterate the collection.
<table>
<tr><th>Method Name</th><th>Usage</th></tr>
<tr><td>hasNext()</td><td>Returns true,
if there are more elements in collection</td></tr>
<tr><td>next()</td><td>Returns the next element in the iteration</td></tr>
<tr><td>remove()</td><td>Removes the last element returned by iterator</td></tr>
</table></td></tr>
<tr><td>2.</td><td>forEach()</td><td>Java1.8</td><td>Iterate Collection using Lambda expression. Lambda expression is called for each element in the collection.</td></tr>
</table>

```java
public class Main {
    public static void main(String[] args) {

        List<Integer> values = new ArrayList<>();
        values.add(1);
        values.add(2);
        values.add(3);
        values.add(4);


        //using iterator
        System.out.println("Iterating the values using iterator method");
        Iterator<Integer> valuesIterator = values.iterator();
        while (valuesIterator.hasNext()){
            int val = valuesIterator.next();
            System.out.println(val);
            if(val == 3){
                valuesIterator.remove();
            }
        }

        System.out.println("Iterating the values using for-each loop");
        for(int val : values){
            System.out.println(val);
        }

        //using forEach method
        System.out.println("testing forEach method");
        values.forEach((Integer val) -> System.out.println(val));
    }
}
```

Output:

```
Iterating the values using iterator method
1
2
3
4
Iterating the values using for-each loop
1
2
4
testing forEach method
1
2
4
```

Collection: **It represents the group of objects. Its an interface which provides methods to work on group of objects.**

Below are the most common used methods which are implemented by its child classes like ArrayList, Stack, LinkedList etc.

S.NO	METHODS	Available in	USAGE
1.	size()	Java1.2	It returns the total number of elements present in the collection.
2.	isEmpty()	Java1.2	Used to check if collection is empty or has some value. It return true/false.
3.	contains()	Java1.2	Used to search an element in the collection, returns true/false.
4.	toArray()	Java1.2	It convert collection into an Array.
5.	add()	Java1.2	Used to insert an element in the collection.
6.	remove()	Java1.2	Used to remove an element from the collection.
7.	addAll()	Java1.2	Used to insert one collection in another collection.
8.	removeAll()	Java1.2	Remove all the elements from the collections, which are present in the collection passed in the parameter.
9.	clear()	Java1.2	Remove all the elements from the collection.
10.	equals()	Java1.2	Used to check if 2 collections are equal or not.
11.	stream() and parallelStream()	Java1.8	Provide effective way to work with collection like filtering, processing data etc.
12.	iterator()	Java1.2	As Iterable interface added in java 1.5, so before this, this method was used to iterate the collection and still can be used.

```java
public class Main {
    public static void main(String[] args) {
        List<Integer> values = new ArrayList<>();
        values.add(2);
        values.add(3);
        values.add(4);
        //size
        System.out.println("size: " + values.size());
        //isEmpty
        System.out.println("isEmpty: " + values.isEmpty());
        //contains
        System.out.println("contains: " + values.contains(5));
        //add
        values.add(5);
        System.out.println("added: " + values.contains(5));
        //remove using index
        values.remove( index: 3);
        System.out.println("removed using index: " + values.contains(5));
        //remove using Object, removes the first occurrence of the value
        values.remove(Integer.valueOf( i: 3));
        System.out.println("removed using object: " + values.contains(3));
        Stack<Integer> stackValues = new Stack<>();
        stackValues.add(6);
        stackValues.add(7);
        stackValues.add(8);
        //addAll
        values.addAll(stackValues);
        System.out.println("addAll test using containsAll: " + values.containsAll(stackValues));
        //containsAll
        values.remove(Integer.valueOf( i: 7));
        System.out.println("containsAll after removing 1 element: " + values.containsAll(stackValues));
        //removeAll
        values.removeAll(stackValues);
        System.out.println("removeAll: " + values.contains(8));
        //clear
        values.clear();
        System.out.println("clear: " + values.isEmpty());
    }
}
```

Output:

```
size: 3
isEmpty: false
contains: false
added: true
removed using index: false
removed using object: false
addAll test using containsAll: true
containsAll after removing 1 element: false
removeAll: false
clear: true
```

Collection vs Collections

Collection is part of Java Collection Framework. And its an interface, which expose various methods which is implemented by various collection classes like ArrayList, Stack, LinkedList etc.

Collections is a Utility class and provide static methods, which are used to operate on collections like sorting, swapping, searching , reverse, copy etc.

```java
public class Main {
    public static void main(String[] args) {
        List<Integer> values = new ArrayList<>();
        values.add(1);
        values.add(3);
        values.add(2);
        values.add(4);

        System.out.println("max value:" + Collections.max(values));
        System.out.println("min value:" + Collections.min(values));
        Collections.sort(values);
        System.out.println("sorted");
        values.forEach((Integer val) -> System.out.println(val));
    }
}
```

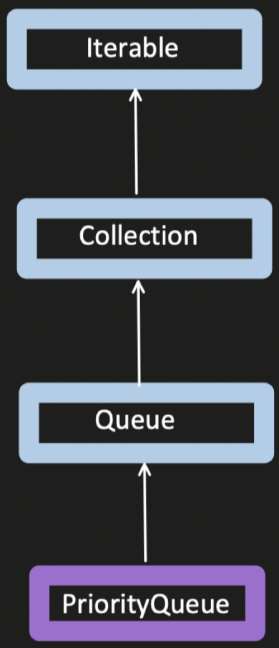
Methods
sort
binarySearch
get
reverse
shuffle
swap
copy
min
max
rotate
unmodifiableCollection

QUEUE:

- QUEUE is an Interface, child of Collection interface.
- Generally QUEUE follows FIFO approach, but there are exceptions like PriorityQueue
- Supports all the methods available in Collection + some other methods mentioned below:

S.No.	Methods Available in Queue Interface	Usage
1.	add()	- Insert the element into the queue. - True if Insertion is Successful and ***Exception*** if Insertion fails - Null element insertion is not allowed will throw (**NPE**)
2.	offer()	- Insert the element into the queue. - True if Insertion is Successful and False if Insertion fails - Null element insertion is not allowed will throw (**NPE**)
3.	poll()	Retrieves and Removes the head of the queue. Returns null if Queue is Empty.
4.	remove()	Retrieves and Removes the head of the queue. Returns ***Exception***(**NoSuchElementException**) if Queue is Empty.
5.	peek()	Retrieves the value present at the head of the queue but do not removes it. Returns null if Queue is Empty.
6.	element()	Retrieves the value present at the head of the queue but do not removes it. Returns an ***Exception*** (**NoSuchElementException**) if Queue is Empty.

PriorityQueue:



- Its of 2 types, **Minimum Priority Queue** and **Maximum Priority Queue**
- It is based on priority Heap (Min Heap and Max Heap).
- Elements are ordered according to either Natural Ordering (by default) or by ***Comparator*** provided during queue construction time.

Min Priority Queue:

```java
public class MinPriorityQueueExample {

    public static void main(String args[]){

        //min priority queue, used to solve problems of min heap.
        PriorityQueue<Integer> minPQ = new PriorityQueue<>();
        minPQ.add(5);
        minPQ.add(2);
        minPQ.add(8);
        minPQ.add(1);

        //lets print all the values
        minPQ.forEach((Integer val) -> System.out.println(val));

        //remove top element from the PQ and print
        while(!minPQ.isEmpty())
        {
            int val = minPQ.poll();
            System.out.println("remove from top:" + val);
        }
    }
}
```

Output:

```
1
2
8
5
remove from top:1
remove from top:2
remove from top:5
remove from top:8
```

Max Priority Queue:

```java
public class MaxPriorityQueue {

    public static void main(String args[]){

        //max priority queue, used to solve problems of max heap
        PriorityQueue<Integer> maxPQ = new PriorityQueue<>((Integer a, Integer b) -> b-a);
        maxPQ.add(5);
        maxPQ.add(2);
        maxPQ.add(8);
        maxPQ.add(1);

        //lets print all the values
        maxPQ.forEach((Integer val) -> System.out.println(val));

        //remove top element from the PQ and print
        while(!maxPQ.isEmpty()){
            int val = maxPQ.poll();
            System.out.println("remove from top:" + val);
        }
    }
}
```

Output:

```
8
2
5
1
remove from top:8
remove from top:5
remove from top:2
remove from top:1
```

Time complexity:

Add and Offer : O(logn)
Peak : O(1)
Poll and Remove head element: O(logn)
Remove arbitrary element : O(n)

Comparator and Comparable both provides a way to sort the collection of objects.

1. Primitive collection sorting

```java
public class Main {
    public static void main(String[] args) {

        int[]array = {1,2,3,4};
        Arrays.sort(array);
    }
}
```

Sorted in Ascending order

```java
public static void sort( @NotNull int[] a) {
    DualPivotQuicksort.sort(a, left: 0, right: a.length - 1, work: null, workBase: 0, workLen: 0);
}
```

2. Object collection sorting

```java
public class Main {
    public static void main(String[] args) {

        Car[] carArray = new Car[3];
        carArray[0] = new Car( name: "SUV", type: "petrol");
        carArray[1] = new Car( name: "Sedan", type: "diesel");
        carArray[2] = new Car( name: "HatchBack", type: "CNG");

        Arrays.sort(carArray);
    }
}
```

```java
public class Car {

    String carName;
    String carType;

    Car(String name, String type){
        this.carName = name;
        this.carType = type;
    }
}
```

```
Exception in thread "main" java.lang.ClassCastException Create breakpoint : Car cannot be cast to java.lang.Comparable
    at java.util.ComparableTimSort.countRunAndMakeAscending(ComparableTimSort.java:320)
    at java.util.ComparableTimSort.sort(ComparableTimSort.java:188)
    at java.util.Arrays.sort(Arrays.java:1246)
    at Main.main(Main.java:11)
```

How to sort the Object Array?

With the help of

Comparator

`int compare(T obj1, T obj2)`

- Sorting algorithm uses this ***compare*** method of Comparator to compare 2 variables and decide whether to swap the variables or not.

- Method returns:
  1: if v1 is bigger than v2
  0: if v1 and v2 is equals
  -1 : if v1 is smaller than v2

Mostly in algorithm, if this method return 1, it swap the values.

Comparable

`int compareTo(T obj2);`

Sorting algorithm uses this ***compareTo*** method of Comparator to compare 2 variables and decide whether to swap the variables or not.

```java
public class Main {
    public static void main(String[] args) {

        Integer a[] = {6,4,1,9,2,11};
        Arrays.sort(a, (Integer val1, Integer val2) -> val1-val2);

        for(int v : a){
            System.out.println(v);
        }
    }
}
```

Output:

```
1
2
4
6
9
11
```

```java
public class Main {
    public static void main(String[] args) {

        Integer a[] = {6,4,1,9,2,11};
        Arrays.sort(a, (Integer val1, Integer val2) -> val2-val1);

        for(int v : a){
            System.out.println(v);
        }
    }
}
```

Output:

```
11
9
6
4
2
1
```

```java
public class Car {

    String carName;
    String carType;

    Car(String name, String type){
        this.carName = name;
        this.carType = type;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {

        Car[] carArray = new Car[3];
        carArray[0] = new Car( name: "SUV", type: "petrol");
        carArray[1] = new Car( name: "Sedan", type: "diesel");
        carArray[2] = new Car( name: "HatchBack" , type: "cng");

        Arrays.sort(carArray, (Car obj1, Car obj2) -> obj2.carType.compareTo(obj1.carType));

        for(Car car : carArray){
            System.out.println(car.carName + ".." + car.carType);
        }
    }
}
```

Output:

```
SUV..petrol
Sedan..diesel
HatchBack..cng
```

```java
public class Main {
    public static void main(String[] args) {

        Car[] carArray = new Car[3];
        carArray[0] = new Car( name: "suv",  type: "petrol");
        carArray[1] = new Car( name: "sedan",  type: "diesel");
        carArray[2] = new Car( name: "hatchback" ,  type: "cng");

        Arrays.sort(carArray, (Car obj1, Car obj2) -> obj1.carName.compareTo(obj2.carName));

        for(Car car : carArray){
            System.out.println(car.carName + ".." + car.carType);
        }
    }
}
```

Output:

```
hatchback..cng
sedan..diesel
suv..petrol
```

```java
public class Main {
    public static void main(String[] args) {

        List<Car> cars = new ArrayList<>();
        cars.add(new Car( name: "suv",  type: "petrol"));
        cars.add(new Car( name: "sedan",  type: "diesel"));
        cars.add(new Car( name: "hatchback",  type: "cng"));

        Collections.sort(cars, (Car ob1, Car ob2) -> ob2.carName.compareTo(ob1.carName));

        cars.forEach((Car carObj) -> System.out.println(carObj.carName + ".." + carObj.carType));

    }
}
```

Output:

```
suv..petrol
sedan..diesel
hatchback..cng
```

```java
public class Car {

    String carName;
    String carType;

    Car(String name, String type){
        this.carName = name;
        this.carType = type;
    }
}
```

```java
public class CarNameComparator implements Comparator<Car> {

    @Override
    public int compare(Car o1, Car o2) {
        return o2.carName.compareTo(o1.carName);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {

        List<Car> cars = new ArrayList<>();
        cars.add(new Car( name: "suv",  type: "petrol"));
        cars.add(new Car( name: "sedan",  type: "diesel"));
        cars.add(new Car( name: "hatchback",  type: "cng"));

        Collections.sort(cars, new CarNameComparator());

        cars.forEach((Car carObj) -> System.out.println(carObj.carName + ".." + carObj.carType));

    }
}
```

Output:

```
suv..petrol
sedan..diesel
hatchback..cng
```

```java
public class Car implements Comparator<Car> {

    String carName;
    String carType;

    Car(){

    }
    Car(String name, String type){
        this.carName = name;
        this.carType = type;
    }

    @Override
    public int compare(Car o1, Car o2) {
        return o1.carName.compareTo(o2.carName);
    }
}
```

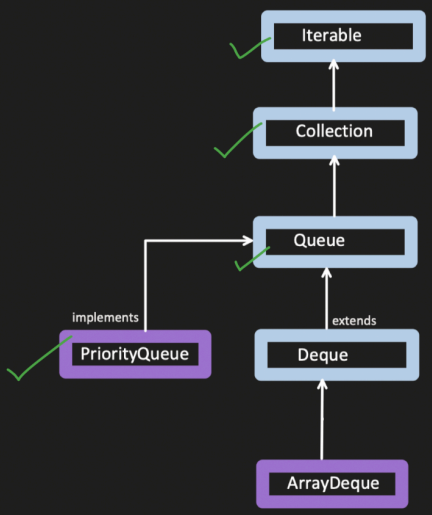
```java
public class Main {
    public static void main(String[] args) {

        List<Car> cars = new ArrayList<>();
        cars.add(new Car( name: "suv",  type: "petrol"));
        cars.add(new Car( name: "sedan",  type: "diesel"));
        cars.add(new Car( name: "hatchback",  type: "cng"));

        Collections.sort(cars, new Car());

        cars.forEach((Car carObj) -> System.out.println(carObj.carName + ".." + carObj.carType));

    }
}
```

Output:

```
hatchback..cng
sedan..diesel
suv..petrol
```

Deque:

Stands for Double Ended Queue. Means addition and removal can be done from both the sides of the queue.



Methods available in Deque Interface:

Methods available in Collection, Queue Interface + Methods available in Deque interface

Queue:
add(), offer(), poll(), remove(), peek(), element()

Deque:

	Throw Exception	Do Not Throw Exception	Throw Exception	Do Not Throw Exception
Insert Operations	addFirst(e)	offerFirst(e) *return true/false*	addLast(e)	offerLast(e) *return true/false*
Remove Operations	removeFirst()	pollFirst() *return null if deque empty*	removeLast()	pollLast() *return null if deque empty*
Examine Operations	getFirst()	peekFirst() *return null if deque empty*	getLast()	peekLast() *return null if deque empty*

Using this methods, we can even use Deque to implement STACK (LIFO) and QUEUE (FIFO) both.
To use it as Stack, push() and pop() method are available.
push() -> internally calls addFirst()
pop() -> internally calls removeFirst()

ArrayDeque:

ArrayDeque (Concrete class), implements the methods which are available in Queue and Deque Interface.

```java
public class Main {
    public static void main(String[] args) {


        ArrayDeque<Integer> arrayDequeAsQueue = new ArrayDeque<>();
        //Insertion
        arrayDequeAsQueue.addLast( e: 1);
        arrayDequeAsQueue.addLast( e: 5);
        arrayDequeAsQueue.addLast( e: 10);

        //Deletion
         int element = arrayDequeAsQueue.removeFirst();
         System.out.println(element);

         //LIFO (last in first out)
        ArrayDeque<Integer> arrayDequeAsStack = new ArrayDeque<>();
        arrayDequeAsStack.addFirst( e: 1);
        arrayDequeAsStack.addFirst( e: 5);
        arrayDequeAsStack.addFirst( e: 10);

        //Deletion
        int removedElem = arrayDequeAsStack.removeFirst();
        System.out.println(removedElem);
    }
}
```

add() -> internally calls addLast() method
offer() -> calles offerLast() method
 poll() -> calls pollFirst() method
 remove() -> calls removeFirst() method
peek() -> calls peekFirst() method
element() -> calls getFirst() method

Time Complexity:

- Insertion: Amortized(Most of the time or Average) complexity is O(1) except few cases like
  O(n) : when queue size threshold reached and try to insert an element at end or front, then its O(n) as values are copied to new queue with bigger size.
- Deletion: O(1)
- Search: O(1)

Space Complexity:
O(n)

Collection Till Now	isThreadSafe	Maintains Insertion Order	Null Elements allowed	Duplicate elements allowed	Thread safeVersion

Priority Queue	No	NO	NO	Yes	PriorityBlockingQueue
					`public class Main {`
					`  public static void main(String[] args) {`
					`  PriorityBlockingQueue<Integer> pq = new PriorityBlockingQueue<>();`
					`  pq.add(5);`
					`  pq.add(2);`
					`  System.out.println(pq.peek());`
					`  }`
					`}`

ArrayDeque	No	YES	NO	Yes	ConcurrentLinkedDeque
					`public class Main {`
					`  public static void main(String[] args) {`
					`  ConcurrentLinkedDeque<Integer> ob = new ConcurrentLinkedDeque<>();`
					`  ob.addFirst( e: 2);`
					`  ob.addLast( e: 1);`
					`  System.out.println(ob.removeLast());`
					`  }`
					`}`

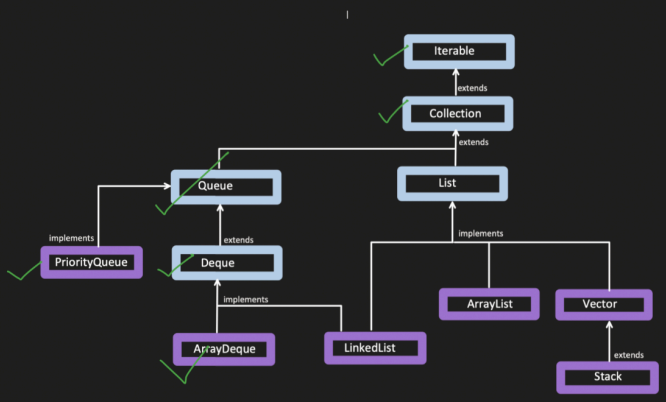
List:

List is a ordered collection of an objects. In which duplicate values can be stored.

How does it differs from Queue?

- Queue is also a collection of an objects but in queue insertion/removal/access can only happen either at start or end of the queue. While in List, data can be inserted, removed or access from anywhere.

- In List, user can decide where to insert or access using index (starting from 0)



Methods available in List Interface:

Methods available in Collection Interface + new Methods defined in List Interface

Collection Interface Methods as explained previously:

S.NO	METHODS	USAGE
1.	size()	It returns the total number of elements present in the collection.
2.	isEmpty()	Used to check if collection is empty or has some value. It return true/false.
3.	contains()	Used to search an element in the collection, returns true/false.
4.	toArray()	It convert collection into an Array.
5.	add(E element)	insert element at end of the collection
6.	remove(E element)	remove the first occurence of the element from the list
7.	addAll(Collection c)	Used to insert all the element of the specified collection in the end of the list.
8.	removeAll(Collection c)	Remove all the elements from the collections, which are present in the collection passed in the parameter.
9.	clear()	Remove all the elements from the collection.
10.	equals()	Used to check if 2 collections are equal or not.
11.	stream() and parallelStream()	Provide effective way to work with collection like filtering, processing data etc.
12.	iterator()	As Iterable interface added in java 1.5, so before this, this method was used to iterate the collection and still can be used.

New Methods defined in the List Interface:

S.No.	Methods Available in Queue Interface	Usage
1.	add(int index, E element)	- Insert the element at the specific position in the list. if there is any element present at that position, it shift it to its Right
2.	addAll(int index, Collection c)	Insert all the elements of the specified collection into this list at specific index, and shift the element at this index and subsequent element to the right.
3.	replaceAll(UniaryOperator op)	relaces each element of the list, with the resulf of appliying the operator the element.
2.	sort(Comparator c)	sort based on the order provided by the comparator.
3.	get(int index)	Return the element at the specified position in the list.
4.	set(int index, Element e)	Replace the element at the specified index in the list with the element provided.
5.	remove(int index)	Remove the element from the index and shift subsequent elements to the left.
6.	indexOf(Object o)	Returns the index of the first occurrence of the specified element in the list. -1 if list does not contains the element.
7.	lastIndexOf(Object o)	Returns the index of the lastoccurrence of the specified element in the list. -1 if list does not contains the element.
8.	ListIterator<E> listIterator()	listIterator return the object of ListIterator. ListIterator (interface) extends from the Iterator (interface) It has all the methods which are avaiable in **Iterator** and helps to iterate in forward direction like: Method Name | Usage hasNext() | Returns true, if there are more elements in collection next() | Returns the next element in the iteration remove() | Removes the last element returned by iterator + New methods which are intorduced in ListIterator, which help to iterate in backward direction: Method Name | Usage boolean hasPrevious() | Returns true, if there are more elements in list while traversing in reverse order. Else throw exception E previous() | Returns the previous element and move the cursor position backward. int nextIndex() | Returns the index of the next element. (return the size of the list, if its at the end of the list) int previousIndex() | Returns the index of the previous element. (return -1 of the list, if its at the begining of the list) set(E e) | Replaces the last element returned by next or previous with the specified element. add(E e) | Insert the specified element immediately before the element that would be returned by the next and after the element that would be returned by the previous. Subsequent call to next() would be uaffected.
9.	ListIterator<E> listIterator(int index)	start the iterator from the specific index. specified index indicates the first element that would be returned by an initial call to next.
10.	List<E> subList(int fromIndex, int toIndex)	return the portion of the list. fromIndex - Inclusive toIndex - Exclusive if fromIndex == toIndex, return sublist is empty. Any changes in sublist, will change in main list also and vise versa.

```java
public class Main {
    public static void main(String[] args) {

        List<Integer> list1 = new ArrayList<>();

        //add(int index, Element e)
        list1.add( index: 0,  element: 100);
        list1.add( index: 1,  element: 200);
        list1.add( index: 2,  element: 300);

        //addAll(int index, Collection c)
        List<Integer> list2 = new ArrayList<>();
        list2.add( index: 0,  element: 400);
        list2.add( index: 1,  element: 500);
        list2.add( index: 2,  element: 600);

        list1.addAll( index: 2, list2);
        list1.forEach((Integer val) -> System.out.println(val));

        //replaceAll(UnaryOperator op)
        list1.replaceAll((Integer val) -> -1 * val);
        System.out.println("after replace all");
        list1.forEach((Integer val) -> System.out.println(val));


        //sort(Comparator c)
        list1.sort((Integer val1, Integer val2) -> val1 - val2);
        System.out.println("after sorting in increasing order");
        list1.forEach((Integer val) -> System.out.println(val));

        //get(int index)
        System.out.println("value present at index 2 is: " + list1.get(2));

        //set(int index, Element e)
        list1.set(2, -4000);
        System.out.println("after set method");

        list1.forEach((Integer val) -> System.out.println(val));

        //remove(int index)
        list1.remove( index: 2);
        System.out.println("after removing");
        list1.forEach((Integer val) -> System.out.println(val));

        //indexOf(Object o)
        System.out.println("index of -200 Integer object is: " + list1.indexOf(-200));


        //need to provide the index in listIterator, from where it has to start.
        ListIterator<Integer> listIterator1 = list1.listIterator(list1.size());

        //traversing backward direction
        while (listIterator1.hasPrevious()){
            int previousVal = listIterator1.previous();
            System.out.println("traversing backward: " + previousVal + " nextIndex: " + listIterator1.nextIndex() + " previous index: " + listIterator1.previousIndex());
            if(previousVal == -100) {
                listIterator1.set(-50);
            }
        }
        list1.forEach((Integer val) -> System.out.println("after set:" + val));

        //traversing forward direction
        ListIterator<Integer> listIterator2 = list1.listIterator();
        while (listIterator2.hasNext()){
            int val = listIterator2.next();
            System.out.println("traversing forward: " + val + " nextIndex: " + listIterator2.nextIndex() + " previous index: " + listIterator2.previousIndex());
            if(val == -200) {
                listIterator2.add(-100);
            }
        }
        list1.forEach((Integer val) -> System.out.println("after add:" + val));

        List<Integer> subList = list1.subList(1, 4);
        subList.forEach((Integer val) -> System.out.println("subList: " + val));

        subList.add(-900);
        list1.forEach((Integer val) -> System.out.println("after value added in sublist: " + val));
    }
}
```

```
Output:
100
200
400
500
600
300
after replace all
-100
-200
-400
-500
-600
-300
after sorting in increasing order
-600
-500
-400
-300
-200
-100
value present at index 2 is: -400
after set method
-600
-500
-4000
-300
-200
-100
after removing
-600
-500
-300
-200
-100
index of -200 Integer object is: 3
traversing backward: -100 nextIndex: 4 previous index: 3
traversing backward: -200 nextIndex: 3 previous index: 2
traversing backward: -300 nextIndex: 2 previous index: 1
traversing backward: -500 nextIndex: 1 previous index: 0
traversing backward: -600 nextIndex: 0 previous index: -1
after set:-600
after set:-500
after set:-300
after set:-200
after set:-50
traversing forward: -600 nextIndex: 1 previous index: 0
traversing forward: -500 nextIndex: 2 previous index: 1
traversing forward: -300 nextIndex: 3 previous index: 2
traversing forward: -200 nextIndex: 4 previous index: 3
traversing forward: -50 nextIndex: 6 previous index: 5
after add:-600
after add:-500
after add:-300
after add:-200
after add:-100
after add:-50
sublist: -500
sublist: -300
sublist: -200
after value added in sublist: -600
after value added in sublist: -500
after value added in sublist: -300
after value added in sublist: -200
after value added in sublist: -900
after value added in sublist: -100
after value added in sublist: -50
```

Time Complexity:

- Insertion:
  O(1) : when inserting the element at the end of the array. And space is sufficient
  O(n) : when inserting the lement at the particular index of the array, then it require shifting of the values,
  O(n) : when array size threshold reached and try to insert an element at end, then also its O(n) as values are copied to new array with bigger array size.

- Deletion: O(n), we have to shift all the values

- Search: O(1)

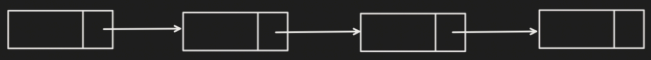
Space Complexity:
O(n)

Collection Till Now	isThreadSafe	Maintains Insertion Order	Null Elements allowed	Duplicate elements allowed	Thread safeVersion

ArrayList	No	Yes	Yes	Yes	CopyOnWriteArrayList
					`public class Test {`
					`public static void main(String args[]){`
					`List<Integer> list = new CopyOnWriteArrayList<>();`
					`list.add( index: 0,  element: 100);`
					`System.out.println(list.get(0));`
					`}`
					`}`

LinkedList: Data structure used is Linked List

- **LinkedList implements both Deque and List interface.**
- Means it support Dequeue methods like : "getFirst", "getLast", "removeFirst", "removeLast" etc...
  +
  It also support index based operations like List : "get(index)" , "add(index, object)" etc.



```java
public class LinkedListExample {

    public static void main(String args[]){

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

        //using deque functionality
        list.addLast( e: 200);
        list.addLast( e: 300);
        list.addLast( e: 400);
        list.addFirst( e: 100);
        System.out.println(list.getFirst());

        //using list functionality
        LinkedList<Integer> list2 = new LinkedList<>();
        list2.add( index: 0,  element: 100);
        list2.add( index: 1,  element: 300);
        list2.add( index: 2,  element: 400);
        list2.add( index: 1,  element: 200);
        System.out.println(list2.get(1) + " and " + list2.get(2));
    }
}
```

Output:

```
100
200 and 300
```

Time Complexity:

- Insertion at start and end : O(1)
- Insertion at particular index : O(n) for lookup of the index + O(1) for adding
- Search: O(n)
- Deletion at start or end: O(1)
- Deletion at specific index: O(n) for the lookup of the index + O(1) for removal

Space Complexity:
O(n)

Collection Till Now	isThreadSafe	Maintains Insertion Order	Null Elements allowed	Duplicate elements allowed	Thread safeVersion
LinkedList	No	Yes	Yes	Yes	CopyOnWriteArrayList `public class Test { public static void main(String args[]){ List<Integer> list = new CopyOnWriteArrayList<>(); list.add( index: 0,  element: 100); System.out.println(list.get(0)); } }`

Vector:

- **Exactly same as arrayList, elements can be access via index.**
- **But its Thread safe**
- Puts lock when operation is performed on vector.
- Less efficient then ArrayList as for each operation it do lock/unlock internally.

```java
public class VectorExample {

    public static void main(String args[]){

        Vector<Integer> obj = new Vector<>();
        obj.add( index: 0,  element: 200);

        System.out.println(obj.get(0));
    }
}
```

Collection Till Now	isThreadSafe	Maintains Insertion Order	Null Elements allowed	Duplicate elements allowed	Thread safeVersion
Vector	Yes	Yes	Yes	Yes	N/A

Stack:

- **Represent LIFO (Last in First out) operation**
- **Since it extends Vector, its method is also Synchronized.**

How its different from Deque : Deque is not thread safe, stack is.

```java
public class StackExample {

    public static void main(String args[]){

        Stack<Integer> stack = new Stack<>();
        stack.push( item: 1);
        stack.push( item: 2);

        System.out.println(stack.pop());

    }
}
```

Time Complexity:

- Insertion : O(1)
- Deletion : O(1)
- Search: O(n)

Space Complexity:
O(n)

Collection Till Now	isThreadSafe	Maintains Insertion Order	Null Elements allowed	Duplicate elements allowed	Thread safeVersion
Stack	Yes	NO	Yes	Yes	N/A