Java Collections

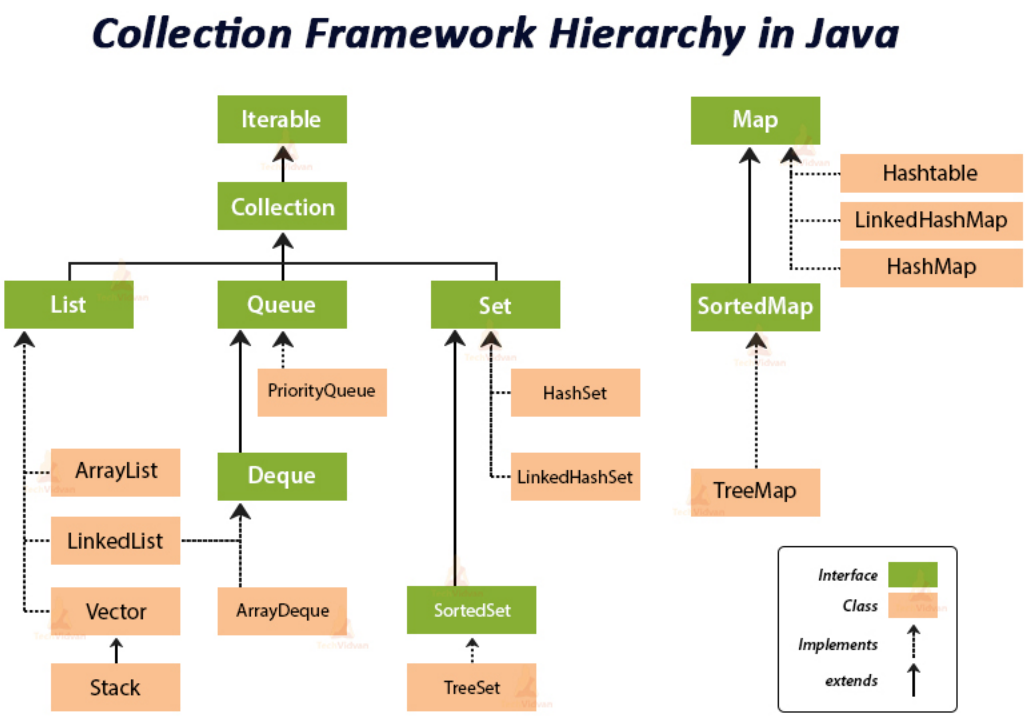
Created	@February 5, 2022 4:07 PM
Reviewed	☐

Table of contents

Table of contents
Collection Framework
List
Queue
Set
Map

Java Collections

▼ Collection Framework

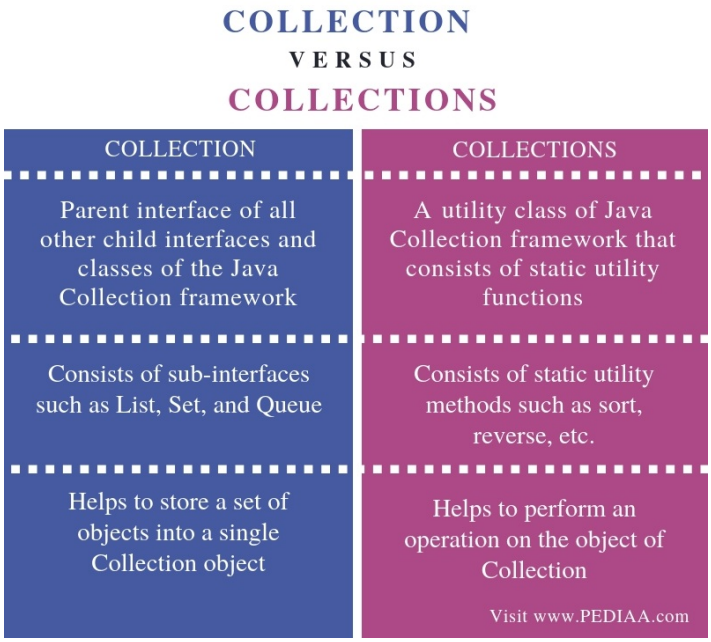



```
interface List extends Collection
// List is an interface which is implemented by ArrayList class,
// LinkedList class and vector class.

class ArrayList implements List
```

The keyword **extends** is used when a class wants to inherit all the properties from another class or an interface that wants to inherit an interface.	We use the **implements** keyword when we want a class to implement an interface.

https://www.youtube.com/watch?v=rzA7UJ-hQn4



Collections class consists exclusively of static methods that operate on or return collections.

sort(*list_only****), sort(****list_only, comparator****), max(****collection****), min(****collection****), reverseOrder(), frequency(****collection, object****)*** are some of its static methods.

> **Methods inherited from class java.lang.Object:**
> *clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait*

▼ List

The Java List interface, java.util.List, represents an ordered sequence of objects. The elements contained in a Java List can be inserted, accessed, iterated and removed **according to the order** in which they appear internally in the Java List. The ordering of the elements is why this data structure is called a List.

You **can add any Java object to a List.** If the List is not typed, using Java Generics, then you can even mix objects of different types (classes) in the same List. Mixing objects of different types in the same List is **not often done in practice**, however.

```
List<Integer> values = Arrays.asList(212, 324, 435, 566,  121);
```

▼ How iterator works understand using LinkedList?

```
Consider the iterator pointer at null when you initialise.
             null  1   2   3
              ^
Do listIt.next:     1   2   3   {It moves pointer forward then returns 1}
                    ^
Do listIt.next:     1   2   3   {It moves pointer forward then returns 2}
                        ^
Do listIt.previous: 1   2   3   {First it returns 2 then moves backward}
                    ^
It.previous:  null  1   2   3   {First it returns 1 then moves backward}
              ^
```

```
LinkedList<Integer> list = new LinkedList<>();
list.add(1);
list.add(2);
list.add(3);        //list = [1,2,3]
```

```
ListIterator listIt = list.listIterator();
System.out.println(listIt.next());      //1
System.out.println(listIt.next());      //2
System.out.println(listIt.previous());  //2
System.out.println(listIt.previous());  //1
```

▼ ArrayList

Need of arraylist: Need to know the size of array from the beginning.

Imp:

Function	Purpose	Time complexity
add	add element at end **or** given index and pushes other elements	O(n) [Due to shifting of other elements in list]
addAll	add a list into another list	
remove	remove element from given index **or** a given value	O(n)
get	get element from given index	
clear	empty whole list	
set	replace value at particular index	O(n)
contains	return true or false if value is present	
indexOf	provide index of given value. -1 if not present	
lastIndexOf	provide index of last occurance of a value	
size	number of elements in the list	
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.	

```
java.util.Iterator<Integer> it = list.iterator();   //iterate in forward direction
while(it.hasNext())
{
    sout(it.next())
}

ListIterator listIt = list.listIterator();      //in iterate in both direction
System.out.println(listIt.next());
System.out.println(listIt.previous());
```

An **Iterator** is an interface in Java and we can traverse the elements of a list in a forward direction whereas a **ListIterator is** **an interface that extends the Iterator interface** and we can traverse the elements in both forward and backward directions.

Advantage: Fast

Disadvantage:

1. Number of elements not known from beginning. Waste space.
2. Once, the limit of the size of array is reached we need to create a new one and copy all the object preferences to new array.

   Since ArrayList is a dynamic array and we do not have to specify the size while creating it, the size of the array automatically increases when we dynamically add and remove items. Though the actual library implementation may be more complex, the following is a very basic idea explaining the working of the array when the array becomes full and if we try to add an item:

   - Creates a bigger-sized memory on heap memory = 10 [*bydefault - to know check the ArrayList constructor by pressing Ctrl*] (for example memory of double size).
   - Copies the current memory elements to the new memory that is of n + n/2 + 1 size.
   - New item is added now as there is bigger memory available now.

- Delete the old memory.

For arraylist see program **P3_arraylist.java.**

1. It can contain duplicate elements.
2. It maintains insertion order.
3. It is non synchronized.
4. It allows random access because array works at the index basis.
5. In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

▼ Stack

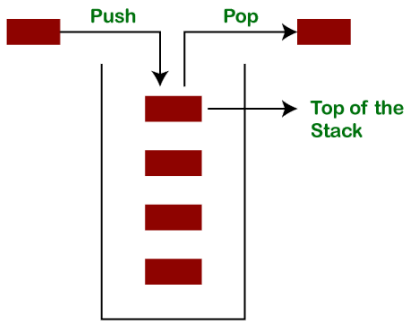
Function	Purpose
push(E item)	add element at the top
peek	view the topmost element
pop	remove the topmost element and return it
empty()	The method checks the stack is empty or not.
search(Object O)	The method searches the specified object and returns the **position of the object from TOP**.

```
/* s1 includes [1,2,3,4] in this form
Index 3 | 4  <- TOP
Index 2 | 3
Index 1 | 2
Index 0 | 1             */

ListIterator<Integer> it = s1.listIterator();
//iterator starts from bottom or can say index 0.

while (it.hasNext())
{
    System.out.println(it.next());
}

/*
s1 = [1, 2, 3, 4]
Search result of 4 is 1
Search result of 3 is 2
Search result of 2 is 3
Index of value 2 is 1
1
2
3
4
-----At the end the pointer would be pointing 4-----
it.previous() = 4
it.previous() = 3
it.next() = 3
*/
```



▼ LinkedList

Function	Purpose
void addFirst(E e)	It is used to insert the given element at the beginning of a list.
void addLast(E e)	It is used to append the given element to the end of a list.
Object clone()	It is used to return a **shallow copy** of an ArrayList.
Iterator<E> descendingIterator()	It is used to return an iterator over the elements in a deque in reverse sequential order. Here, if we do .next to get the previous element.
ListIterator<E> listIterator(int index)	It is used to return a list-iterator of the elements in proper sequence, starting at the specified position in the list.
remove()	**Does removeFirst() internally.** (Even though it is a linkedlist)

peekFirst, peekLast, pollFirst, pollLast, removeFirst, removeLast is also present.

Deep, Shallow and Lazy Copy with Java Examples - GeeksforGeeks

In object-oriented programming, object copying is creating a copy of an existing object, the resulting object is called an object copy or simply copy of the original object.There are several ways to copy an object, most commonly by a copy constructor or cloning.We can define Cloning as "create a copy of object".

https://www.geeksforgeeks.org/deep-shallow-lazy-copy-java-examples/

Advantage: Easily expandable. Easily add elements in between.

1. It can contain duplicate elements.
2. It maintains insertion order.
3. It is non synchronized.
4. Manipulation is fast because no shifting needs to occur.
5. It can be used as a list, stack or queue.

Disadvantage:
Slow to access elements in order to find an element. Everytime need to start from beginning and then iterate further.

How LinkedList works internally?

Since a LinkedList acts as a dynamic array and we do not have to specify the size while creating it, the size of the list automatically increases when we dynamically add and remove items. And also, the elements are not stored in a continuous fashion. Therefore, there is no need to increase the size.

Internally, the LinkedList is implemented using the **doubly linked list data structure**. The main difference between a normal linked list and a doubly LinkedList is that a **doubly linked list contains an extra pointer**, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

▼ ArrayList vs LinkedList

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

▼ Queue

▼ Queue interface

The ONLY functions provided by Queue interface:

Function	Purpose
offer(E)	add element at the rear side. Returns true or false. Never throws an error.
add(E)	It is used to insert the specified element into this deque and return true upon success. It **throws error if not added.**
poll (not pop)	remove element from front and returns it. **Return null if empty**
remove	same as poll but throws exception if queue is empty
peek	only view element from front. **Return null if empty**

> Apart from this there are **methods from Collections** interface:
> *addAll, clear, contains, containsAll, equals, hashCode, isEmpty, iterator, parallelStream, remove, r*
>
> *forEach* from **Iterable** interface.

Iterating the Queue

There are multiple ways to iterate through the Queue. The most famous way is converting the queue to the array and traversing using the for loop. However, the queue also has an inbuilt iterator which can be used to iterate through the queue.

```
Queue<String> pq = new PriorityQueue<>();
Iterator iterator = pq.iterator();
```

Here, as it queue listIterator is not available. Normal iterator is only present.

▼ Linked List

Since *Queue* is an **interface**, objects cannot be created of the type queue. We always need a class which extends this list in order to create an object. A queue can be defined as:

```
// Obj i.e. Integer is the type of the object to be stored in Queue
Queue<Integer> queue = new LinkedList<> ();
```

Apart from the functions present in table in Queue interface there are **NO MORE FUNCTIONS** available which you were able to find in LinkedList when implemented using the List like *descendingIterator, addFirst, addLast,.....*

▼ PriorityQueue

A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a **Queue** follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that's when the PriorityQueue comes into play. The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

Same functions as in LinkedList which is implemented using queue.

Only *difference* is that it takes a **comparator as an argument**. If not passed it uses min heap in background and the smallest element is at index 0.

```
Queue<Integer> pq1 = new PriorityQueue<>(Comparator.reverseOrder());
//Comparator.reverseOrder() returns a comparator
```

While in this case, the **max element will be at index 0**. So, **on polling you get that element and again the max element in the left out PriorityQueue is set to index 0**.

Hence, **order is not maintained.**

▼ Deque

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

offer is same as *offerLast*.

Stack Method	Equivalent Deque Method
`push(e)`	`addLast(e)`
`pop()`	`removeLast()`
`peek()`	`peekLast()`

```
Deque<Integer> dq = new LinkedList<>();
Deque<Integer> dq1 = new ArrayDeque<>();
```

Key differences:

1. The *ArrayDeque* class is the **resizable array implementation of the *Deque* interface** and *LinkedList* class is the list implementation.
2. NULL elements can be added to *LinkedList* but not in *ArrayDeque*
3. ***ArrayDeque* is more efficient than the *LinkedList* for add and remove operation** at both ends and LinkedList implementation is efficient for removing the current element during the iteration
4. The *LinkedList* implementation consumes more memory than the *ArrayDeque*

▼ ArrayDeque

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

ArrayDeque implements Deque

The important points about ArrayDeque class are:

- Unlike Queue, we can add or remove elements from both sides.
- **Null** elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

▼ Set

▼ Set Interface

- Order not maintained
- No duplicate value can be present
- Allows null value

Function	Purpose
add(E e)	Adds the specified element to this set if it is not already present (optional operation). Returns **true** if value added or **false** if value not added.
clear()	Removes all of the elements from this set (optional operation).
contains(Object o)	Returns true if this set contains the specified element.
remove(Object o)	Removes the specified element from this set if it is present (optional operation).
size()	Returns the number of elements in this set (its cardinality).
isEmpty()	True if the set is empty

▼ HashSet

Same methods as above. Child class *HashSet* does not have new methods it just overrides few methods of set. Only clone method (returns shallow copy) is added which isn't there in the set.

```
Set<Integer> set = new HashSet<>();
set.clone();       //gives error
HashSet<Integer> set1 = new HashSet<>();
set1.clone();       //this doesn't give error
```

Operations in HashSet takes place in O(1) as it is optimsed while in TreeSet it takes place in O(log(n)) as it is sorted.

▼ LinkedHashSet

Hash table and linked list implementation of the Set interface, with predictable iteration order. This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the **order in which elements were inserted into the set** (*insertion-order*). Note that insertion order is *not* affected if an element is *re-inserted* into the set. (An element e is reinserted into a set s if s.add(e) is invoked when s.contains(e) would return true immediately prior to the invocation.)

It has no new methods defined at all. All methods inherited from the **parent class set** and others.

▼ TreeSet

Operations in TreeSet takes place in O(log(n)) as it is sorted.

Function	Purpose
first()	It returns the first (lowest) element currently in this sorted set.
last()	It returns the last (highest) element currently in this sorted set.
E ceiling(E e) similarly floor	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
Iterator descendingIterator()	It is used iterate the elements in descending order.
NavigableSet descendingSet()	It returns the elements in reverse order.
headSet(E toElement)	It returns the group of elements that are less than the specified element.
E pollFirst()	It is used to retrieve and remove the lowest(first) element.

Function	Purpose
E pollLast()	It is used to retrieve and remove the highest(last) element.

▼ Map

▼ HashMap

Stores value in key, value form. Replaces the value if same key is referred.

```
package com.java.collections;

import java.util.HashMap;

import java.util.Map;

public class LearnHashMap
{
    public static void main(String[] args)
    {
        try
        {
            Map<Integer, String> str = new HashMap<>();

            str.put(1, "One");

            str.put(2, "Too");

            str.put(3, "Three");

            System.out.println(str);

            str.put(2,"Two");           //values updated

            System.out.println("str.put(2,\"Two\") -> "+str);

            if(!str.containsKey(3))         //similarly there is containsValue("One")
            {
                str.put(3, "Third");        //will not be replaced
            }

            System.out.println("containsKey(3) -> "+str);

            str.putIfAbsent(1, "ONE");

            System.out.println("str.putIfAbsent(1, \"ONE\") -> "+str);

            str.putIfAbsent(4, "Four");

            System.out.println("str.putIfAbsent(4, \"Four\") -> "+str);     //checks key

            System.out.println("\nPrinting sets");

            for(Map.Entry<Integer, String> e: str.entrySet())   //returns a set of user defined format
            {
                System.out.println(e);

                System.out.println(e.getValue());

                System.out.println(e.getKey());
            }

            System.out.println("\nPrinting keys");

            for (Integer key: str.keySet())
            {
                System.out.println(key);
            }

        }
        catch (Exception ex)
        {
            ex.printStackTrace();
```

```
        }
    }
}
//OUTPUT
{1=One, 2=Too, 3=Three}
str.put(2,"Two") -> {1=One, 2=Two, 3=Three}
containsKey(3) -> {1=One, 2=Two, 3=Three}
str.putIfAbsent(1, "ONE") -> {1=One, 2=Two, 3=Three}
str.putIfAbsent(4, "Four") -> {1=One, 2=Two, 3=Three, 4=Four}

Printing sets
1=One
One
1
2=Two
Two
2
3=Three
Three
3
4=Four
Four
4

Printing keys
1
2
3
4
```

▼ TreeMap

It sorts based on the key.

```
package com.java.collections;

import java.util.Map;

import java.util.SortedMap;

import java.util.TreeMap;

public class LearnTreeMap
{
    public static void main(String[] args)
    {
        try
        {
            //here operations take place in O(log(n)) time.
            Map<Integer, String> map = new TreeMap<>();
//            SortedMap<Integer, String> map = new TreeMap<>();     //TreeMap implements SortedMap

            map.put(2, "two");

            map.put(1,"one");

            map.put(3, "three");       //present in sorted order

            System.out.println(map);

            //same operations as we did in HASHMAP

            map.put(2, "four");     //value at the key 2 would be updated.

            System.out.println(map);

            if(!map.containsKey(3))        //it contains key so won't change value of key 3
            {
                map.put(3,"five");
            }

            System.out.println(map);

            map.putIfAbsent(3, "six");      //it is present so won't overwrite

            System.out.println(map+"\n");
```

```
            //ITERATION
            for (Map.Entry<Integer, String> m: map.entrySet())
            {
                System.out.print(m+"\t");

                System.out.println(m.getKey() +" "+m.getValue());
            }

            System.out.println();

            for(Integer i: map.keySet())
            {
                System.out.print(i+", ");
            }

            System.out.println();

            for(String s: map.values())
            {
                System.out.print(s+", ");
            }

            System.out.println("\n"+map.isEmpty());

            System.out.println(map.containsKey(12));

            System.out.println(map.containsValue("one"));

            map.remove("four");         // in remove a key is passed and the set is deleted
                                            // even key, value can be passed

            System.out.println(map);


        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }

    }
}
//OUTPUT
{1=one, 2=two, 3=three}
{1=one, 2=four, 3=three}
{1=one, 2=four, 3=three}
{1=one, 2=four, 3=three}

1=one 1 one
2=four  2 four
3=three 3 three

1, 2, 3,
one, four, three,
false
false
true
```

▼ LinkedHashMap

Maintains sequence of adding. If added on same key then value is replaced.

Uses internally:

Data structure	Uses
HashSet	HashMap
HashMap	hashcode. Array and linkedlist
ArrayList	array of Object class. **Object[] Array.**
LinkedList	Doubly Linked List
Stack	array