UNIT 4

Collections in Java

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).

A Collection represents a single unit of objects, i.e., a group.

Framework in Java

It provides readymade architecture.

It represents a set of classes and interfaces.

It is optional.

What is Collection framework

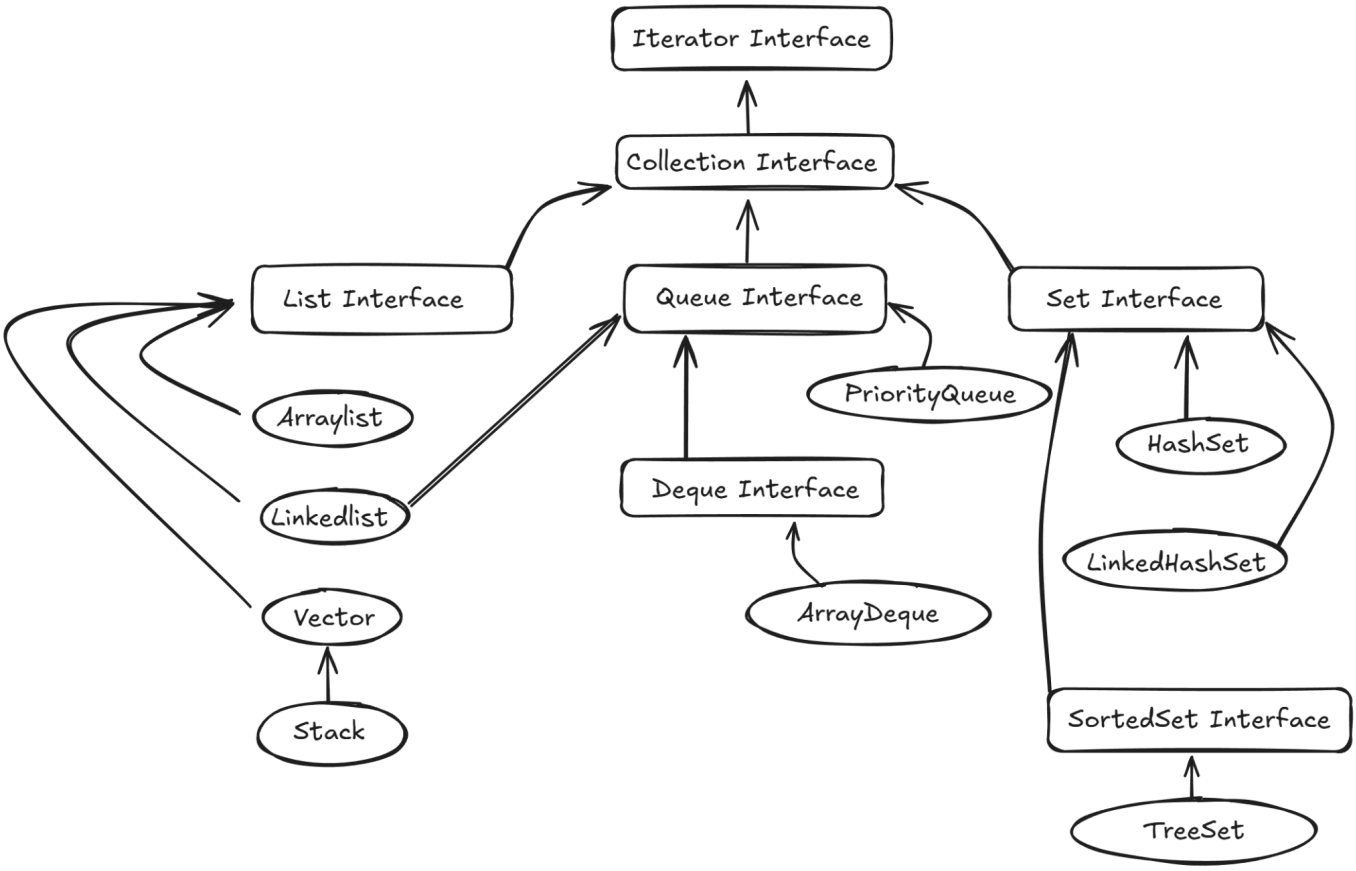
The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

Interfaces and its implementations, i.e., classes

Algorithm

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.



Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only. **All programs use of the Iterator interface to traverse a collection.**

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
It contains only one abstract method.

The **default method** of the Iterator interface in Java is forEachRemaining.
The Iterator interface has the following method:

```
default void forEachRemaining(Consumer<? super E> action)
```

This method performs the given action for each remaining element until all elements have been processed or the action throws an exception.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the
methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.
Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.
List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
To instantiate the List interface, we must use :

```
1. List <data-type> list1= new ArrayList();
2. List <data-type> list2 = new LinkedList();
3. List <data-type> list3 = new Vector();
4. List <data-type> list4 = new Stack();
```

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data
types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;
```

```java
class TestJavaCollection1 {
    public static void main(String args[]) {
        ArrayList<String> list = new ArrayList<String>(); // Creating ArrayList

        // Adding elements to the list
        list.add("Ravi");
        list.add("Vijay");
        list.add("Ravi"); // Duplicate allowed
        list.add("Ajay");

        // Traversing list through Iterator
        Iterator<String> itr = list.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Difference b/w Array and ArrayList

Basis	Array	ArrayList
Definition	An **array** is a dynamically-created object. It serves as a container that holds the constant number of values of the same type. It has a contiguous memory location.	The **ArrayList** is a class of Java **Collections** framework. It contains popular classes like **Vector, HashTable**, and **HashMap**.
Static/ Dynamic	Array is **static** in size.	ArrayList is **dynamic** in size.
Resizable	An array is a **fixed-length** data structure.	ArrayList is a **variable-length** data structure. It can be resized itself when needed.
Initialization	It is mandatory to provide the size of an array while initializing it directly or indirectly.	We can create an instance of ArrayList without specifying its size. Java creates ArrayList of default size.
Performance	It performs **fast** in comparison to ArrayList because of fixed size.	ArrayList is internally backed by the array in Java. The resize operation in ArrayList slows down the performance.
Primitive/ Generic type	An array can store both **objects** and **primitives** type.	We cannot store **primitive** type in ArrayList. It automatically converts primitive type to object.
Iterating Values	We use **for** loop or **for each** loop to iterate over an array.	We use an **iterator** to iterate over ArrayList.

Type-Safety	We cannot use generics along with array because it is not a convertible type of array.	ArrayList allows us to store only **generic/ type, that's why it is type safe.**
Length	Array provides a **length** variable which denotes the length of an array.	ArrayList provides the **size()** method to determine the size of ArrayList.
Adding Elements	We can add elements in an array by using the **assignment** operator.	Java provides the **add()** method to add elements in the ArrayList.
Single/ Multi Dimensional	Array can be **multi-dimensional**.	ArrayList is always **single-dimensional**.

LinkedList

LinkedList implements the Collection interface. Java LinkedList class uses a 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:It uses a doubly linked list internally to store the elements. It can store the
duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

```
import java.util.*;

public class TestJavaCollection2 {
    public static void main(String args[]) {
        LinkedList<String> al = new LinkedList<String>();

        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi"); // Duplicate allowed
        al.add("Ajay");

        Iterator<String> itr = al.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Vector

Vector is like the *dynamic array* which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the *List* interface, so we can use all the methods of List interface here.
It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case. The Iterators returned by the Vector class are *fail-fast*. In case of concurrent modification, it fails and throws the ConcurrentModificationException.

```
import java.util.*;
public class TestJavaCollection3 {
    public static void main(String args[]) {
        Vector<String> v = new Vector<String>();
        v.add("Ayush");
        v.add("Amit");
        v.add("Ashish");
        v.add("Garima");
        Iterator<String> itr = v.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Stack

The **stack** is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First Out** (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the **Stack class** that provides different operations such as push, pop, search, etc.

The stack data structure has the two most important operations that are **push** and **pop**. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack.

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

```
import java.util.*;

public class TestJavaCollection4 {
    public static void main(String args[]) {
        Stack<String> stack = new Stack<String>();

        stack.push("Ayush");
        stack.push("Garvit");
        stack.push("Amit");
        stack.push("Ashish");
        stack.push("Garima");

        stack.pop(); // Removes "Garima"

        Iterator<String> itr = stack.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements
which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.
Queue interface can be instantiated as:

Queue<String> q1 = **new** PriorityQueue();

Queue<String> q2 = **new** ArrayDeque();

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue (OPTIONAL)

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.
Deque can be instantiated as:

```
Deque d = new ArrayDeque();
```

```
import java.util.*;

public class TestJavaCollection6 {

    public static void main(String[] args) {
        // Creating Deque and adding elements
        Deque<String> deque = new ArrayDeque<String>();
        deque.add("Gautam");
        deque.add("Karan");
        deque.add("Ajay");

        // Traversing elements
        for (String str : deque) {
            System.out.println(str);
        }
    }
}
```

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.
ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Set Interface

Set Interface in Java is present in java.util package. It 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. Set is implemented by HashSet, LinkedHashSet, and TreeSet.
Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

Basic Methods

boolean add(): Adds the specified element to the set if it is not already present.
boolean addAll(): Adds all of the elements in the specified collection to the set if they are not already present. boolean remove(): Removes the specified element from the set if it is present.
boolean removeAll(): Removes from the set all of its elements that are contained in the specified collection.

boolean retainAll(): Retains only the elements in the set that are contained in the specified collection . void clear(): Removes all of the elements from the set.

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

```
import java.util.*;

public class TestJavaCollection7 {
    public static void main(String args[]) {
        // Creating HashSet and adding elements
        HashSet<String> set = new HashSet<String>();

        set.add("Ravi");
        set.add("Vijay");
        set.add("Ravi"); // Duplicate, will not be added
        set.add("Ajay");

        // Traversing elements
        Iterator<String> itr = set.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and
implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

```
import java.util.*;

public class TestJavaCollection8 {
    public static void main(String args[]) {
        LinkedHashSet<String> set = new LinkedHashSet<String>();

        set.add("Ravi");
        set.add("Vijay");
        set.add("Ravi"); // Duplicate, will not be added
        set.add("Ajay");

        Iterator<String> itr = set.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional
methods that inhibit the natural ordering of the elements.

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique
elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

```
import java.util.*;

public class TestJavaCollection9 {
    public static void main(String args[]) {
        // Creating and adding elements
        TreeSet<String> set = new TreeSet<String>();

        set.add("Ravi");
        set.add("Vijay");
        set.add("Ravi"); // Duplicate, will not be added
```

```
        set.add("Ajay");

        // Traversing elements (in sorted order)
        Iterator<String> itr = set.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A M ap contains unique keys.
A Map is useful if you have to search, update or delete elements on the basis of a key.
A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null ke ys and values, but TreeMap doesn't allow any null key or value.

Java HashMap

Hashing It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.
Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be
unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perfor m operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.
HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as
well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key a nd V for value. It inherits the AbstractMap class and implements the Map interface.

Points to remember

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

```
import java.util.*;

public class HashMapExample1 {
    public static void main(String args[]) {
        // Creating HashMap
        HashMap<Integer, String> map = new HashMap<Integer, String>();

        // Put elements in Map
        map.put(1, "Mango");
        map.put(2, "Apple");
```

```
        map.put(3, "Banana");
        map.put(4, "Grapes");

        System.out.println("Iterating HashMap...");
        for (Map.Entry<Integer, String> m : map.entrySet()) {
            System.out.println(m.getKey() + " " + m.getValue());
        }
    }
}
```

Java LinkedHashMap class

Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iterat ion order. It inherits HashMap class and implements the Map interface.

Points to remember

- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

```
import java.util.*;

class LinkedHashMap1 {
    public static void main(String args[]) {

        LinkedHashMap<Integer, String> hm = new LinkedHashMap<Integer, String>();

        hm.put(100, "Amit");
        hm.put(101, "Vijay");
        hm.put(102, "Rahul");

        for (Map.Entry<Integer, String> m : hm.entrySet()) {
            System.out.println(m.getKey() + " " + m.getValue());
        }
    }
}
```

Java TreeMap class

Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-va lue pairs in sorted order.
The important points about Java TreeMap class are:

- Java TreeMap contains values based on the key.

  It implements the NavigableMap interface and extends AbstractMap class.
- Java TreeMap contains only unique elements.
- Java TreeMap cannot have a null key but can have multiple null values.
- Java TreeMap is non synchronized.
- Java TreeMap maintains ascending order.

```
import java.util.*;
class TreeMap1{
public static void main(String args[]){
   TreeMap<Integer,String> map=new TreeMap<Integer,String>();
      map.put(100,"Amit");
      map.put(102,"Ravi");
      map.put(101,"Vijay");
      map.put(103,"Rahul");

      for(Map.Entry m:map.entrySet()){
       System.out.println(m.getKey()+" "+m.getValue());
      }
 }
}
```

Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

Points to remember

- A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Example: remove()

```
import java.util.*;

public class Hashtable2 {
    public static void main(String args[]) {
        Hashtable<Integer, String> map = new Hashtable<Integer, String>();

        map.put(100, "Amit");
        map.put(102, "Ravi");
        map.put(101, "Vijay");
        map.put(103, "Rahul");

        System.out.println("Before remove: " + map);

        // Remove value for key 102
        map.remove(102);

        System.out.println("After remove: " + map);
    }
}
```

Difference between HashMap and Hashtable

HashMap	Hashtable
HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread safe and can be shared with many threads.
HashMap **allows one null key and multiple null values**.	Hashtable **doesn't allow any null key or value**.
HashMap is a **new class introduced in JDK 1.2**.	Hashtable is a **legacy class**.
HashMap is **fast**.	Hashtable is **slow**.
We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
HashMap is **traversed by Iterator**.	Hashtable is **traversed by Enumerator and Iterator**.
Iterator in HashMap is **fail-fast**.	Enumerator in Hashtable is **not fail-fast**.
HashMap inherits **AbstractMap** class.	Hashtable inherits **Dictionary** class.

Sorting in Collection

Collections
class provides static methods for sorting the elements of a collection. If collection elements are of a Set type, we can use TreeSet. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.
We can sort the elements of:

- String objects
- Wrapper class objects
- User-defined class objects

Example to sort string objects

```
import java.util.*;

class TestSort1 {
    public static void main(String args[]) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("Viru");
        al.add("Saurav");
        al.add("Mukesh");
        al.add("Tahir");

        Collections.sort(al);

        Iterator<String> itr = al.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Java Comparable interface

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang
package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can
sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

```
class Student implements Comparable<Student> {
    int rollno;
    String name;
    int age;

    Student(int rollno, String name, int age) {
        this.rollno = rollno;
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Student st) {
        if (this.age == st.age)
            return 0;
        else if (this.age > st.age)
```

```
            return 1;
        else
            return -1;
    }
}
```

```
import java.util.*;

public class TestSort1 {
    public static void main(String args[]) {
        ArrayList<Student> al = new ArrayList<Student>();
        al.add(new Student(101, "Vijay", 23));
        al.add(new Student(106, "Ajay", 27));
        al.add(new Student(105, "Jai", 21));

        Collections.sort(al);

        for (Student st : al) {
            System.out.println(st.rollno + " " + st.name + " " + st.age);
        }
    }
}
```

Java Comparator interface

Java Comparator interface is used to order the objects of a user-defined class.
This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).
It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Difference between Comparable and Comparator

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such

	as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

Properties class in Java

The **properties** object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.

An Advantage of the properties file

Recompilation is not required if the information is changed from a properties file: If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.