07

COLLECTION FRAMEWROK

OBJECT ORIENTED PROGRAMMING LAB

SESSION - 07

CSE215 | DAFFODIL INTERANTIONAL UNIVERSITY

What is Collection framework

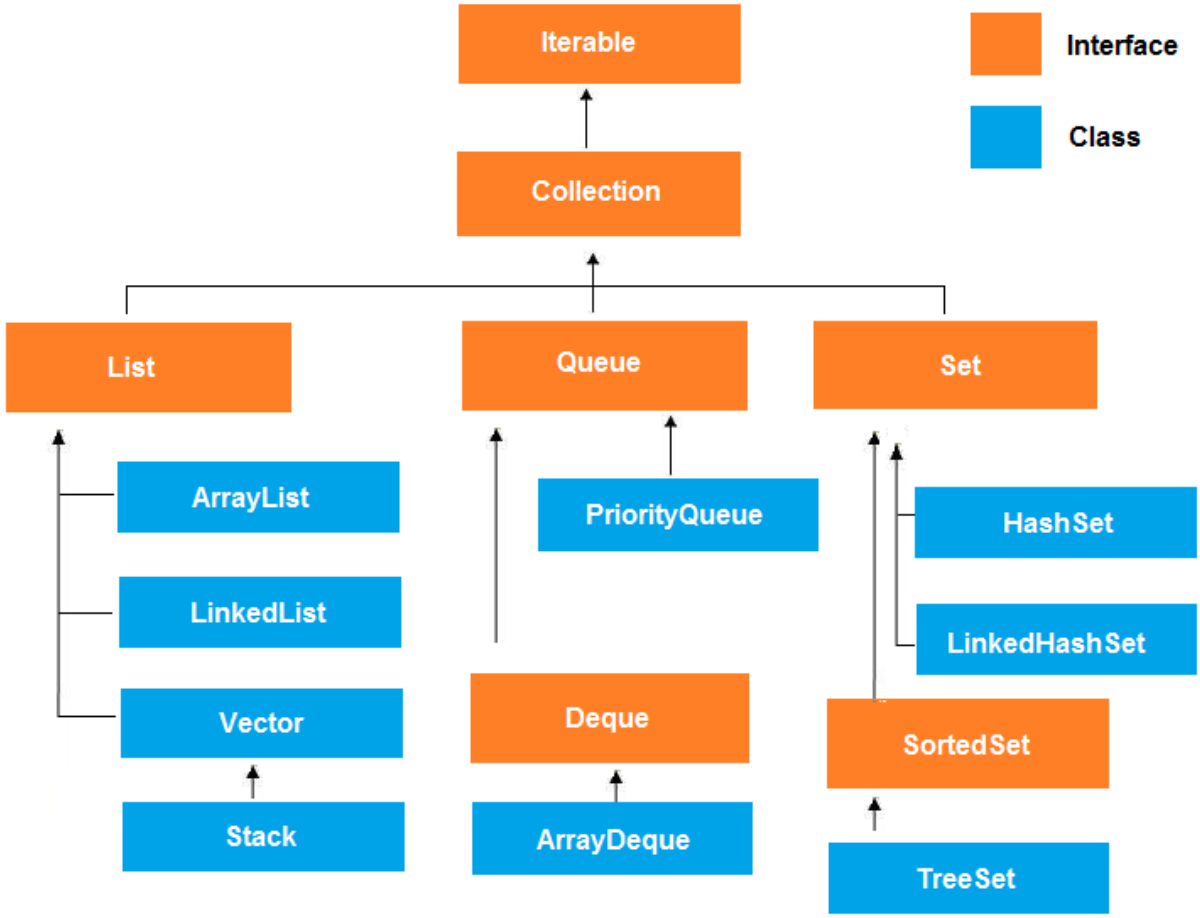
Collections in java is a framework that provides an architecture to store and manipulate the group of objects. It has:

1. Interfaces and its implementations i.e. classes
2. Algorithm

What is framework in java

- provides readymade architecture.
- represents set of classes and interface.
- is optional.

Hierarchy of Collection Framework



Collection Interface, Class & Class Hierarchy:

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following –

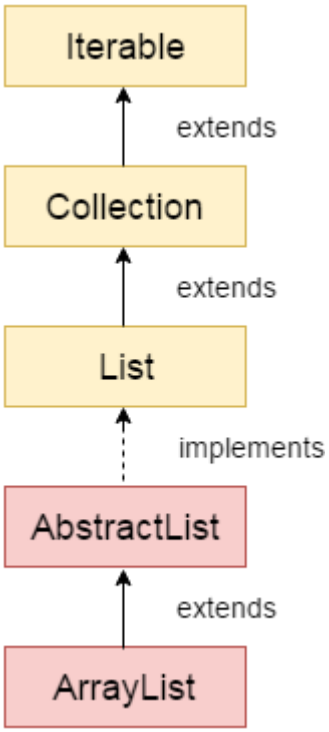
- **Interfaces** – These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations, i.e., Classes** – These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms** – These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.



Java ArrayList Methods.

Sr.No.	Method & Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).
2	**boolean add(Object o)** Appends the specified element to the end of this list.
3	**boolean addAll(Collection c)** Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException, if the specified collection is null.
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null.
5	**void clear()** Removes all of the elements from this list.
6	**int size()** Returns the number of elements in this list.
7	**Object get(int index)** Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
8	**int indexOf(Object o)** Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

Example 7.1: Use of ArrayList methods.

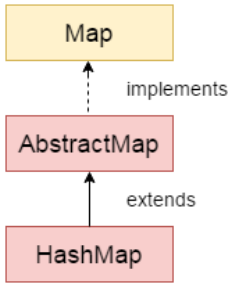
```
import java.util.*;
public class ArrayListDemo {
   public static void main(String args[]) {
      // create an array list
      ArrayList al = new ArrayList();
      System.out.println("Initial size of al: " + al.size());
      // add elements to the array list
      al.add("C");
      al.add("A");
      al.add("E");
      al.add("B");
      al.add("D");
      al.add("F");
      al.add(1, "A2");
      System.out.println("Size of al after additions: " + al.size());
      // display the array list
      System.out.println("Contents of al: " + al);
      // Remove elements from the array list
      al.remove("F");
      al.remove(2);
      System.out.println("Size of al after deletions: " + al.size());
      System.out.println("Contents of al: " + al);
   }
}
```

Output:

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

Java HashMap class

Java HashMap class implements the map interface by using a hashtable. It inherits AbstractMap class and implements Map interface.



The important points about Java HashMap class are:

- A Hash Map contains values based on the key.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It maintains no order.

Java ArrayList Methods.

Sr.No	Method & Description
1	**void clear( )** Resets and empties the hash table.
2	**int size( )** Returns the number of entries in the hash table.
3	**Object get(Object key)** Returns the object that contains the value associated with the key. If the key is not in the hash table, a null object is returned.
4	**Object put(Object key, Object value)** Inserts a key and a value into the hash table. Returns null if the key isn't already in the hash table;.
5	**boolean isEmpty( )** Returns true if the hash table is empty; returns false if it contains at least one key.
6	**Object remove(Object key)** Removes the key and its value. Returns the value associated with the key.

Example 7.2: Use of HashMap methods

```
import java.util.*;
public class HashTableDemo {
   public static void main(String args[]) {
      // Create a hash map
      Hashtable balance = new Hashtable();
      Enumeration names;
      String str;
      double bal;
      balance.put("Zara", new Double(3434.34));
      balance.put("Mahnaz", new Double(123.22));
      balance.put("Ayan", new Double(1378.00));
      balance.put("Daisy", new Double(99.22));
      balance.put("Qadir", new Double(-19.08));
      // Show all balances in hash table.
      names = balance.keys();

      while(names.hasMoreElements()) {
         str = (String) names.nextElement();
         System.out.println(str + ": " + balance.get(str));
      }
      System.out.println();
      // Deposit 1,000 into Zara's account
      bal = ((Double)balance.get("Zara")).doubleValue();
      balance.put("Zara", new Double(bal + 1000));
      System.out.println("Zara's new balance: " + balance.get("Zara"));
   }
}
```

Output

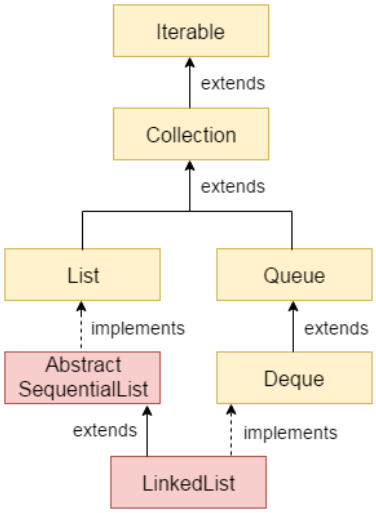
```
Qadir: -19.08
Zara: 3434.34
Mahnaz: 123.22
Daisy: 99.22
Ayan: 1378.0

Zara's new balance: 4434.34
```

Java LinkedList class

Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
- Java LinkedList class can be used as list, stack or queue.



Sr.No.	Method & Description
1	**boolean add(Object o)** Appends the specified element to the end of this list.
2	**boolean addAll(Collection c)** Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException if the specified collection is null.
3	**void addFirst(Object o)** Inserts the given element at the beginning of this list.
4	**void addLast(Object o)** Appends the given element to the end of this list.
5	**void clear()** Removes all of the elements from this list.

6	**Object get(int index)** Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
7	**int indexOf(Object o)** Returns the index in this list of the first occurrence of the specified element, or -1 if the list does not contain this element.
8	**Object remove(int index)** Removes the element at the specified position in this list. Throws NoSuchElementException if this list is empty.
9	**boolean contains(Object o)** Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).
10	**Object getFirst()** Returns the first element in this list. Throws NoSuchElementException if this list is empty.

Example 7.3: Use of LinkedList methods.

```
import java.util.*;
public class LinkedListDemo {
   public static void main(String args[]) {
      // create a linked list
      LinkedList ll = new LinkedList();
      // add elements to the linked list
      ll.add("F");
      ll.add("B");
      ll.add("D");
      ll.add("E");
      ll.add("C");
      ll.addLast("Z");
```

```
        ll.addFirst("A");
        ll.add(1, "A2");
        System.out.println("Original contents of ll: " + ll);
        // remove elements from the linked list
        ll.remove("F");
        ll.remove(2);
        System.out.println("Contents of ll after deletion: " + ll);
        // remove first and last elements
        ll.removeFirst();
        ll.removeLast();
        System.out.println("ll after deleting first and last: " + ll);

        // get and set a value
        Object val = ll.get(2);
        ll.set(2, (String) val + " Changed");
        System.out.println("ll after change: " + ll);
    }
}
```

Output:

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```