**Collection**

What is collection

A collection is simply an object that represents an group of objects, know as its elements.

Collections is a utility class which provides static methods.

Collection is a parent interface for Set,List,Queue.

Iterable is a parent interface for Collection, List, Set, Queue. By using iterable interface we can use for each loop, so this interface is extend by every interface which is mentioned so we can use for each loop. (Iterable has for each loop method defined)

What is collection framework

It provides a set of interfaces and classes that help in managing group of objects.

Before the introduction of the collection framework in JDK 1.2, java used to rely on a variety of classes like Vector, Stack,HashTable and arrays to store and manipulate groups of objects. However these classes have several drawbacks.

* Inconsistency: Each class had a different way of managing collections, leading to confusion and steep learning curve.
* Lack of inter-operability: These classes were not designed to work together seamlessly.
* No common interface: There was no common interface for all these classes, which meant you couldn’t write generic algorithms that could operate on different types of collections.

To solve this problem, the collection framework was introduced.

* Unified architecture: A consistent set of interfaces for all collections.
* Inter-operability: Collections can be easily interchanged and manipulated in a uniform way.
* Reusability: Generic algorithms can be written that work with any collection.
* Efficiency: The framework provides efficient algorithms for basic operations like searching, sorting and manipulation.

Key Interfaces in the collection framework

The collection framework is primarily built around a set of interfaces.

* Collection: The root interfaces for all the other collection types.
* List: An ordered collection that can contain duplicate elements (eg: Array list, LinkedList)
* Set: A collection that cannot contain duplicate elements (eg: HashSet, Linked HashSet)
* Map: An interface that represents a collection of key value pairs (eg: HashMap, Linked HashMap)

List

The List interface in Java is a part of the java.util package and is a sub-interface of the Collection interface. It provides a way to store an ordered collection of elements (known as a sequence). Lists allow for precise control over where elements are inserted and can contain duplicate elements.

The List interface is implemented by several classes in the Java Collection Framework, such as *ArrayList*, *LinkedList*, *Vector*, and *Stack*.

**Key Features of the List Interface**

* **Order Preservation**
* **Index-Based Access**
* **Allows Duplicates**

**Array List Class**

An ArrayList is a resizable array implementation of the List interface. Unlike arrays in Java, which have a fixed size, an *ArrayList can change its size dynamically* as elements are added or removed. This flexibility makes it a popular choice when the number of elements in a list isn't known in advance.

Unlike a regular array, which has a fixed size, an ArrayList can grow and shrink as elements are added or removed. This dynamic resizing is achieved by creating a new array when the current array is full and copying the elements to the new array.

Internally, the ArrayList is implemented as an array of Object references. When you add elements to an ArrayList, you're essentially storing these elements in this internal array.

When you create an ArrayList, it has an initial capacity (default is 10). The capacity refers to the size of the internal array that can hold elements before needing to resize.

**Adding Elements**

When we add an element to an ArrayList, the following steps occur

Check Capacity: Before adding the new element, ArrayList checks if there is enough space in the internal array (elementData). If the array is full, it needs to be resized.

Resize if Necessary: If the internal array is full, the ArrayList will create a new array with a larger capacity (usually 1.5 times the current capacity) and copy the elements from the old array to the new array.

Add the Element: The new element is then added to the internal array at the appropriate index, and the size is incremented.

**Resizing the Array**

* Initial Capacity: By default, the initial capacity is 10. This means the internal array can hold 10 elements before it needs to grow.
* Growth Factor: When the internal array is full, a new array is created with a size 1.5 times the old array. This growth factor balances memory efficiency and resizing cost.
* Copying Elements: When resizing occurs, all elements from the old array are copied to the new array, which is an O(n) operation, where n is the number of elements in the ArrayList.

**Removing Elements**

* Check Bounds: The ArrayList first checks if the index is within the valid range.
* Remove the Element: The element is removed, and all elements to the right of the removed element are shifted one position to the left to fill the gap.
* Reduce Size: The size is decremented by 1.

**Declaring ArrayList**

* 1. **List<String> list = Arrays.asList(array);**
* **What it does**:  
  Creates a fixed-size list backed by the given array.
* **Key points**:
  + **Changes to the array** are reflected in the list and vice versa.
  + **Size is fixed** → you cannot add or remove elements (add() / remove() will throw UnsupportedOperationException).
  + You can **replace** elements (set(index, value) works).
  1. **List<String> list = new ArrayList<>();**
* **What it does**:  
  Creates a **mutable**, growable, and resizable list.
* **Key points**:
  + Fully independent — not backed by any array.
  + Can freely add(), remove(), set(), clear(), etc.
  + Most commonly used in real-world code for flexibility.
  1. **List<String> list = List.of(); *(Java 9+)***
* **What it does**:  
  Creates an **immutable** list with given elements.
* **Key points**:
  + Size is fixed and elements **cannot be changed**.
  + Both add(), remove() and set() will throw UnsupportedOperationException.
  + Null elements **not allowed** → NullPointerException if you try.

**Key Points**

If you want complete flexibility, go with new ArrayList<>().  
If you want a fixed-size view of an existing array, use Arrays.asList().  
If you want a read-only list, use List.of().

**Different ways to add elements in ArrayList**

// Different ways to add elements  
ArrayList<Integer> list=new ArrayList<>();  
 ArrayList<Integer> list1=new ArrayList<>();  
 list1.add(1);  
 list1.add(2);  
   
 // normal way  
 list.add(23);  
 // adding using index  
 list.add(1,22);  
 // adding all elements once  
 list.addAll(list1);  
 // adding all elements once using index  
 list.addAll(0,list1);

**Accessing element from Array List**

ArrayList<Integer> list1=new ArrayList<>();  
 list1.add(1);  
 list1.add(2);

list1.get(0);

**Modifying element from Array List**

ArrayList<Integer> list1=new ArrayList<>();  
 list1.add(1);  
 list1.add(2);

list1.set(0,22);

**Removing the elements from ArrayList**

ArrayList<Integer> list1=new ArrayList<>();  
 list1.add(1);  
 list1.add(2);

list.remove(1); // removes the value 2 because its using index

list.remove(Integer.valueOf(1)); // here we are wrapping the class its takes as a object and remove the value 1.

**Converting Arraylist into Array**

ArrayList<Integer> list1 = new ArrayList<>();  
list1.add(1);  
list1.add(2);  
/\*  
Converting arraylist into array just need to declare 0 if the arraylist  
has any size  
 \*/  
Integer[] array = list1.toArray(new Integer[0]);  
  
for (int num:array){  
 System.*out*.println(num);  
}

**Sorting**

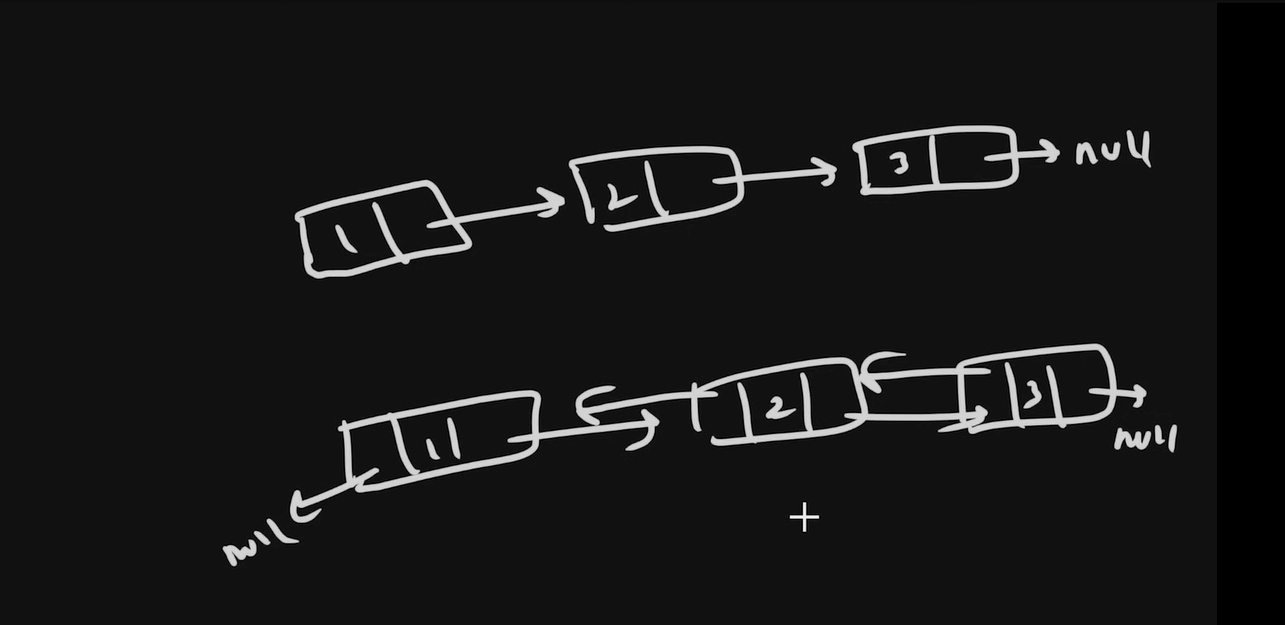
// Sorting  
ArrayList<Integer> list1 = new ArrayList<>();  
list1.add(11);  
list1.add(2);  
// Ascending order  
Collections.*sort*(list1);  
System.*out*.println(list1);  
// descending order  
Collections.*sort*(list1, Collections.*reverseOrder*());  
System.*out*.println(list1);

**Linked List**

The LinkedList class in Java is a part of the Collection framework and implements the List interface. Unlike an ArrayList, which uses a dynamic array to store the elements, a LinkedList stores its elements as nodes in a doubly linked list. This provides different performance characteristics and usage scenarios compared to ArrayList.

A LinkedList is a linear data structure where each element is a separate object called a node. Each node contains two parts:

* Data: The value stored in the node.
* Pointers: Two pointers, one pointing to the next node (next) and the other pointing to the previous node (previous).

****

**The first image is diagram of single linked list or (Linked list) shows how data is strore using pointers. In first block data one is stored and showing the reference of 2 block data, means in first block data will store and address of 2nd data will be there at the end the address is null because there is no other data is there.**

**The second image shows double linked list in that case the first block data is there and address of second block data is there, in second block data is there and first block address is there.**

Performance Considerations

LinkedList has different performance characteristics compared to ArrayList:

* Insertions and Deletions: LinkedList is better for frequent insertions and deletions in the middle of the list because it doesn't require shifting elements, unlike ArrayList.
* Random Access: LinkedList has slower random access (get(int index)) compared to ArrayList because it has to traverse the list from the beginning to reach the desired index.
* Memory Overhead: LinkedList requires more memory than ArrayList because each node in a linked list requires extra memory to store references to the next and previous nodes.

**OOPS Concepts**

Four pillars of oops

Encapsulation

Inheritance

Polymorphism

Abstraction

**4.Abstraction**

**Hiding the implementation details and showing the functionality is known as abstraction.**

Abstract class is declared with the abstract keyword.

It may have both abstract & non-abstract methods (methods with bodies).

An abstract is a Java modifier applicable for class & methods in Java but not for variables.

Java abstract class is a class that cannot be instantiated by itself;

it needs to be subclassed by another class to use its properties.

An abstract class is declared using the "abstract" keyword.

1) An instance of an abstract class cannot be created.

2) Normal constructor is allowed but a constructor with abstract keyword is not allowed.

3) We can have an abstract class without any abstract method.

4) We can have final method but not an abstract final method.

5) We can define static methods in an abstract class.

If a class contains at least one abstract method then compulsory we should declare a class as abstract.

(Methods: abstract, variables)

Methods (public, private, static, final, instance, protected, local, private, final)

🔑 Interview line

“No, we cannot declare variables as abstract in Java because variables do not have implementation — only methods can be abstract.”