


Agenda

1. Java Collection Framework 
2. Collection Interface
3. Interfaces that extends Collection Interface :
4. Map Interface
5. Comparable
6. Comparators

Java Collection Framework

- Any group of individual objects which are represented as a single unit is known as a collection of objects.
- A framework is a set of classes and interfaces which provide a ready-made architecture.

Interface Car {

function start();
function stop();

}

class Car implements Car {

start() {
|
| 3
stop() {
| 3
| 3

}

Interface

usb charging port

Phone →

Laptop →

Camera →

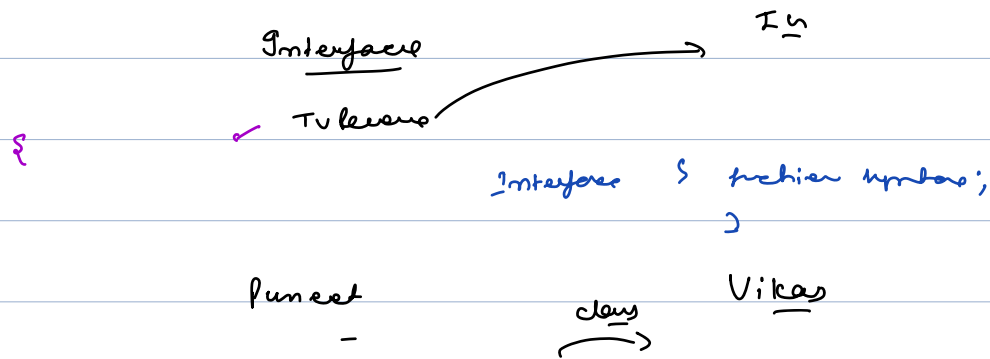
ArrayList → list.add()

Vector → vector.add()

Interface (add
get)

ArrayList

Vector



```

Interface TV Remote {
    volume up();
    volume down();
}

```

Java Collection Framework :-

- **The Java Collections Framework (JCF)** is a set of classes and interfaces that implement commonly reusable collection data structures like **List, Set, Queue, Map, etc.** The JCF is organized into interfaces and implementations of those interfaces. The interfaces define the functionality of the collection data structures, and the implementations provide concrete implementations of those interfaces.

Need of Java Collection Framework

before jdk 1.2 ,

- Before the Collection Framework(or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were **Arrays or Vectors, or Hashtables**.
- All of these collections had no common interface. Therefore, though the main aim of all the collections is the same, the implementation of all these collections was defined independently and had no correlation among them. Hence, it is very difficult for the users to remember all the different methods, syntax, and constructors present in every collection class.

```
public static void main(String[] args)
{
    // Creating instances of the array,
    // vector and hashtable
    int arr[] = new int[] { 1, 2, 3, 4 };
    Vector<Integer> v = new Vector();
    Hashtable<Integer, String> h = new Hashtable();

    // Adding the elements into the
    // vector
    v.addElement(1);
    v.addElement(2);

    // Adding the element into the
    // hashtable
    h.put(1, "geeks");
    h.put(2, "4geeks");
}
```

```
// Array instance creation requires [],
// while Vector and hashtable require ()
// Vector element insertion requires addElement(),
// but hashtable element insertion requires put()

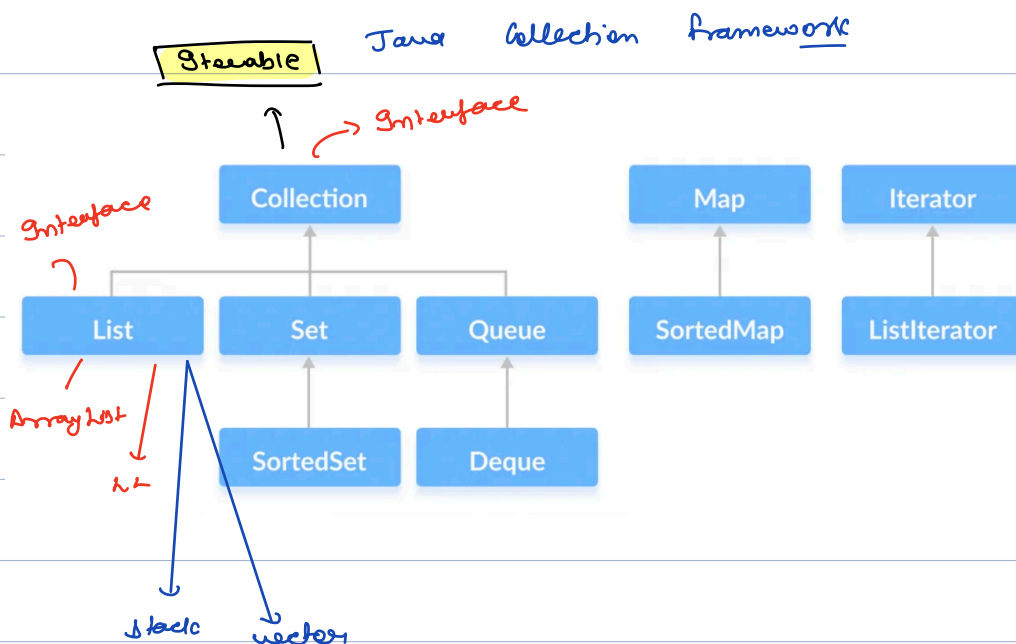
// Accessing the first element of the
// array, vector and hashtable
System.out.println(arr[0]);
System.out.println(v.elementAt(0));
System.out.println(h.get(1));

// Array elements are accessed using [],
// vector elements using elementAt()
// and hashtable elements using get()
}
```

Advantages of JCF :-

- **Consistent API** - The API has a basic set of interfaces like Collection, Set, List, or Map, all the classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.
- **Reduces programming effort** - A programmer doesn't have to worry about the design of the Collection but rather he can focus on its best use in his program. Therefore, the basic concept of Object-oriented programming (i.e.) abstraction has been successfully implemented.
- **Increases program speed and quality** - Increases performance by providing high-performance implementations of useful data structures and algorithms because in this case, the programmer need not think of the best implementation of a specific data structure. He can simply use the best implementation to drastically boost the performance of his algorithm/program.

java.util.* → contains all classes & objects that are categorized into java collection framework.



Collection Interface :-

- one of
- The Collection interface is the root interface of the Java Collections Framework. It is the foundation upon which the collection framework is built. It declares the core methods that all collections will have. The
 - Collection interface is a part of the java.util package.
 - The JDK does not provide any direct implementations of this interface: it provides implementations of more specific sub-interfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.
 - The Collection interface is not directly implemented by any class. However, it is implemented indirectly via its subtypes or subinterfaces like List, Queue, and Set. For Example, the HashSet class implements the Set interface which is a subinterface of the Collection interface.
 - It ~~implements~~ extends the Iterable interface.

extends.

Functions of Collection Interface :-

- **add()** - inserts the specified element to the collection
- **size()** - returns the size of the collection
- **remove()** - removes the specified element from the collection
- **iterator()** - returns an iterator to access elements of the collection
- **addAll()** - adds all the elements of a specified collection to the collection
- **removeAll()** - removes all the elements of the specified collection from the collection
- **clear()** - removes all the elements of the collection

Interface that extends Collection interface

1. **List** : This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects. This also allows duplicate data to be present in it.

Set: A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects.

SortedSet: This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface and is used to handle the data which needs to be sorted.

Queue: As the name suggests, a queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line. This interface is dedicated to storing all the elements where the order of the elements matter. For example, whenever we try to book a ticket, the tickets are sold at the first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket.

Deque: This is a very slight variation of the queue data structure. Deque, also known as a double-ended queue, is a data structure where we can add and remove the elements from both the ends of the queue. This interface extends the queue interface.

List Interface :-

- The **List** interface is a child of **Collection** Interface. The List interface is found in **java.util** package and inherits the Collection interface.
- It is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order, it allows positional access and insertion of elements.
- The implementation classes of the List interface are ArrayList, LinkedList, Vector and Stack. ArrayList and LinkedList are widely used in Java programming.

```
public interface List<E> extends Collection<E> ;
```

1. **ArrayList** - Resizable-array implementation of the List interface
2. **Vector** - Synchronized resizable-array implementation of the List interface
3. **Stack** - Subclass of Vector that implements a standard last-in, first-out stack
4. **LinkedList** - Doubly-linked list implementation of the List and Deque interfaces

1) ArrayList Class

collection



List



ArrayList

- An ArrayList in Java is implemented as a resizable array, also known as a dynamic array. It provides an interface to work with a dynamically sized list of elements, allowing for efficient insertion, deletion, and random access. Here's how an ArrayList is typically implemented:

1. **Backing Array:** The core of an ArrayList is an underlying array that holds the elements. This array is initially created with a default size, and elements are stored sequentially in it.
2. **Resizing:** As elements are added to the ArrayList, the backing array may become full. When this happens, a new larger array is created, and the existing elements are copied from the old array to the new one. This process is called resizing or resizing the array.
3. **Dynamic Sizing:** The resizing operation ensures that the ArrayList can dynamically grow or shrink in size as needed. This dynamic sizing is a key feature that differentiates it from a regular array.
4. **Index-Based Access:** ArrayList allows elements to be accessed by their index. This is achieved by directly accessing the corresponding element in the backing array using the index.
5. **Insertion and Deletion:** When an element is inserted at a specific index or removed from the ArrayList, the other elements may need to be shifted to accommodate the change. This can involve moving multiple elements within the array.
6. **Efficiency:** ArrayList provides efficient constant-time ($O(1)$) access to elements by index. However, insertion or deletion operations at the beginning or middle of the list may require shifting elements and take linear time ($O(n)$), where n is the number of elements.
7. **Automatic Resizing:** Java's ArrayList uses automatic resizing strategies to ensure that the array is appropriately resized when needed. The exact resizing strategy can vary across different implementations and versions of Java.

Break

10:10 pm - 10:20 pm,

```
interface InterfaceA {  
    void methodA();  
}  
  
interface InterfaceB extends InterfaceA {  
    void methodB();  
}
```


Vectors

- A vector provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. This is identical to ArrayList in terms of implementation. However, the primary difference between a vector and an ArrayList is that a Vector is synchronized and an ArrayList is non-synchronized.

```
// Size of the vector
int n = 5;

// Declaring the List with initial size n
List<Integer> v = new Vector<Integer>(n);

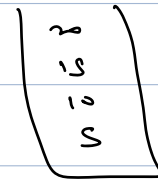
// Appending the new elements
// at the end of the list
for (int i = 1; i <= n; i++)
    v.add(i);

// Printing elements
System.out.println(v);

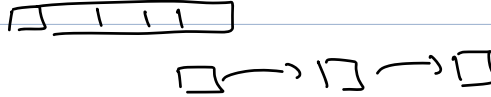
// Remove element at index 3
v.remove(3);
```

Stack

Stack is a class that is implemented in the collection framework and extends the vector class models and implements the Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek.



LinkedList



- LinkedList is a class that is implemented in the collection framework which inherently implements the **linked list data structure**.
- It is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part.
- The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays.

* Set Interface

- The Set interface extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items.
- We can store at most one null value in Set.
- This interface contains the methods inherited from the Collection interface and adds a feature that restricts the insertion of the duplicate elements.

- **HashSet** is one of the widely used classes which implements the Set interface.

* SortedSet Interface :-

- The SortedSet interface present in java.util package extends the Set interface present in the collection framework. It is an interface that implements the mathematical set.
- This interface contains the methods inherited from the Set interface and adds a feature that stores all the elements in this interface to be stored in a sorted manner.

TreeSet class

- TreeSet class which is implemented in the collections framework and implementation of the SortedSet Interface and SortedSet extends Set Interface.
- It behaves like a simple set with the exception that it stores elements in a sorted format. TreeSet uses a tree data structure for storage.
- Objects are stored in sorted, ascending order. But we can iterate in descending order using the method `TreeSet.descendingIterator()`.

Collection

↑

Set

↑

SortedSet

↑

TreeSet (class),

map interface

- In Java, Map Interface is present in java.util package represents a mapping between a key and a value. Java Map interface is not a subtype of the Collection interface. Therefore it behaves a bit differently from the rest of the collection types.
- A map contains unique keys.
- Since Map is an interface, objects cannot be created of the type map.
- There are 2 Map Interface :
 - Map
 - SortedMap
- There are 3 class implementations of maps :
 - HashMap
 - LinkedHashMap
 - TreeMap
- A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null values like the HashMap and LinkedHashMap, but some do not like the TreeMap.
- The order of a map depends on the specific implementations. For example, TreeMap and LinkedHashMap have predictable orders, while HashMap does not.

HashMap

map

↑

HashMap

- HashMap provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs.
- To access a value one must know its key.

LinkedHashMap

- LinkedHashMap is just like HashMap with the additional feature of maintaining an order of elements inserted into it.
 - HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order.
-

TreeMap

- The map is sorted according to the natural ordering of its keys, or by a **Comparator** provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and storing the key-value pairs.
-

Queue Interface

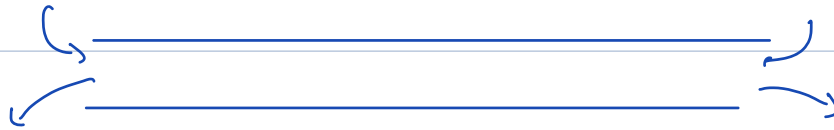
- The Queue interface is present in java.util package and extends the Collection interface is used to hold the elements about to be processed in FIFO(First In First Out) order.
 - It is an ordered list of objects with its use limited to inserting elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle.
-

PriorityQueue class

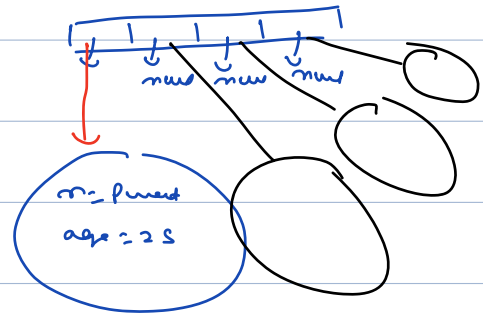
The beauty of this queue is, it removes the element in the order of their priority ,

Deque Interface

- Deque interface present in java.util package is a subtype of the queue interface. The Deque is related to the double-ended queue that supports the addition or removal of elements from either end of the data structure. It can either be used as a queue (first-in-first-out/FIFO) or as a stack (last-in-first-out/LIFO). Deque is the acronym for double-ended queue.
- The Deque (double-ended queue) interface in Java is a subinterface of the Queue interface and extends it to provide a double-ended queue, which is a queue that allows elements to be added and removed from both ends.



← Comparable →



```
main() {
```

```
    Person[] p = new Person[5];  
    p[0] = new Person("Puneet", 25);  
    p[1] = new Person("Vikas", 20);  
    p[2] = new Person("Anit", 28);  
    p[3] = new Person("Jagan", 30);  
    Array.sort(p);
```

```
}
```

```
class Person implements Comparable<Person> {
```

```
    String name;
```

```
    int age;
```

```
    Person (x,y) {
```

```
        | name=x;
```

```
        | age=y
```

```
    public int compareTo(Person other) { // myself -> -1
```

```
        other -> +1,
```

```
        if (age < other.age) {
```

```
            | return -1;
```

```
        } else if (age > other.age) {
```

```
            | return 1;
```

```
        } else {
```

```
            | return 0;
```

```
        }
```

← Comparator Interf ace :-

- In Java, a Comparator is an interface that provides a way to define custom ordering for objects in collections, such as lists or sets. It allows you to specify how elements should be compared and sorted based on specific criteria that you define. The Comparator interface is particularly useful when you want to sort objects in a way that differs from their natural order or when dealing with classes that don't implement the Comparable interface.

```
class Person {  
  
    String name;  
    int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

Collections.sort (,)

```
class AgeComparator implements Comparator<Person> {  
  
    @Override  
    public int compare(Person person1, Person person2) {  
        return Integer.compare(person1.age, person2.age);  
    }  
}
```

```
public static void main(String[] args) {  
  
    ArrayList List<Person> people = new ArrayList<>();  
  
    people.add(new Person("Alice", 28));  
    people.add(new Person("Bob", 22));  
    people.add(new Person("Charlie", 25));  
  
    // Sort the list using the AgeComparator  
    Collections.sort(people, new AgeComparator());  
  
    // Iterate the List of people and check if it is now sorted on the basis of age or not.  
    for (Person person : people) {  
        System.out.println(person.name + " - " + person.age);  
    }  
}
```