Introduction to Collections

In the previous classes we have seen there are different ways to store a data. We have seen the first method called as variable approach, then Arrays.

But each of these had their own limitations. Let us briefly look into what were those limitations.

Variable Approach: In variable approach we had difficulty in creating the references for each variable separately. And not only creating them, in fact accessing them was difficult too because it's definitely not easy to remember all the references. These limitations were overcome by the next approach which is Array approach.

Array Approach: In array approach we had the advantages of storing the same type of data in series one after the other, which made it easy to store the values as well as access. But this approach came with its own limitations which were,

- Only homogeneous can be stored.
- Require contiguous memory allocations in RAM.
- Cannot grow or shirnk in size.
- Do not have built-in methods for data manipulation.

Why Collections???

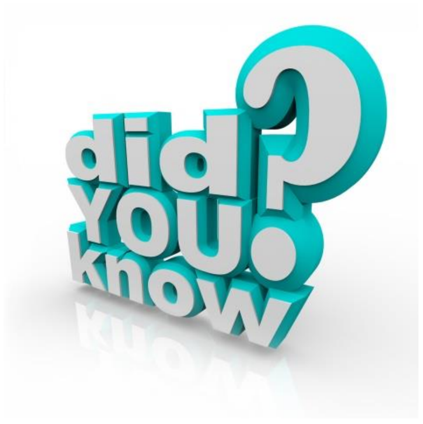
To overcome the limitations that we came across with variable, array type approach we are now going to see what Collections consists of…

The **Collection in Java** is a framework that **provides an architecture to store and manipulate the group of objects.**

Java Collections can achieve all the operations that you perform on a data such as **searching, sorting, insertion, manipulation, and deletion.**

Java Collection means **a single unit of objects**. Java Collection framework provides many interfaces **(Set, List, Queue, Deque)** and classes **(ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).**

What is a framework?



- It provides readymade architecture.
- It represents a set of classes and interfaces.

What is Collection framework?

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

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

ArrayLists

The ArrayList class extends AbstractList and implements the List interface. **ArrayList supports dynamic arrays that can grow as needed**.

Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

Let us see if actually ArrayList can store heterogeneous data with a simple example below

```java
import java.util.ArrayList;
class Demo
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add("ROHIT");
        al.add(true);
        al.add(99.9f);
        System.out.println(al);
    }
}
```

Output:

```
[10, 20, 30, ROHIT, true, 99.9]
Press any key to continue . . .
```

Following is the list of the constructors provided by the ArrayList class.

Sr.No.	Constructor & Description
1	**ArrayList( )** This constructor builds an empty array list.
2	**ArrayList(Collection c)** This constructor builds an array list that is initialized with the elements of the collection **c**.
3	**ArrayList(int capacity)** This constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

Apart from the methods inherited from its parent classes, ArrayList defines the following 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	**Object clone()** Returns a shallow copy of this ArrayList.
7	**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)).
8	**void ensureCapacity(int minCapacity)** Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

9	**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()).
10	**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.
11	**int lastIndexOf(Object o)** Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
12	**Object remove(int index)** Removes the element at the specified position in this list. Throws IndexOutOfBoundsException if the index out is of range (index < 0 || index >= size()).
13	**protected void removeRange(int fromIndex, int toIndex)** Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
14	**Object set(int index, Object element)** Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
15	**int size()** Returns the number of elements in this list.
16	**Object[] toArray()** Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.
17	**Object[] toArray(Object[] a)** Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.
18	**void trimToSize()** Trims the capacity of this ArrayList instance to be the list's current size.