Unit-4

Collections in Java

Any group of individual objects that are represented as a single unit is known as a Java Collection of Objects.

In Java, a separate framework named the *"Collection Framework"* has been defined in JDK 1.2 which holds all the Java Collection Classes and Interface in it.

In Java, the Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main "root" interfaces of Java collection classes.

What is a Framework in Java?

A framework is a set of classes and interfaces which provide a ready-made architecture. In order to implement a new feature or a class, there is no need to define a framework. However, an optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task.

Advantages of the Java Collection Framework

Since the lack of a collection framework gave rise to the above set of disadvantages, the following are the advantages of the collection framework.

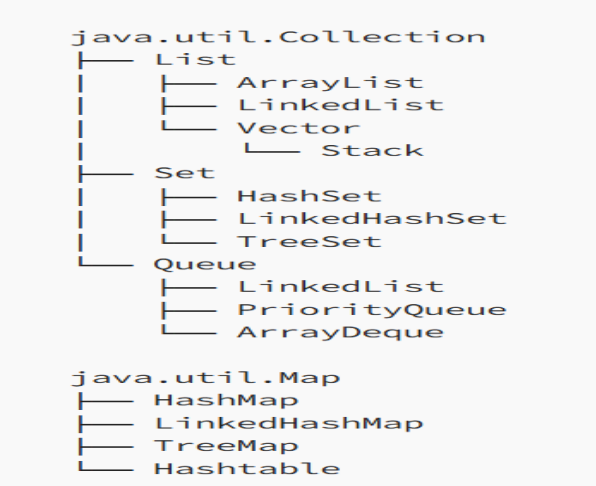
1. **Consistent API:** The API has a basic set of interfaces like *Collection*, *Set*, *List*, or *Map*, all the classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have *some* common set of methods.

2. **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.

3. **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.

Hierarchy of the Collection Framework in Java



- **Difference between a class and an interface**

- **Class:** A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

- **Interface:** Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, nobody). Interfaces specify what a class must do and not how. It is the blueprint of the class.

Methods of the Collection Interface

This interface contains various methods which can be directly used by all the collections which implement this interface. They are:

Method	Description
add(Object)	This method is used to add an object to the collection.
addAll(Collection c)	This method adds all the elements in the given collection to this collection.
clear()	This method removes all of the elements

Method	Description
	from this collection.
contains(Object o)	This method returns true if the collection contains the specified element.
containsAll(Collection c)	This method returns true if the collection contains all of the elements in the given collection.
equals(Object o)	This method compares the specified object with this collection for equality.
hashCode()	This method is used to return the hash code value for this collection.
isEmpty()	This method returns true if this collection contains no elements.
iterator()	This method returns an iterator over the elements in this collection.
parallelStream()	This method returns a parallel Stream with this collection as its source.
remove(Object o)	This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object.
removeAll(Collection c)	This method is used to remove all the objects mentioned in the given collection from the collection.
removeIf(Predicate filter)	This method is used to remove all the elements of this collection that satisfy the given predicate.
retainAll(Collection c)	This method is used to retain only the elements in this collection that are contained in the specified collection.

Method	Description
size()	This method is used to return the number of elements in the collection.
spliterator()	This method is used to create a Spliterator over the elements in this collection.
stream()	This method is used to return a sequential Stream with this collection as its source.
toArray()	This method is used to return an array containing all of the elements in this collection.

Interfaces that Extend the Java Collections Interface

The collection framework contains multiple interfaces where every interface is used to store a specific type of data. The following are the interfaces present in the framework.

1. Iterable Interface

This is the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections. Therefore, this interface contains only one abstract method which is the iterator. It returns the

Iterator iterator();

Methods of Iterator

The Iterator interface provides 4 methods that can be used to perform various operations on elements of collections.

hasNext() - returns true if there exists an element in the collection

next() - returns the next element of the collection

remove() - removes the last element returned by the next()

forEachRemaining() - performs the specified action for each remaining element of the collection

2. Collection Interface

This interface extends the iterable interface and is implemented by all the classes in the collection framework. This interface contains all the basic methods which every collection has like adding the data into the collection, removing the data, clearing the data, etc. All these methods are implemented in this interface because these methods are implemented by all the classes irrespective of their style of implementation. And also, having these methods in this interface ensures that the names of the methods are universal for all the collections. Therefore, in short, we can say that this interface builds a foundation on which the collection classes are implemented.

3. List Interface

This is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collections of the objects. This also allows duplicate data to be present in it. This list interface is implemented by various classes like ArrayList, Vector, Stack, etc. Since all the subclasses implement the list, we can instantiate a list object with any of these classes.

For example:

List <T> al = new ArrayList<> ();
List <T> ll = new LinkedList<> ();
List <T> v = new Vector<> ();
Where T is the type of the object

The classes which implement the List interface are as follows:

i). ArrayList

ArrayList 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. The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection. Java ArrayList allows us to randomly access the list.

ArrayList can not be used for primitive types, like int, char, etc. We will need a wrapper class for such cases.

ii). LinkedList

The LinkedList class is an implementation of the LinkedList data structure which 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.

iii). Vector

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.

iv). Stack

Stack 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 empty, search, and peek. The class can also be referred to as the subclass of Vector.

4. Queue Interface

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 on a first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket. There are various classes like PriorityQueue, ArrayDeque, etc. Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes.

5. Set Interface

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. This set interface is implemented by various classes like HashSet, TreeSet, LinkedHashSet, etc. Since all the subclasses implement the set, we can instantiate a set object with any of these classes.

For example:

Set<T> hs = new HashSet<> ();
Set<T> lhs = new LinkedHashSet<> ();
Set<T> ts = new TreeSet<> ();
Where T is the type of the object.

The following are the classes that implement the Set interface:

i). HashSet

The HashSet class is an inherent implementation of the hash table data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hashcode. This class also allows the insertion of NULL elements.

ii). LinkedHashSet

A LinkedHashSet is very similar to a HashSet. The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements.

6. Sorted Set Interface

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. The class which implements this interface is TreeSet. Since this class implements the SortedSet, we can instantiate a SortedSet object with this class.

TreeSet

The TreeSet class uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface. It can also be ordered by a Comparator provided at a set creation time, depending on which constructor is used.

```
java.util.Collection
├── List
│     ├── ArrayList
│     ├── LinkedList
│     └── Vector
│           └── Stack
├── Set
│     ├── HashSet
│     ├── LinkedHashSet
│     └── TreeSet
└── Queue
      ├── LinkedList
      ├── PriorityQueue
      └── ArrayDeque

java.util.Map
├── HashMap
├── LinkedHashMap
├── TreeMap
└── Hashtable
```

Working of Queue Data Structure

In queues, elements are stored and accessed in First In, First Out manner. That is, elements are added from the behind and removed from the front.

Methods of Queue

The Queue interface includes all the methods of the Collection interface. It is because Collection is the super interface of Queue.

Some of the commonly used methods of the Queue interface are:

- add() - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.
- offer() - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.
- element() - Returns the head of the queue. Throws an exception if the queue is empty.
- peek() - Returns the head of the queue. Returns null if the queue is empty.
- remove() - Returns and removes the head of the queue. Throws an exception if the queue is empty.
- poll() - Returns and removes the head of the queue. Returns null if the queue is empty.

- Ex: Implementing the LinkedList Class

```
import java.util.Queue;
import java.util.LinkedList;

class Main {

    public static void main(String[] args) {
        // Creating Queue using the LinkedList class
        Queue<Integer> numbers = new LinkedList<>();

        // offer elements to the Queue
        numbers.offer(1);
        numbers.offer(2);
        numbers.offer(3);
        System.out.println("Queue: " + numbers);

        // Access elements of the Queue
        int accessedNumber = numbers.peek();
        System.out.println("Accessed Element: " + accessedNumber);

        // Remove elements from the Queue
        int removedNumber = numbers.poll();
        System.out.println("Removed Element: " + removedNumber);

        System.out.println("Updated Queue: " + numbers);
    }
}
```

Output

```
Queue: [1, 2, 3]
Accessed Element: 1
Removed Element: 1
Updated Queue: [2, 3]
```

Ex: **Implementing the PriorityQueue Class**

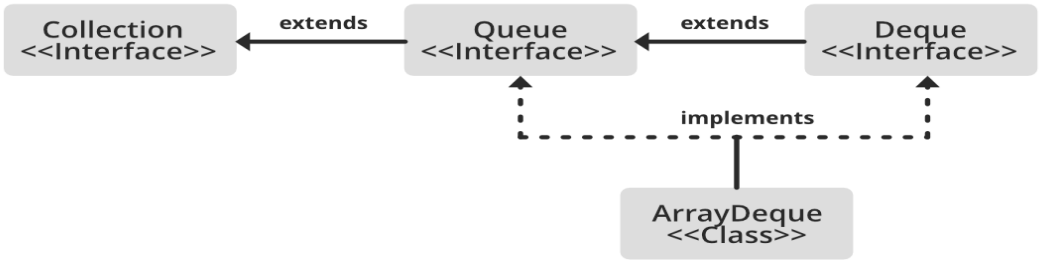
```
import java.util.Queue;

import java.util.PriorityQueue;

class Main {
```

```
    public static void main(String[] args) {
        // Creating Queue using the PriorityQueue class
        Queue<Integer> numbers = new PriorityQueue<>();

        // offer elements to the Queue
        numbers.offer(5);
        numbers.offer(1);
        numbers.offer(2);
        System.out.println("Queue: " + numbers);

        // Access elements of the Queue
        int accessedNumber = numbers.peek();
        System.out.println("Accessed Element: " + accessedNumber);

        // Remove elements from the Queue
        int removedNumber = numbers.poll();
        System.out.println("Removed Element: " + removedNumber);

        System.out.println("Updated Queue: " + numbers);
    }
}
```

Output

```
Queue: [1, 5, 2]
Accessed Element: 1
Removed Element: 1
Updated Queue: [2, 5]
```

Methods of ArrayDeque : The ArrayDeque class provides implementations for all the methods present in Queue and Deque interface.




```java
import java.util.ArrayDeque;

class Main {

    public static void main(String[] args) {

        ArrayDeque<String> animals= new ArrayDeque<>();


        // Using add()

        animals.add("Dog");


        // Using addFirst()

        animals.addFirst("Cat");


        // Using addLast()

        animals.addLast("Horse");

        System.out.println("ArrayDeque: " + animals);
```

```
    }

}
```

Output

```
ArrayDeque: [Cat, Dog, Horse]
```

Insert elements using offer(), offerFirst() and offerLast()

- `offer()` - inserts the specified element at the end of the array deque
- `offerFirst()` - inserts the specified element at the beginning of the array deque
- `offerLast()` - inserts the specified element at the end of the array deque

Note: If the array deque is full

- **the add() method will throw an exception**
- **the offer() method returns false**

```
import java.util.ArrayDeque;

class Main {

    public static void main(String[] args) {

        ArrayDeque<String> animals= new ArrayDeque<>();

        // Using offer()

        animals.offer("Dog");


        // Using offerFirst()

        animals.offerFirst("Cat");


        // Using offerLast()

        animals.offerLast("Horse");

        System.out.println("ArrayDeque: " + animals);

    }
```

```
}
```

Output

```
ArrayDeque: [Cat, Dog, Horse]
```

7. Map Interface

A map is a data structure that supports the key-value pair for mapping the data.

This interface doesn't support duplicate keys because the same key cannot have multiple mappings, however, it allows duplicate values in different keys.

A map is useful if there is data and we wish to perform operations on the basis of the key.

This map interface is implemented by various classes like HashMap, TreeMap, etc.

Since all the subclasses implement the map, we can instantiate a map object with any of these classes.

For example:

Map<T> hm = new HashMap<> ();
Map<T> tm = new TreeMap<> ();

Where T is the type of the object.

HashMap

HashMap provides the basic implementation of the Map interface of Java.

It stores the data in (Key, Value) pairs.

To access a value in a HashMap, we must know its key.

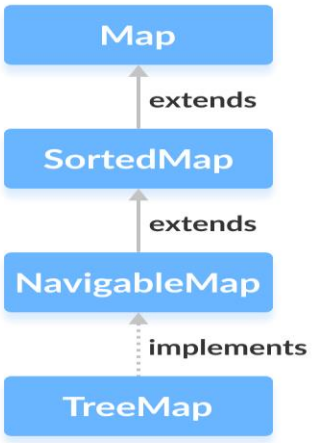
HashMap uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String so that the indexing and search operations are faster.

HashSet also uses HashMap internally.

Java TreeMap

The TreeMap class of the Java collections framework provides the tree data structure implementation.

It implements the NavigableMap interface.



Hashtable class

Hashtable class, introduced as part of the Java Collections framework, implements a hash table that maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method. The **java.util.Hashtable** class is a class in Java that provides a **key-value** data structure, similar to the Map interface.

- It is similar to HashMap, but is synchronized.
- Hashtable stores key/value pair in hash table.
- In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.
- HashMap doesn't provide any Enumeration, while Hashtable provides not fail-fast Enumeration.

Ex :

```
import java.util.Hashtable;
public class Main
{
   public static void main(String args[])
   {
        // Create a Hashtable of String  keys and Integer values
```

```
        Hashtable<String, Integer> ht = new Hashtable<>();
        // Adding elements to the Hashtable
        ht.put("One ", 1);
        ht.put("Two ", 2);
        ht.put("Three ", 3);
              // Displaying the Hashtable elements
        System.out.println("Hashtable Elements: " + ht);
    }
}
```

In Java, both Comparable and Comparator interfaces are used for sorting objects.

The main **difference between Comparable and Comparator** is:

- **Comparable Interface and Comparator Interface**

- **Comparable:** It is used to define the **natural ordering of the objects** within the class.
- **Comparator**: It is used to define **custom sorting logic** externally.

Difference Between Comparable and Comparator

The table below demonstrates the difference between comparable and comparator in Java.

Features	**Comparable**	**Comparator**
Definition	It defines natural ordering within the class.	It defines external or custom sorting logic.
Method	compareTo()	compare()
Implementation	It is implemented in the class.	It is implemented in a separate class.
Sorting Criteria	Natural order sorting	Custom order sorting

Features	Comparable	Comparator
Usage	It is used for a single sorting order.	It is used for multiple sorting orders.

```
import java.util.ArrayList;
import java.util.Collections;
class Emp implements Comparable<Emp>
{
	int eid,sal;
	String ename;
	public Emp(int eid,String ename, int sal)
	{
		this.eid=eid;
		this.ename=ename;
		this.sal=sal;
	}


	public int compareTo(Emp e)
	{
		if(sal==e.sal)
			return 0;
		else if(sal>e.sal)
			return 1;
		else
			return -1;
	}

}
```

Or

```
	public int compareTo(Emp e)
	{
		return ename.compareTo(e.ename);
	}
```

```
}
public class Main
{
   public static void main(String[] args)
        {
         ArrayList<Emp> p=new ArrayList<>();
         p.add(new Emp(1,"ravi",10000));
        p.add(new Emp(12,"rashi",5000));

        p.add(new Emp(13,"ravj",15000));

        p.add(new Emp(1,"raju",2000));
        Collections.sort(p);

        for (Emp e:p)
        {
                System.out.println(e.eid+"\t"+e.ename+"\t"+e.sal);
        }

        }
}
```

Properties Class in Java

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. It belongs to **java.util** package. **Properties** define the following instance variable. This variable holds a default property list associated with a **Properties** object.

> ***Properties defaults:*** *This variable holds a default property list associated with a Properties object.*

Features of Properties class:

- **Properties** is a subclass of Hashtable.
- It is used to maintain a list of values in which the key is a string and the value is also a string i.e; it can be used to store and retrieve string type data from the properties file.

- Properties class can specify other properties list as it's the default. If a particular key property is not present in the original Properties list, the default properties will be searched.
- Properties object does not require external synchronization and Multiple threads can share a single Properties object.
- Also, it can be used to retrieve the properties of the system.

Advantage of a Properties file

In the event that any data is changed from the properties record, you don't have to recompile the java class. It is utilized to store data that is to be changed habitually.