Java Collection Framework

1. Collections in Java
2. Collections Class in Java
3. Collection Interface in Java
4. Details of List Interface (ArrayList, LinkedList, Vector, Stack)
5. Details of Set Interface (SortedSet(I), TreeSet, Hashset, LinkedHashSet)
6. Details of Queue Interface (Deque(I), PriorityQueue, ArrayDeque)
7. Details of Map Interface (HashMap, LinkedHashMap, TreeMap)
8. Cursors interfaces of Collection (Enumeration, Iterator, ListIterator)
9. Sorting interfaces of Collection (Comparable, Comparator)

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.

Need for a Separate Collection Framework in Java

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. And also, it is very difficult for the users to remember all the different methods, syntax, and constructors present in every collection class.

Let's understand this with an example of adding an element in a hashtable and a vector.

Example:

```
class CollectionDemo {
    public static void main(String[] args) {
        int arr[] = new int[] { 1, 2, 3, 4 };
        Vector<Integer> v = new Vector();
        Hashtable<Integer, String> h = new Hashtable();
        v.addElement(1);
        v.addElement(2);
        h.put(1, "Test");
        h.put(2, "Test1");
        System.out.println(arr[0]);
        System.out.println(v.elementAt(0));
        System.out.println(h.get(1));
    }
}
Output
1
1
Test
```

Note: As we can observe, none of these collections(Array, Vector, or Hashtable) implements a standard member access interface, it was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback is that most of the 'Vector' methods are final, meaning we cannot extend the 'Vector' class to implement a similar kind of Collection. Therefore, Java developers decided to come up with a common interface to deal with the above-mentioned problems and introduced the Collection Framework in JDK 1.2 post which both, legacy Vectors and Hashtables were modified to conform to the Collection Framework.

Advantages of the Java Collection Framework

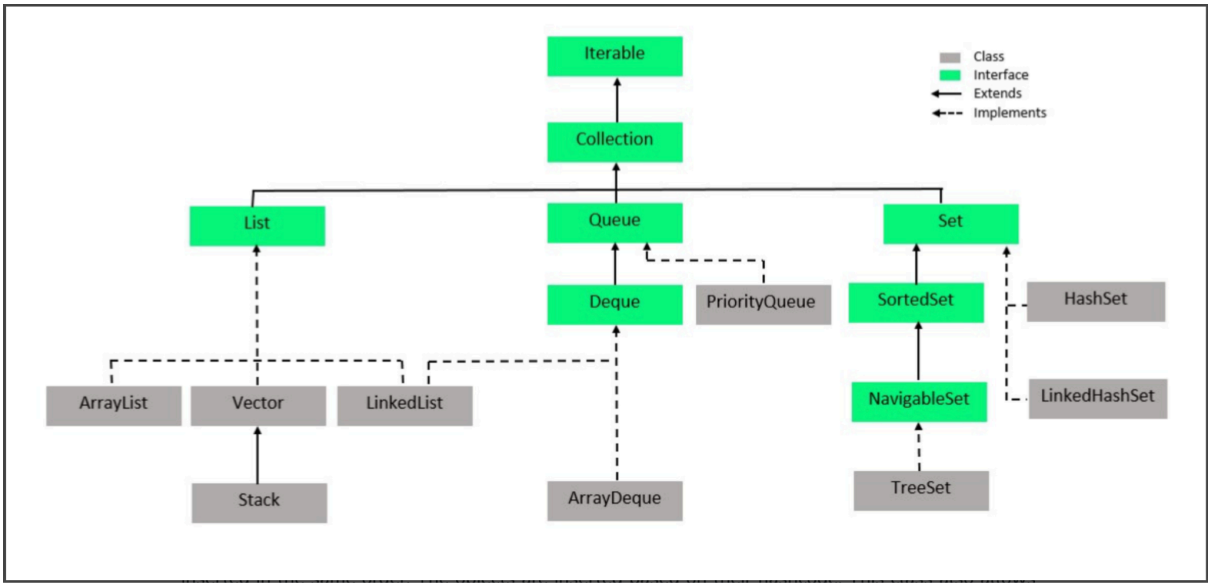
Since the lack of a collection framework gave rise to the above set of disadvantages, the following are the advantages of the collection framework.

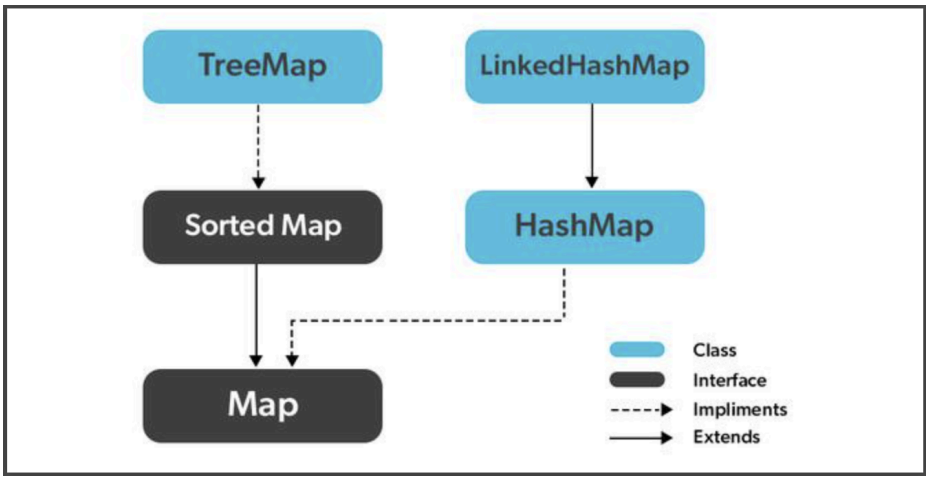
- Consistent API: The API has a basic set of interfaces like Collection, Set, List, or Map, all the classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.
- Reduces programming effort: A programmer doesn't have to worry about the design of the Collection 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.

Hierarchy of the Collection Framework in Java

The utility package, (java.util) contains all the classes and interfaces that are required by the collection framework. The collection framework contains an interface named an iterable interface which provides the iterator to iterate

through all the collections. This interface is extended by the main collection interface which acts as a root for the collection framework. All the collections extend this collection interface thereby extending the properties of the iterator and the methods of this interface. The following figure illustrates the hierarchy of the collection framework.





Collections Class in Java

Collections class in Java is one of the utility classes in the Java Collections Framework. The java.util package contains the Collections class in Java. The Java Collections class is used with the static methods that operate on the collections or return the collection. All the methods of this class throw a NullPointerException if the collection or object passed to the methods is null.

Example 1: Here, we will use ArrayList, which is a class from the Java Collections framework. It allows for storing elements in a list by maintaining the insertion order and also allows duplicates.

```
import java.util.ArrayList;

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

        al.add("Apple");
        al.add("Banana");
        al.add("Apple");

        System.out.println("" + al);
    }
}

Output
[Apple, Banana, Apple]
```

Collection Class Declaration

The Declaration of collection class is listed below:

```
public class Collections extends Object
```

Remember: Object is the parent class of all the classes.

Common Java Collection Classes

Collection Framework contains both classes and interfaces. Although both seem the same, there are certain differences between Collection classes and the Collections framework.

The Collection classes in Java are mentioned below:

1. ArrayList

ArrayList is a class implemented using a list interface that provides the functionality of a dynamic array where the size of the array is not fixed.

Syntax:

```
ArrayList<_type_> var_name = new ArrayList<_type_>();
```

2. Vector

Vector is a Part of the collection class that implements a dynamic array that can grow or shrink its size as required.

Syntax:

```
public class Vector<E> extends AbstractList<E> implements List<E>,
RandomAccess,
Cloneable, Serializable
```

3. Stack

Stack is a part of the Java collection class that models and implements a Stack data structure. It is based on the basic principle of last-in-first-out(LIFO).

Syntax:

```
public class Stack<E> extends Vector<E>
```

4. LinkedList

LinkedList class is an implementation of the LinkedList data structure. It can store the elements that are not stored in contiguous locations and every element is a separate object with a different data part and different address part.

Syntax:

```
LinkedList<_type_> var_name = new LinkedList<_type_>();
```

5. HashSet

HashSet is implemented using the Hashtable data structure. It helps us add, remove, or find items very quickly, no matter how many items there are.

Syntax:

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>,
Cloneable, Serializable
```

Note: HashSet internally uses HashMap not Hashtable.

6. LinkedHashSet

LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements.

Syntax:

```
public class LinkedHashSet<E> extends HashSet<E> implements Set<E>,
Cloneable, Serializable
```

7. TreeSet

TreeSet class is an implementation of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether an explicit comparator is provided or not.

Syntax:

```
TreeSet<E> set = new TreeSet<>();
```

8. PriorityQueue

The PriorityQueue keeps the elements in order based on their importance. We can use the normal order or can choose our own way to decide which item comes first.

Syntax:

```
public class PriorityQueue<E> extends AbstractQueue<E> implements
Serializable
```

9. ArrayDeque

ArrayDeque is a class that works with double ended queue. It uses a resizable array to store elements. It provides constant-time for adding and removing elements from both the ends.

Syntax:

```
public class ArrayDeque<E> extends
AbstractCollection<E> implements Deque<E>, Cloneable,Serializable
```

10. HashMap

HashMap Class is similar to HashTable but the data is unsynchronized. It stores the element in (Key, Value) pairs, and we can access the elements with the help of index.

Syntax:

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

11. EnumMap

EnumMap extends AbstractMap and implements the Map interface in Java.

Syntax:

```
public class EnumMap<K extends Enum<K>,V> extends
AbstractMap<K,V> implements Serializable, Cloneable
```

12. AbstractMap

The AbstractMap class is a part of the Java Collection Framework. It implements the Map interface to provide a structure to it, by doing so it makes the further implementations easier.

Syntax:

```
public abstract class AbstractMap<K,V> implements Map<K,V>
```

13. TreeMap

A TreeMap is implemented using a Red-Black tree. It is a kind of map that keys the keys in order. The keys are sorted either by their usual order or by a custom rule we give when creating the map.

Syntax:

```
SortedMap<K, V> m = Collections.synchronizedSortedMap(new TreeMap<>());
```

Java Collections Class Fields

The collection class contains 3 fields as listed below which can be used to return immutable entities.

1. EMPTY_LIST to get an immutable empty List
2. EMPTY_SET to get an immutable empty Set
3. EMPTY_MAP to get an immutable empty Map

Methods of Collections Class

Now let us discuss methods that are present inside this class so that we can use these inbuilt functionalities later on in our program. Below are the methods have been listed below in a tabular format as shown below as follows:

Methods	Description

addAll(Collection<? super T> c, T... elements)	This method is used to insert the specified collection elements to the specified collection.
asLifoQueue(Deque<T> deque)	This method returns a view of a Deque as a Last-in-first-out (Lifo) Queue.
binarySearch(List<? extends Comparable> list, T key)	This method searches the key using binary search in the specified list.
binarySearch(List<? extends T> list, T key, Comparator<? super T> c)	This method searches the specified list for the specified object using the binary search algorithm.
checkedCollection(Collection<E> c, Class<E> type)	This method returns a dynamically typesafe view of the specified collection.
checkedList(List<E> list, Class<E> type)	This method returns a dynamically typesafe view of the specified list.
checkedMap(Map<K,V> m, Class<K> keyType, Class<V> valueType)	This method returns a dynamically typesafe view of the specified map.
checkedNavigableMap(NavigableMap< K,V> m, Class<K> keyType, Class<V> valueType)	This method returns a dynamically typesafe view of the specified navigable map.
checkedNavigableSet(NavigableSet<E > s, Class<E> type)	This method returns a dynamically typesafe view of the specified navigable set.
checkedQueue(Queue<E> queue, Class<E> type)	This method returns a dynamically typesafe view of the specified queue.
checkedSet(Set<E> s, Class<E> type)	This method returns a dynamically typesafe view of the specified set.
checkedSortedMap(SortedMap<K,V> m, Class<K> keyType, Class<V> valueType)	This method returns a dynamically typesafe view of the specified sorted map.
copy(List<? super T> dest, List<? extends T> src)	This method copies all of the elements from one list into another.
disjoint(Collection<?> c1, Collection<?> c2)	This method returns true if the two specified collections have no elements in common.
emptyEnumeration()	This method returns an enumeration that has no elements.

emptyIterator()	This method returns an iterator that has no elements.
emptyList()	This method returns an empty list (immutable).
emptyListIterator()	This method returns a list iterator that has no elements.
emptyMap()	This method returns an empty map (immutable).
emptyNavigableMap()	This method returns an empty navigable map (immutable).
emptyNavigableSet()	This method returns an empty navigable set (immutable).
emptySet()	This method returns an empty set (immutable).
emptySortedMap()	This method returns an empty sorted map (immutable).
emptySortedSet()	This method returns an empty sorted set (immutable).
enumeration(Collection<T> c)	This method returns an enumeration over the specified collection.
fill(List<? super T> list, T obj)	This method replaces all of the elements of the specified list with the specified element.
frequency(Collection<?> c, Object o)	This method returns the number of elements in the specified collection equal to the specified object.
indexOfSubList(List<?> source, List<?> target)	This method returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
lastIndexOfSubList(List<?> source, List<?> target)	This method returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
ist(Enumeration<T> e)	This method returns an array list

	containing the elements returned by the specified enumeration in the order they are returned by the enumeration.
max(Collection<? extends T> coll)	This method returns the maximum element of the given collection, according to the natural ordering of its elements.
max(Collection<? extends T> coll, Comparator<? super T> comp)	This method returns the maximum element of the given collection, according to the order induced by the specified comparator.
min(Collection<? extends T> coll)	This method returns the minimum element of the given collection, according to the natural ordering of its elements.
min(Collection<? extends T> coll, Comparator<? super T> comp)	This method returns the minimum element of the given collection, according to the order induced by the specified comparator.
nCopies(int n, T o)	This method returns an immutable list consisting of n copies of the specified object.
newSetFromMap(Map<E,Boolean> map)	This method returns a set backed by the specified map.
replaceAll(List<T> list, T oldVal, T newVal)	This method replaces all occurrences of one specified value in a list with another.
reverse(List<?> list)	This method reverses the order of the elements in the specified list
reverseOrder()	This method returns a comparator that imposes the reverse of the natural ordering on a collection of objects that implement the Comparable interface.
reverseOrder(Comparator<T> cmp)	This method returns a comparator that imposes the reverse ordering of the specified comparator.
rotate(List<?> list, int distance)	This method rotates the elements in the specified list by the specified distance.

shuffle(List<?> list)	This method randomly permutes the specified list using a default source of randomness.
shuffle(List<?> list, Random rnd)	This method randomly permutes the specified list using the specified source of randomness.
singletonMap(K key, V value)	This method returns an immutable map, mapping only the specified key to the specified value.
singleton(T o)	This method returns an immutable set containing only the specified object.
singletonList(T o)	This method returns an immutable list containing only the specified object.
sort(List<T> list)	This method sorts the specified list into ascending order, according to the natural ordering of its elements.
sort(List<T> list, Comparator<? super T> c)	This method sorts the specified list according to the order induced by the specified comparator.
swap(List<?> list, int i, int j)	This method swaps the elements at the specified positions in the specified list.
synchronizedCollection(Collection<T> c)	This method returns a synchronized (thread-safe) collection backed by the specified collection.
synchronizedList(List<T> list)	This method returns a synchronized (thread-safe) list backed by the specified list.
synchronizedMap(Map<K,V> m)	This method returns a synchronized (thread-safe) map backed by the specified map.
synchronizedNavigableMap(NavigableMap<K,V> m)	This method returns a synchronized (thread-safe) navigable map backed by the specified navigable map.
synchronizedNavigableSet(NavigableSet<T> s)	This method returns a synchronized (thread-safe) navigable set backed by the specified navigable set.

synchronizedSet(Set<T> s) .	This method returns a synchronized (thread-safe) set backed by the specified set
synchronizedSortedMap(SortedMap<K ,V> m)	This method returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
synchronizedSortedSet(SortedSet<T> s)	This method returns a synchronized (thread-safe) sorted set backed by the specified sorted set.
unmodifiableCollection(Collection<? extends T> c)	This method returns an unmodifiable view of the specified collection.
unmodifiableList(List<? extends T> list)	This method returns an unmodifiable view of the specified list.
unmodifiableNavigableMap(Navigable Map<K,? extends V> m)	This method returns an unmodifiable view of the specified navigable map.
unmodifiableNavigableSet(NavigableS et<T> s)	This method returns an unmodifiable view of the specified navigable set.
unmodifiableSet(Set<? extends T> s)	This method returns an unmodifiable view of the specified set.
unmodifiableSortedMap(SortedMap<K ,? extends V> m)	This method returns an unmodifiable view of the specified sorted map.
unmodifiableSortedSet(SortedSet<T> s)	This method returns an unmodifiable view of the specified sorted set.

Now, we have listed all the methods. It is clear how important they are in writing optimized Java code. The Collections class is widely used and its methods appear in almost every optimized Java program. Here, we will implement these methods and also discuss their operations.

Java Collections Example

Examples of Collections Classes in Java are mentioned below:

1. Adding Elements to the Collections
2. Sorting a Collection
3. Searching in a Collection
4. Copying Elements
5. Disjoint Collection

1. Adding Elements to the Collections Class Object

The addAll() method of java.util.Collections class is used to add all the specified elements to the specified collection. Elements to be added may be specified individually or as an array.

Example:

```
// Adding Elements
// Using addAll() method

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Main {

    public static void main(String[] args) {

        List<String> l = new ArrayList<>();

        // Adding elements to the list
        l.add("Shoes");
        l.add("Toys");

        // Add one or more elements
        Collections.addAll(l, "Fruits", "Bat", "Ball");

        for (int i = 0; i < l.size(); i++) {
            System.out.print(l.get(i) + " ");
        }
    }
}

Output
Shoes Toys Fruits Bat Ball
```

2. Sorting a Collection

Collections.sort() is used to sort the elements present in the specified list of Collections in ascending order. Collections.reverseOrder() is used to sort in descending order.

Example:

```
// Sorting a Collections using sort() method
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Main {

    public static void main(String[] args) {
        List<String> l = new ArrayList<>();

        l.add("Shoes");
        l.add("Toys");


        Collections.addAll(l, "Fruits", "Bat", "Mouse");

        Collections.sort(l);

        for (int i = 0; i < l.size(); i++) {
            System.out.print(l.get(i) + " ");
        }

        System.out.println();

        Collections.sort(l, Collections.reverseOrder());

        for (int i = 0; i < l.size(); i++) {
            System.out.print(l.get(i) + " ");
        }
    }
}

Output
Bat Fruits Mouse Shoes Toys
Toys Shoes Mouse Fruits Bat
```

3. Searching in a Collection

Collections.binarySearch() method returns the position of an object in a sorted list. To use this method, the list should be sorted in ascending order, otherwise, the result returned from the method will be wrong. If the element exists in the list, the method will return the position of the element in the sorted list, if the element does not exist in the list then this method will return a negative number that shows where the item would be inserted in the list - 1.

Example:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Search {

    public static void main(String[] args) {

        List<String> l = new ArrayList<>();

        l.add("Shoes");
        l.add("Toys");
        l.add("Horse");
        l.add("Ball");
        l.add("Grapes");

        Collections.sort(l);

        System.out.println(
            "The index of Horse is: "
            + Collections.binarySearch(l, "Horse"));

        // BinarySearch on the List
        System.out.println(
            "The index of Dog is: "
            + Collections.binarySearch(l, "Dog"));
    }
}

Output
The index of Horse is: 2
The index of Dog is: -2
```

Note: The list must be sorted before using binarySearch to get the correct results.

4. Copying Elements

The copy() method of Collections class is used to copy all the elements from one list into another. After the operation, the index of each copied element in the destination list will be identical to its index in the source list. The destination list must be at least as long as the source list. If it is longer, the remaining elements in the destination list are unaffected.

Example:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Main {
    public static void main(String[] args) {
        List<String> l1 = new ArrayList<>();
        l1.add("Shoes");
        l1.add("Toys");
        l1.add("Horse");
        l1.add("Tiger");

        System.out.println("The Original Destination list is: ");

        for (int i = 0; i < l1.size(); i++) {
            System.out.print(l1.get(i) + " ");
        }
        System.out.println();

        List<String> l2 = new ArrayList<>();
        l2.add("Bat");
        l2.add("Frog");
        l2.add("Lion");
        Collections.copy(l1, l2);

        System.out.println("The Destination List After copying is: ");

        for (int i = 0; i < l1.size(); i++) {
            System.out.print(l1.get(i) + " ");
        }
    }
}

Output
The Original Destination list is:
Shoes Toys Horse Tiger
The Destination List After copying is:
Bat Frog Lion Tiger
```

5. Disjoint Collection

Collections.disjoint() is used to check whether two specified collections have nothing in common. It returns true if the two collections do not have any element in common.

Example:

```
// Working of Disjoint Function
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Main {

    public static void main(String[] args) {

        List<String> l1 = new ArrayList<>();

        // Add elements to l1
        l1.add("Shoes");
        l1.add("Toys");
        l1.add("Horse");
        l1.add("Tiger");

        List<String> l2 = new ArrayList<>();

        // Add elements to l2
        l2.add("Bat");
        l2.add("Frog");
        l2.add("Lion");

        // Check if disjoint or not
        System.out.println(
            Collections.disjoint(l1, l2));
    }
}

Output
true
```

Collection Interface in Java

The Collection interface in Java is a core member of the Java Collections Framework located in the java.util package. It is one of the root interfaces of the Java Collection Hierarchy. The Collection interface is not directly implemented by

any class. Instead, it is implemented indirectly through its sub-interfaces like List, Queue, and Set.

For Example, the ArrayList class implements the List interface, a sub-interface of the Collection interface.

Example:

```
import java.util.*;

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

        // Create a Collection using ArrayList
        Collection<String> c = new ArrayList<>();

        // Adding elements to the collection
        c.add("Apple");
        c.add("Banana");
        c.add("Orange");

        System.out.println("Collection: " + c);
    }
}

Output
Collection: [Apple, Banana, Orange]
```

Collection Interface Declaration

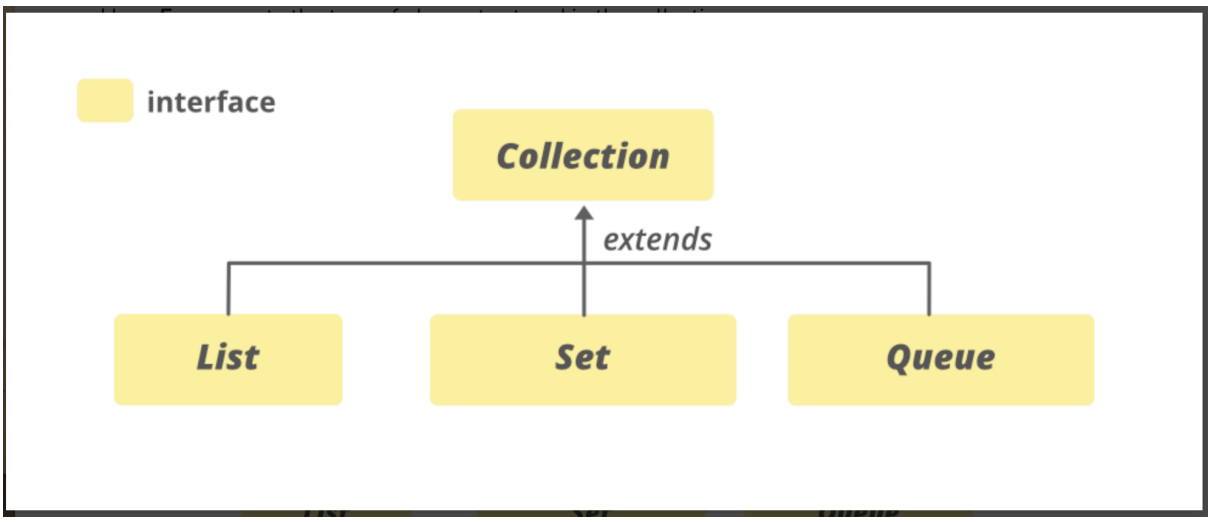
```
public interface Collection<E> extends Iterable<E>
```

Here, E represents the type of elements stored in the collection.

Note: In the above syntax, we can replace any class with ArrayList if that class implements the Collection interface.

Hierarchy of Collection Interface

The Collection interface is part of a hierarchy that extends Iterable, meaning collections can be traversed.



Collection Interface in Java-with-Examples
The hierarchy also includes several key sub-interfaces:

- Iterable
- Collection
  - List
  - Set
  - Queue
  - Deque
  - SortedSet
  - NavigableSet

Sub-Interfaces of Collection Interface

The subInterfaces are sometimes called Collection Types or SubTypes of Collection. These include the following:

1. List

The List interface represents an ordered collection that allows duplicates. It is implemented by classes like ArrayList, LinkedList, and Vector. Lists allow elements to be accessed by their index position.

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

2. Set

A set is an unordered collection of objects in which duplicate values cannot be stored. This set interface is implemented by various classes like HashSet, TreeSet, LinkedHashSet, etc.

```
public interface Set<E> extends Collection<E>
```

3. 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. The class which implements this interface is TreeSet.

```
public interface SortedSet<E> extends Set<E>
```

4. Queue

The Queue interface represents a collection that follows FIFO (First-In-First-Out) order. It is implemented by classes like PriorityQueue, Deque, ArrayDeque, etc.

```
public interface Queue<E> extends Collection<E>
```

5. Deque

The Deque interface extends Queue and allows elements to be added or removed from both ends of the queue. It is implemented by ArrayDeque and LinkedList.

```
public interface Deque<E> extends Queue<E>
```

6. NavigableSet

The NavigableSet interface extends SortedSet and provides additional methods for navigation such as finding the closest element.

```
public interface NavigableSet<E> extends SortedSet<E>
```

Implementing Classes

- AbstractCollection
- AbstractList
- AbstractQueue
- AbstractSequentialList
- AbstractSet
- ArrayBlockingQueue
- ArrayDeque
- ArrayList
- ConcurrentLinkedDeque
- ConcurrentLinkedQueue
- ConcurrentSkipListSet
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- DelayQueue
- EnumSet
- HashSet
- LinkedBlockingDeque
- LinkedBlockingQueue
- LinkedHashSet
- LinkedList
- LinkedTransferQueue
- PriorityBlockingQueue
- PriorityQueue
- Stack
- TreeSet
- Vector

Note: The Collection Interface is not limited to the above classes, there are many more classes.

Methods of Collection Interface

Method	Description
add(E e)	Ensures that this collection contains the specified element (optional operation).
addAll(Collection<? extends E> c)	Adds all the elements in the specified collection to this collection (optional operation).
clear()	Removes all the elements from this collection (optional operation).
contains(Object o) Returns true if this collection contains the specified element.	
containsAll(Collection<?> c)	Returns true if this collection contains all the elements in the specified collection.
equals(Object o)	Compare the specified object with this collection for equality.
hashCode()	Returns the hash code value for this collection.
isEmpty()	Returns true if this collection contains no elements.
iterator()	Returns an iterator over the elements in this collection.
parallelStream()	Returns a possibly parallel Stream with this collection as its source.
remove(Object o)	Removes a single instance of the specified element from this collection, if it is present (optional operation).
removeAll(Collection<?> c)	Removes all of this collection's elements that are also contained in the specified collection (optional operation).
removeIf(Predicate<? super E> filter)	Removes all the elements of this collection that satisfy the given predicate.
retainAll(Collection<?> c)	Retains only the elements in this

	collection that are contained in the specified collection (optional operation).
size()	Returns the number of elements in this collection.
spliterator()	Creates a Spliterator over the elements in this collection. stream() Returns a sequential Stream with this collection as its source.
toArray()	Returns an array containing all the elements in this collection.
toArray(IntFunction<T[]> generator)	Returns an array containing all the elements in this collection, using the provided generator function to allocate the returned array.
toArray(T[] a)	Returns an array containing all the elements in this collection; the runtime type of the returned array is that of the specified array.

Method Declared in Iterable Interface:

Method	Description
forEach(Consumer<? super T> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.

Java Collection Interface Examples

1. Adding Elements to a Collection

The add(E e) and addAll(Collection c) methods provided by Collection can be used to add elements.

```
// Adding elements to the Collection
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // create an empty ArrayList with an initial capacity
        Collection<Integer> l1 = new ArrayList<Integer>(5);

        // use add() method to add elements in the list
        l1.add(15);
        l1.add(20);
        l1.add(25);

        // prints all the elements available in list
        for (Integer n : l1) {
            System.out.println("Number = " + n);
        }

        // Creating another empty ArrayList
        Collection<Integer> l2 = new ArrayList<Integer>();

        // Appending the collection to the list
        l2.addAll(l1);

        // displaying the modified ArrayList
        System.out.println("The new ArrayList is: " + l2);
    }
}

Output
Number = 15
Number = 20
Number = 25
The new ArrayList is: [15, 20, 25]
```

2. Removing Elements from a Collections

The remove(E e) and removeAll(Collection c) methods can be used to remove a particular element or a Collection of elements from a collection.

```
// Removing elements from a Collection
import java.util.*;

public class Main {
    public static void main(String[] argv) throws Exception {

        // Creating object of HashSet
        Collection<Integer> hs1 = new HashSet<Integer>();

        hs1.add(1);
        hs1.add(2);
        hs1.add(3);
        hs1.add(4);
        hs1.add(5);

        System.out.println("Initial set: " + hs1);

        // remove a particular element
        hs1.remove(4);

        System.out.println("Set after removing 4: " + hs1);

        // Creating another object of HashSet
        Collection<Integer> hs2 = new HashSet<Integer>();
        hs2.add(1);
        hs2.add(2);
        hs2.add(3);

        System.out.println("Collection Elements to be removed: " + hs2);

        // Removing elements from hs1 specified in hs2
        // using removeAll() method
        hs1.removeAll(hs2);

        System.out.println("Set 1 after removeAll() operation: " + hs1);
    }
}

Output
Initial set: [1, 2, 3, 4, 5]
Set after removing 4: [1, 2, 3, 5]
Collection Elements to be removed: [1, 2, 3]
Set 1 after removeAll() operation: [5]
```

3. Iterating over a Collection

To iterate over the elements of Collection we can use the iterator() method.

```
// Iterating over a Collection
import java.util.*;

public class Main {

    public static void main(String[] args) {
        // Create and populate the list
        Collection<String> l = new LinkedList<>();

        l.add("Test");
        l.add("Test1");
        l.add("Test2");

        // Displaying the list
        System.out.println("The list is:" + l);

        // Create an iterator for the list
        // using iterator() method
        Iterator<String> it = l.iterator();

        // Displaying the values after iterating
        // through the list
        System.out.println("\nThe iterator values" + " of list are: ");
        while (it.hasNext()) {
            System.out.print(it.next() + " ");
        }
    }
}

Output
The list is:[Test, Test1, Test2]

The iterator values of list are:
Test Test1 Test2
```

Details of Java List Interface

The List Interface in Java extends the Collection Interface and is a part of the java.util package. It is used to store the ordered collections of elements. In a Java List, we can organize and manage the data sequentially.

Key Features:

- Maintained the order of elements in which they are added.
- Allows duplicate elements.
- The implementation classes of the List interface are ArrayList, LinkedList, Stack, and Vector.
- It can add Null values that depend on the implementation.
- The List interface offers methods to access elements by their index and includes the listIterator() method, which returns a ListIterator.
- Using ListIterator, we can traverse the list in both forward and backward directions.

Example:

```
class ListExample {
    public static void main(String[] args) {
        List<String> li = new ArrayList<>();
        li.add("Java");
        li.add("Python");
        li.add("DSA");
        li.add("C++");
        System.out.println("Elements of List are:");
        for (String s : li) {
            System.out.println(s);
        }
        System.out.println("Element at Index 1: "+ li.get(1));
        li.set(1, "JavaScript");
        System.out.println("Updated List: " + li);
        li.remove("C++");
        System.out.println("List After Removing Element: " + li);
    }
}
Output:
Elements of List are:
Java
Python
DSA
C++
Element at Index 1: Python
Updated List: [Java, JavaScript, DSA, C++]
List After Removing Element: [Java, JavaScript, DSA]
```

Declaration of Java List Interface

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

The common implementation classes of the List interface are **ArrayList**, **LinkedList**, **Stack**, and **Vector**:

- **ArrayList** and **LinkedList** are the most widely used due to their dynamic resizing and efficient performance for specific operations.
- **Stack** is a subclass of Vector, designed for Last-In-First-Out (LIFO) operations.
- **Vector** is considered a legacy class and is rarely used in modern Java programming. It is replaced by ArrayList and java.util.concurrent package.

Syntax of List Interface

```
List<Obj> list = new ArrayList<Obj> ();
```

Methods of the List Interface

Methods	Description
add(int index, element)	This method is used with Java List Interface to add an element at a particular index in the list. When a single parameter is passed, it simply adds the element at the end of the list.
addAll(int index, Collection collection)	This method is used with List Interface in Java to add all the elements in the given collection to the list. When a single parameter is passed, it adds all the elements of the given collection at the end of the list.
size()	This method is used with Java List Interface to return the size of the list.
clear()	This method is used to remove all the elements in the list. However, the reference of the list created is still stored.
remove(int index)	This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1.

remove(element)	This method is used with Java List Interface to remove the first occurrence of the given element in the list.
get(int index)	This method returns elements at the specified index.
set(int index, element)	This method replaces elements at a given index with the new element. This function returns the element which was just replaced by a new element.
indexOf(element)	This method returns the first occurrence of the given element or -1 if the element is not present in the list.
lastIndexOf(element)	This method returns the last occurrence of the given element or -1 if the element is not present in the list.
equals(element)	This method is used with Java List Interface to compare the equality of the given element with the elements of the list.
hashCode()	This method is used with List Interface in Java to return the hashcode value of the given list.
isEmpty()	This method is used with Java List Interface to check if the list is empty or not. It returns true if the list is empty, else false.
contains(element)	This method is used with List Interface in Java to check if the list contains the given element or not. It returns true if the list contains the element.
containsAll(Collection collection)	This method is used with Java List Interface to check if the list contains all the collection of elements.
sort(Comparator comp)	This method is used with List Interface in Java to sort the elements of the list on the basis of the given comparator.

Implementation classes of the List interface:

1. ArrayList
2. Vector
3. Stack
4. LinkedList

1.ArrayList:

Java ArrayList is a part of the collections framework and it is a class of java.util package. It provides us with dynamic-sized arrays in Java.

- The main advantage of ArrayList is that, unlike normal arrays, we don't need to mention the size when creating ArrayList. It automatically adjusts its capacity as elements are added or removed.
- It may be slower than standard arrays, but it is helpful when the size is not known in advance. Note that creating a large fixed-sized array would cause a waste of space.
- Since ArrayList is part of the collections framework, it has better interoperability with other collections. For example, conversion to a HashSet is straightforward.
- With generics, ArrayList<T> ensures type safety at compile-time.

Example:

```
// Java Program to demonstrate ArrayList
import java.util.ArrayList;

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

        // Creating an ArrayList
        ArrayList<Integer> a = new ArrayList<Integer>();

        // Adding Element in ArrayList
        a.add(1);
        a.add(2);
        a.add(3);

        // Printing ArrayList
        System.out.println(a);

    }
}

Output
[1, 2, 3]
```

Syntax of ArrayList

```
ArrayList<Integer> arr = new ArrayList<Integer>();
```

Note: You can also create a generic ArrayList

Important Features of ArrayList in Java

- ArrayList inherits AbstractList class and implements the List interface.
- ArrayList is initialized by size. However, the size 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 need a wrapper class for such cases.
- ArrayList in Java can be seen as a vector in C++.
- ArrayList is not Synchronized. Its equivalent synchronized class in Java is Vector.

Constructors in ArrayList in Java

In order to Create an ArrayList, we need to create an object of the ArrayList class. The ArrayList class consists of various constructors which allow the possible creation of the array list. The following are the constructors available in this class:

Constructor	**Description**	**Initialize and Declare ArrayList**
ArrayList()	This constructor is used to build an empty array list.	ArrayList arr = new ArrayList();
ArrayList(Collection c)	This constructor is used to build an array list initialized with the elements from the collection c.	ArrayList arr = new ArrayList(c);
ArrayList(int capacity)	This constructor is used to build an array list with the initial capacity being specified.	ArrayList arr = new ArrayList(N);

Java ArrayList Methods

Method	Description
add(int index, Object element)	This method is used to insert a specific element at a specific position index in a list.
add(Object o)	This method is used to append a specific element to the end of a list.
addAll(Collection C)	This method is used to append all the elements from a specific collection to the end of the mentioned list, in such an order that the values are returned by the specified collection's iterator.
addAll(int index, Collection C)	Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list.
clear()	This method is used to remove all the elements from any list.
clone()	This method is used to return a shallow copy of an ArrayList in Java.
contains(Object o)	Returns true if this list contains the specified element.
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.
forEach(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
get(int index)	Returns the element at the specified position in this list.
indexOf(Object O)	The index the first occurrence of a specific element is either returned or -1 in case the element is not in the list.
isEmpty()	Returns true if this list contains no elements.

lastIndexOf(Object O)	The index of the last occurrence of a specific element is either returned or -1 in case the element is not in the list.
listIterator()	Returns a list iterator over the elements in this list (in proper sequence).
listIterator(int index)	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
remove(int index)	Removes the element at the specified position in this list.
remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present.
removeAll(Collection c)	Removes from this list all of its elements that are contained in the specified collection.
removeIf(Predicate filter)	Removes all of the elements of this collection that satisfy the given predicate.
removeRange(int fromIndex, int toIndex)	Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
retainAll(Collection<?> c)	Retains only the elements in this list that are contained in the specified collection.
set(int index, E element)	Replaces the element at the specified position in this list with the specified element.
size()	Returns the number of elements in this list.
spliterator?()	Creates a late-binding and fail-fast Spliterator over the elements in this list.
subList(int fromIndex, int toIndex)	Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

toArray()	This method is used to return an array containing all of the elements in the list in the correct order.
toArray(Object[] O)	It is also used to return an array containing all of the elements in this list in the correct order, the same as the previous method.
trimToSize()	This method is used to trim the capacity of the instance of the ArrayList to the list's current size.

Operations in ArrayList

Now, Using the constructors we have got ArrayList for further operations like Insertion , Deletion and Updation of the elements in ArrayList.

```
class Main {
    public static void main(String args[]){
        ArrayList<String> al = new ArrayList<>();

        al.add("Test");
        al.add("Test1");
        System.out.println("Original List : " + al);

        // Adding Elements at the specific index
        al.add(1, "Test3");
        System.out.println("After Adding element at index 1 : "+ al);

        // 2. Deletion of Element
        al.remove(0);
        System.out.println("Element removed from index 0 : " + al);

        // Removing Element using the value
        al.remove("Test2");
        System.out.println("Element removed : " + al);

        // 3. Updating Values
        al.set(0, "Test");

        // Printing all the elements in an ArrayList
        System.out.println("List after updation of value : "+ al);
    }
}
Try it on GfG Practice
redirect icon

Output
Original List : [Test, Test1]
After Adding element at index 1 : [Test, Test3, Test2]
Element removed from index 0 : [Test3, Test2]
Element removed : [Test3]
List after updation of value : [Test, Test3]
```

Some Key Points of ArrayList in Java

- ArrayList is Underlined data Structure Resizable Array or Growable Array.
- ArrayList Duplicates Are Allowed.
- Insertion Order is Preserved.
- Heterogeneous objects are allowed.
- Null insertion is possible.

Complexity of Java ArrayList

Operation	Time Complexity
Inserting Element in ArrayList	O(1)
Removing Element from ArrayList	O(N)
Traversing Elements in ArrayList	O(N)
Replacing Elements in ArrayList	O(1)

2. LinkedList:

Linked List is a part of the Collection framework present in java.util package. This 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 an address part. The elements are linked using pointers and addresses, and each element is known as a node.

Example:

```
import java.util.LinkedList;
public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> l = new LinkedList<String>();
        l.add("One");
        l.add("Two");
        l.add("Three");
        l.add("Four");
        l.add("Five");
        System.out.println(l);
    }
}
Output
[One, Two, Three, Four, Five]
```

Note: The nodes cannot be accessed directly instead we have to start from the head and follow the link until we find the node that we want.

Internal Working of LinkedList
Since a LinkedList acts as a dynamic array and we do not have to specify the size while creating it, the size of the list automatically increases when we dynamically add and remove items. And also, the elements are not stored in a continuous fashion. Therefore, there is no need to increase the size. Internally, the LinkedList is implemented using the doubly linked list data structure.

Normal List vs Doubly LinkedList:
The main difference between a normal linked list and a doubly LinkedList is that Doubly linked list contains an extra pointer which is known as the previous pointer. In this each node points to both the next and previous nodes.

Constructors in the LinkedList
In order to create a LinkedList, we need to create an object of the LinkedList class. The LinkedList class consists of various constructors that allow the possible creation of the list. The following are the constructors available in this class:

1. LinkedList(): This constructor is used to create an empty linked list. If we wish to create an empty LinkedList with the name ll, then it can be created as:
```
LinkedList ll = new LinkedList();
```

2. LinkedList(Collection C): This constructor is used to create an ordered list that contains all the elements of a specified collection, as returned by the collection's iterator. If we wish to create a LinkedList with the name ll, then, it can be created as:
```
LinkedList ll = new LinkedList(C);
```

Methods for Java LinkedList

Method	Description
add(int index, E element)	This method inserts the specified element at the specified position in this list.
add(E e)	This method appends the specified element to the end of this list.
addAll(int index, Collection<E> c)	This method inserts all of the elements in the specified collection into this list, starting at the specified position.
addAll(Collection<E> c)	This method 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.
addFirst(E e)	This method inserts the specified element at the beginning of this list.
addLast(E e)	This method appends the specified element to the end of this list.
clear()	This method removes all of the elements from this list.
clone()	This method returns a shallow copy of this LinkedList.
contains(Object o)	This method returns true if this list contains the specified element.
descendingIterator()	This method returns an iterator over the elements in this deque in reverse sequential order.
element()	This method retrieves but does not remove the head (first element) of this list.
get(int index)	This method returns the element at the specified position in this list.
getFirst()	This method returns the first element in this list.
getLast()	This method returns the last element in this list.
indexOf(Object o)	This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
lastIndexOf(Object o)	This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
listIterator(int index)	This method returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
offer(E e)	This method adds the specified element as the tail (last element) of this list.

offerFirst(E e)	This method inserts the specified element at the front of this list.
offerLast(E e)	This method inserts the specified element at the end of this list.
peek()	This method retrieves but does not remove the head (first element) of this list.
peekFirst()	This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty.
peekLast()	This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.
poll()	This method retrieves and removes the head (first element) of this list.
pollFirst()	This method retrieves and removes the first element of this list, or returns null if this list is empty.
pollLast()	This method retrieves and removes the last element of this list, or returns null if this list is empty.
pop()	This method pops an element from the stack represented by this list.
push(E e)	This method pushes an element onto the stack represented by this list.
remove()	This method retrieves and removes the head (first element) of this list.
remove(int index)	This method removes the element at the specified position in this list.
remove(Object o)	This method removes the first occurrence of the specified element from this list if it is present.
removeFirst()	This method removes and returns the first element from this list.
removeFirstOccurrence(Object o)	This method removes the first occurrence of the specified element in this list (when traversing the list from head to tail).

removeLast()	This method removes and returns the last element from this list.
removeLastOccurrence(Object o)	This method removes the last occurrence of the specified element in this list (when traversing the list from head to tail).
set(int index, E element)	This method replaces the element at the specified position in this list with the specified element.
size()	This method returns the number of elements in this list.
spliterator()	This method creates a late-binding and fail-fast Spliterator over the elements in this list.
toArray()	This method returns an array containing all of the elements in this list in proper sequence (from first to last element).
toArray(T[] a)	This method returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
toString()	This method returns a string containing all of the elements in this list in proper sequence (from first to the last element), each element is separated by commas and the String is enclosed in square brackets.

Performing Different Operations on LinkedList

1. Adding elements
2. Updating elements
3. Removing elements
4. Iterating over elements
5. To Array()
6. Size();
7. removeFirst()
8. removeLast()

1. Adding Elements: With the help of add() method, we can add elements to an ArrayList. This method can perform multiple operations based on different parameters. They are:

- **add(Object):** This method is used to add an element at the end of the LinkedList.
- **add(int index, Object):** This method is used to add an element at a specific index in the LinkedList.

Example:

```
import java.util.*;

public class Main {

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

        ll.add("Test");
        ll.add("Test3");
        ll.add(1, "Test2");

        System.out.println(ll);
    }
}

Output
[Test, Test2, Test3]
```

2. Changing Elements: With the of set() method, we can change elements in a LinkedList. This method takes an index and the updated element which needs to be inserted at that index.

Example:

```
import java.util.*;

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

        ll.add("Test");
        ll.add("Test3");
        ll.add(1, "test4");

        System.out.println("Initial LinkedList " + ll);
```

```
        ll.set(1, "Test2");
        System.out.println("Updated LinkedList " + ll);
    }
}

Output
Initial LinkedList [Test, Test4, Test3]
Updated LinkedList [Test, Test2, Test3]
```

3. Removing Elements: With the help of remove() method we can remove elements from a LinkedList. This method can perform multiple operations based on different parameters. They are:

- remove(Object): This method is used to remove an object from the LinkedList. If there are multiple objects, then the first occurrence of the object is removed.
- remove(int index): This method removes the element at the specific index in the LinkedList. After removing, the list updates so the elements shift, and the LinkedList reflects the change.

Example:

```
import java.util.*;

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

        ll.add("Test");
        ll.add("Test3");
        ll.add(1, "Test2");

        System.out.println("Initial LinkedList " + ll);
        ll.remove(1);

        System.out.println("After the Index Removal " + ll);

        ll.remove("Test3");
        System.out.println("After the Object Removal " + ll);
    }
}

Output
Initial LinkedList [Test, Test2, Test3]
After the Index Removal [Test, Test3]
After the Object Removal [Test]
```

4. Iterating the LinkedList: There are multiple ways to iterate through LinkedList. The most famous ways are by using the basic for-loop in combination with a get() method to get the element at a specific index and the advanced for-loop.

Example:

```
import java.util.*;

public class Main {

    public static void main(String args[]) {
        LinkedList<String> ll = new LinkedList<>();
        ll.add("Test");
        ll.add("Test3");
        ll.add(1, "Test2");

        for (int i = 0; i < ll.size(); i++) {
            System.out.print(ll.get(i) + " ");
        }

        System.out.println();
        for (String str : ll)
            System.out.print(str + " ");
    }
}

Output
Test, Test2, Test3
Test, Test2, Test3
```

5. Linked list to Array by using toArray(): It converts the elements of the linked list into a new array.

Example:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        LinkedList<Integer> list= new LinkedList<Integer>();
        list.add(123);
        list.add(12);
        list.add(11);
        list.add(1134);
        System.out.println("LinkedList: "+ list);
        Object[] a = list.toArray();
        System.out.print("After converted LinkedList to Array: ");
        for(Object element : a)
        System.out.print(element+" ");
    }
}

Output
LinkedList: [123, 12, 11, 1134]
After converted LinkedList to Array: 123 12 11 1134
```

6. size(): This method returns the total number of elements in the linked list.

Example:

```
import java.util.LinkedList;

public class Main {
    public static void main(String args[]) {
        LinkedList<String> list = new LinkedList<String>();
        list.add("Test");
        list.add("Test1");

        System.out.println("The size is: " + list.size());
    }
}

Output
The size is: 2
```

7. removeFirst(): This method returns the first element of a Linkedlist.

Example:

```
import java.util.LinkedList;

public class Main {
    public static void main(String args[]) {
        LinkedList<Integer> list = new LinkedList<Integer>();
        list.add(10);
        list.add(20);
        list.add(30);
        System.out.println("LinkedList: " + list);
        System.out.println("The  remove first element is: " +
list.removeFirst());
        // Displaying the final list
        System.out.println("Final LinkedList:" + list);
    }
}

Output
LinkedList:[10, 20, 30]
The  remove first element is: 10
Final LinkedList:[20, 30]
```

8. removelast(): This method removes the last element from the Linkedlist.

Example:

```
import java.util.LinkedList;

public class Main {
    public static void main(String args[]) {
        LinkedList<Integer> list = new LinkedList<Integer>();
        list.add(10);
        list.add(20);
        list.add(30);
        System.out.println("LinkedList:" + list);
        // Remove the tail using removeLast()
        System.out.println("The last element is removed: " +
list.removeLast());
        // Displaying the final list
        System.out.println("Final LinkedList:" + list);
        // Remove the tail using removeLast()
        System.out.println("The last element is removed: " +
list.removeLast());
```

```
        // Displaying the final list
        System.out.println("Final LinkedList:" + list);
    }
}

Output
LinkedList:[10, 20, 30]
The last element is removed: 30
Final LinkedList:[10, 20]
The last element is removed: 20
Final LinkedList:[10]
```

Here is a simple example that demonstrates how to use a LinkedList in Java:

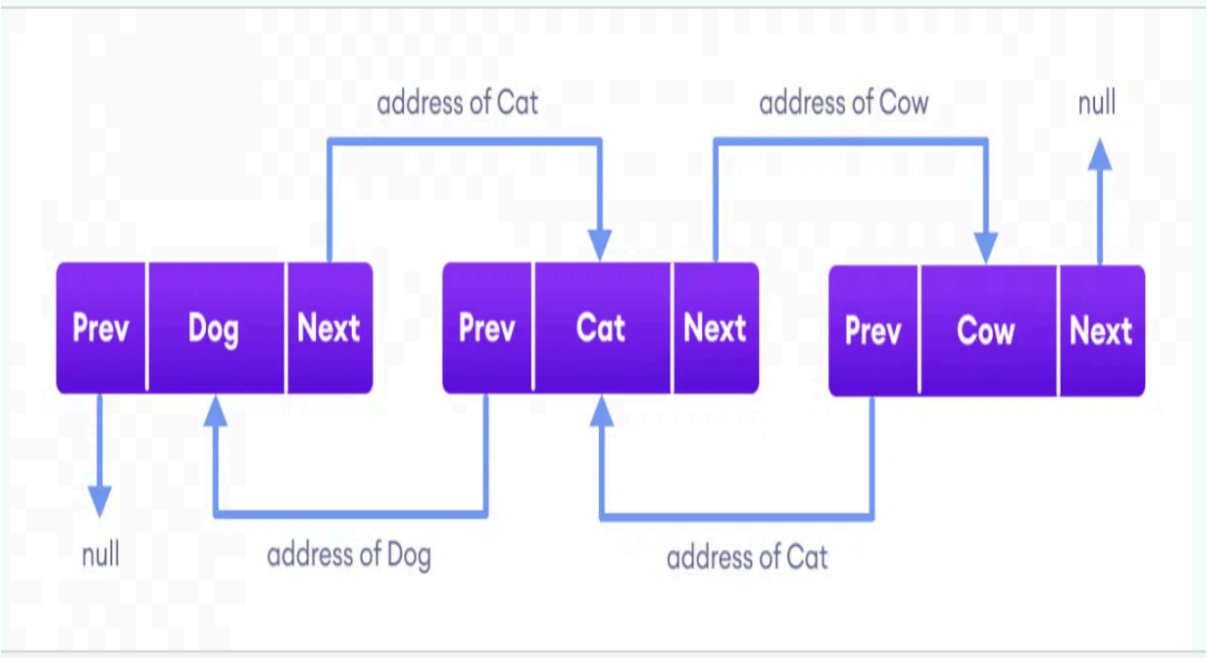
```
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        // Create a new linked list
        LinkedList<Integer> linkedList = new LinkedList<>();

        // Add elements to the linked list
        linkedList.add(1);
        linkedList.add(2);
        linkedList.add(3);

        // Add an element to the beginning of the linked list
        linkedList.addFirst(0);

        // Add an element to the end of the linked list
        linkedList.addLast(4);

        // Print the elements of the linked list
        for (int i : linkedList) {
            System.out.println(i);
        }
    }
}

Output
0
1
2
3
4
```

Advantages

- The linked list can increase or decrease its size whenever we need, so there is no need to set the size of a Linkedlist before using it.
- Adding or removing elements in the middle of a Linkedlist is a very simple task. We just need to change the links between nodes without moving other elements.
- In Linkedlist we can move in forward and backward direction because each element knows about the one before and after it.

Disadvantages

- Finding an element in a linked list takes more time because we have to go through the list from the start.
- In Linkedlist each element stores extra information that's why it takes more memory.



3. Vector

The Underlying Data Structure is Resizable Array ORGrowable Array.

- Insertion Order is Preserved.
- Duplicate Objects are allowed.
- Heterogeneous Objects are allowed.
- null Insertion is Possible.
- Implements Serializable, Cloneable and RandomAccess interfaces.
- Every Method Present Inside Vector is Synchronized and Hence Vector Object is
- Thread Safe.
- Vector is the Best Choice if Our Frequent Operation is Retrieval.

Constructors:

Constructor	**Description**
Vector v = new Vector();	Creates an Empty Vector Object with Default Initial Capacity 10. Once Vector Reaches its Max Capacity then a New Vector Object will be Created with **New Capacity** = **Current Capacity * 2**
Vector v = new Vector(intinitialCapacity);	Here we can define initial capacity of the vector
Vector v = new Vector(intinitialCapacity, intincrementalCapacity);	Here we can define the initial capacity of the vector as well as growing policy.
Vector v = new Vector(Collection c);	Can accept a collection to create a vector.

Methods:

1) To Add Elements:

- add(Object o)Collection
- add(int index, Object o)List
- addElement(Object o) Vector

2) To Remove Elements:

- remove(Object o) Collection
- removeElement(Object o)Vector
- remove(int index) List
- removeElementAt(int index)Vector
- clear() Collection
- removeAllElements()Vector

3) To Retrieve Elements:

- Object get(int index)List
- Object elementAt(int index)Vector
- Object firstElement() Vector
- Object lastElement()Vector

4) Some Other Methods:

- int size()
- int capacity()
- Enumeration element()

Example:

```
import java.util.Vector;
class VectorDemo {
    public static void main(String[] args) {
        Vector v = new Vector();
        System.out.println(v.capacity()); //10
        for(int i = 1; i<=10; i++) {
             v.addElement(i);
        }

        System.out.println(v.capacity()); //10
        v.addElement("A");
        System.out.println(v.capacity()); //20
        System.out.println(v); //[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A]
    }
}
```

5. Stack:

- It is the Child Class of Vector.
- It is a Specially Designed Class for Last In First Out (LIFO) Order.

Constructor:

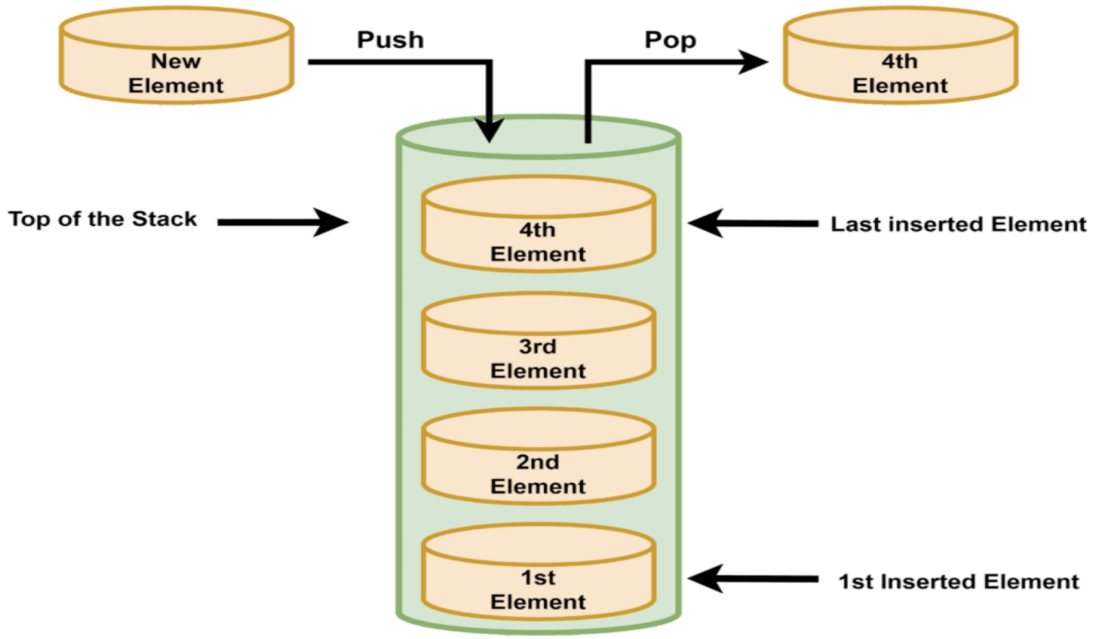
```
Stack s = new Stack();
```

Methods:

1. **Object push(Object o):** To insert an Object into the Stack.
2. **Object pop():** To Remove and Return Top of the Stack.
3. **Object peek():** To Return Top of the Stack without Removal.
4. **boolean empty()**: Returns true if Stack is Empty
5. **int search(Object o):** Returns Offset if the Element is Available Otherwise Returns -1.

Example:

```
import java.util.Stack;
class StackDemo {
    public static void main(String[] args) {
        Stack s = new Stack();
        s.push("A");
        s.push("B");
        s.push("C");
        System.out.println(s); //[A, B, C]
        System.out.println(s.search("A")); //3
        System.out.println(s.search("Z")); //-1
    }
}
```



Details of Set Interface (SortedSet(I), TreeSet, Hashset, LinkedHashSet)

The Set Interface is present in the java.util package and extends the Collection interface. It is an unordered collection of objects in which duplicate values cannot be stored. It is an interface that implements the mathematical set. This interface adds a feature that restricts the insertion of duplicate elements.

- No Specific Order: Does not maintain any specific order of elements (Exceptions: LinkedHashSet and TreeSet).
- Allows One Null Element: Most Set implementations allow a single null element.
- Implementation Classes: HashSet, LinkedHashSet, and TreeSet.
- Thread-Safe Alternatives: For thread-safe operations, use ConcurrentSkipListSet or wrap a set using Collections.synchronizedSet().

Two interfaces extend the set implementation, that is, SortedSet and NavigableSet.

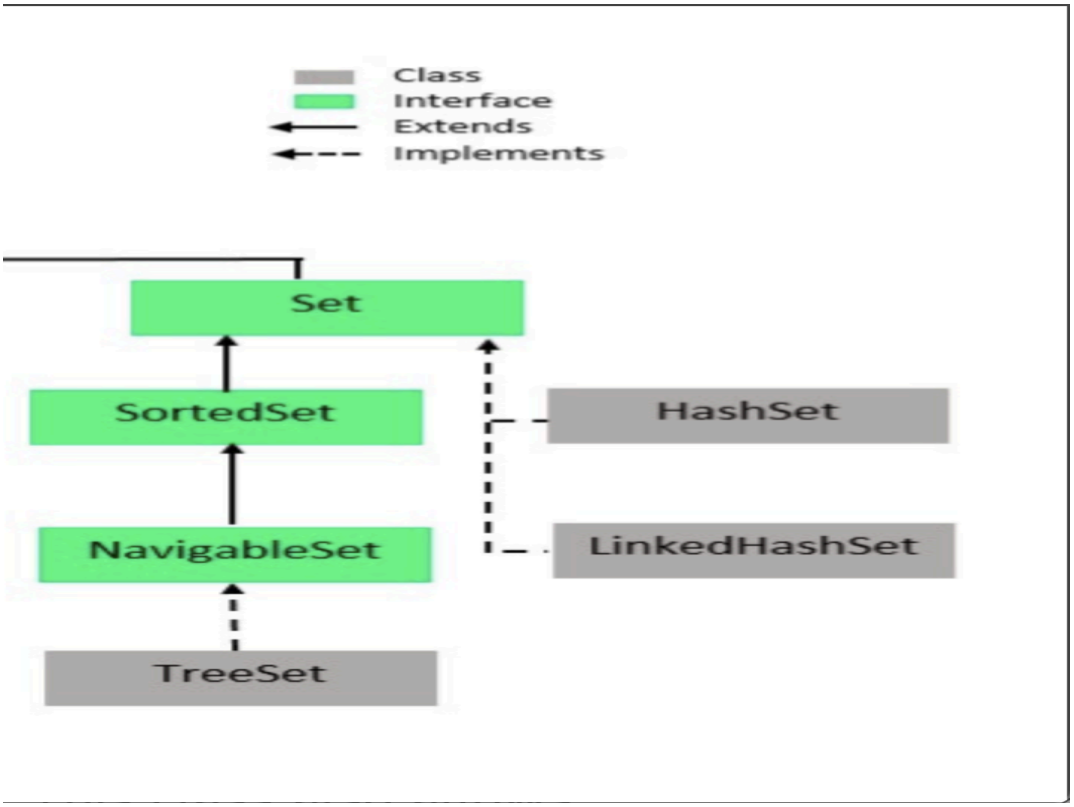
Example: This example demonstrates how to create an empty HashSet.

```
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String args[]) {
        // Create a Set using HashSet
        Set<String> s = new HashSet<>();

        // Displaying the Set
        System.out.println("Set Elements: " + s);
    }
}

Output
Set Elements: []
```

Explanation: In the above example, HashSet will appear as an empty set, as no elements were added. The order of elements in HashSet is not guaranteed, so the elements will be displayed in a random order if any are added.

Hierarchy of Java Set interface

The image below demonstrates the hierarchy of Java Set interface.



In the above image, the navigable set extends the sorted set interface. Since a set doesn't retain the insertion order, the navigable set interface provides the implementation to navigate through the Set. The class which implements the navigable set is a TreeSet which is an implementation of a self-balancing tree. Therefore, this interface provides us with a way to navigate through this tree.

Declaration of Set Interface

The declaration of Set interface is listed below:

```
public interface Set extends Collection
```

Creating Set Objects:

Since Set is an interface, objects cannot be created from the typeset. We always need a class that extends this list in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Set. This type-safe set can be defined as:

```
// Obj is the type of the object to be stored in Set
Set<Obj> set = new HashSet<Obj> ();
```

Methods

Let us discuss methods present in the Set interface provided below in a tabular format below as follows:

Method	Description
add(element)	This method is used to add a specific element to the set. The function adds the element only if the specified element is not already present in the set else the function returns False if the element is already present in the Set.
addAll(collection)	This method is used to append all of the elements from the mentioned collection to the existing set. The elements are added randomly without following any specific order.
clear()	This method is used to remove all the elements from the set but not delete the set. The reference for the set still exists.
contains(element)	This method is used to check whether a specific element is present in the Set or not.
containsAll(collection)	This method is used to check whether the set contains all the elements present in the given collection or not. This method returns true if the set contains all the elements and returns false if any of the elements are missing.
hashCode()	This method is used to get the hashCode value for this instance of the Set. It returns an integer value which is the hashCode value for this instance of the Set.
isEmpty()	This method is used to check whether the set is empty or not.
iterator()	This method is used to return the iterator of the set. The elements from the set are returned in a random order.

remove(element)	This method is used to remove the given element from the set. This method returns True if the specified element is present in the Set otherwise it returns False.
removeAll(collection)	This method is used to remove all the elements from the collection which are present in the set. This method returns true if this set changed as a result of the call.
retainAll(collection)	This method is used to retain all the elements from the set which are mentioned in the given collection. This method returns true if this set changed as a result of the call.
size()	This method is used to get the size of the set. This returns an integer value which signifies the number of elements.
toArray()	This method is used to form an array of the same elements as that of the Set.

Example: Sample Program to Illustrate Set interface

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        s.add("Test");
        s.add("Test1");
        s.add("Test2");
        s.add("Test2");
        s.add("Test3");
        System.out.println(s);
    }
}

Output
[Test, Test1, Test2, Test3]
```

Explanation: In the above example, we can see the elements are in random order as HashSet does not maintain any order and duplicate elements are automatically ignored in HashSet.

Operations on the Set Interface
The set interface allows the users to perform the basic mathematical operation on the set. Let's take two arrays to understand these basic operations.
Let set1 = [1, 3, 2, 4, 8, 9, 0] and set2 = [1, 3, 7, 5, 4, 0, 7, 5]. Then the possible operations on the sets are:

1. Intersection: This operation returns all the common elements from the given two sets. For the above two sets, the intersection would be:

Intersection = [0, 1, 3, 4]

2. Union: This operation adds all the elements in one set with the other. For the above two sets, the union would be:

Union = [0, 1, 2, 3, 4, 5, 7, 8, 9]

3. Difference: This operation removes all the values present in one set from the other set. For the above two sets, the difference would be:

Difference = [2, 8, 9]

Now, let us implement the following operations as defined above as follows:
Example: This example demonstrates how to perform various operations on set.

```
import java.util.*;

public class Main {
    public static void main(String args[]) {
        Set<Integer> a = new HashSet<Integer>();
        a.addAll(Arrays.asList(new Integer[] { 1, 3, 2, 4, 8, 9, 0 }));

        Set<Integer> b = new HashSet<Integer>();
        b.addAll(
             Arrays.asList(new Integer[] { 1, 3, 7, 5, 4, 0, 7, 5 }));

        Set<Integer> u = new HashSet<Integer>(a);
        u.addAll(b);
        System.out.print("Union of the two Set");
        System.out.println(u);
n
        Set<Integer> i = new HashSet<Integer>(a);
        i.retainAll(b);
        System.out.print("Intersection of the two Set");
        System.out.println(i);
```

```
        // To find the symmetric difference
        Set<Integer> d = new HashSet<Integer>(a);
        d.removeAll(b);
        System.out.print("Difference of the two Set");
        System.out.println(d);
    }
}

Output:
Union of the two Set[0, 1, 2, 3, 4, 5, 7, 8, 9]
Intersection of the two Set[0, 1, 3, 4]
Difference of the two Set[2, 8, 9]
```

SortedSet:

- It is the Child Interface of Set.
- If we want to Represent a Group of Individual Objects without Duplicates and all objects will be Inserted According to Some Sorting Order, then we should go for SortedSet.
- The Sorting can be Either Default Natural Sorting OR Customized Sorting Order.
- For String Objects Default Natural Sorting is Alphabetical Order.
- For Numbers Default Natural Sorting is Ascending Order.

Methods

Method	Description
Comparator comparator()	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned.
Object first()	Returns the first element in the invoking sorted set.
SortedSet headSet(Object end)	Returns a SortedSet containing those elements less than end that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
Object last()	Returns the last element in the invoking sorted set.

SortedSet subSet(Object start, Object end)	Returns a SortedSet that includes those elements between start and end.1. Elements in the returned collection are also referenced by the invoking object.
SortedSet tailSet(Object start)	Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

TreeSet:

TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree(red - black 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.

- TreeSet does not allow duplicate elements. Any attempt to add a duplicate element will be ignored.
- It doesn't allow null values and throws a NullPointerException null element is inserted in it.
- TreeSet implements the NavigableSet interface and provides additional methods to navigate the set (e.g., higher(), lower(), ceiling(), and floor()).
- It is not thread safe. For concurrent access, it should be synchronized externally using Collections.synchronizedSet().

It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. The TreeSet implements a NavigableSet interface by inheriting the AbstractSet class.

Example:

```java
// Java Program Implementing TreeSet
import java.util.TreeSet;

public class TreeSetCreation {
    public static void main(String args[]) {
        // Create a TreeSet of Strings
        TreeSet<String> t = new TreeSet<>();

        // Displaying the TreeSet (which is empty at this point)
        System.out.println("TreeSet elements: " + t);
    }
}

Output
TreeSet elements: []
```

Internal Working of TreeSet in Java

TreeSet is basically an implementation of a self-balancing binary search tree like a Red-Black Tree. Therefore operations like add, remove, and search takes O(log(N)) time. The reason is that in a self-balancing tree, it is made sure that the height of the tree is always O(log(N)) for all the operations. Therefore, this is considered as one of the most efficient data structures in order to store the huge sorted data and perform operations on it. However, operations like printing N elements in the sorted order take O(N) time.

Now let us discuss Synchronized TreeSet prior moving ahead. The implementation of a TreeSet is not synchronized. This means that if multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSortedSet method. This is best done at the creation time, to prevent accidental unsynchronized access to the set. It can be achieved as shown below as follows:

```java
TreeSet ts = new TreeSet();
Set syncSet = Collections.synchronziedSet(ts);
```

Constructors of TreeSet Class are as follows:

In order to create a TreeSet, we need to create an object of the TreeSet class. The TreeSet class consists of various constructors which allow the possible creation of the TreeSet. The following are the constructors available in this class:

1. TreeSet(): This constructor is used to build an empty TreeSet object in which elements will get stored in default natural sorting order.

Syntax: If we wish to create an empty TreeSet with the name ts, then, it can be created as:

```
TreeSet ts = new TreeSet();
```

2. TreeSet(Comparator): This constructor is used to build an empty TreeSet object in which elements will need an external specification of the sorting order.

Syntax: If we wish to create an empty TreeSet with the name ts with an external sorting phenomenon, then, it can be created as:

```
TreeSet ts = new TreeSet(Comparator comp);
```

3. TreeSet(Collection): This constructor is used to build a TreeSet object containing all the elements from the given collection in which elements will get stored in default natural sorting order. In short, this constructor is used when any conversion is needed from any Collection object to TreeSet object.

Syntax: If we wish to create a TreeSet with the name ts, then, it can be created as follows:

```
TreeSet t = new TreeSet(Collection col);
```

4. TreeSet(SortedSet): This constructor is used to build a TreeSet object containing all the elements from the given sortedset in which elements will get stored in default natural sorting order. In short, this constructor is used to convert the SortedSet object to the TreeSet object.

Syntax: If we wish to create a TreeSet with the name ts, then, it can be created as follows:

```
TreeSet t = new TreeSet(SortedSet s);
```

Methods in TreeSet Class

Methods in TreeSet Class are depicted below in tabular format which later on we will be implementing to showcase in the implementation part.

TreeSet implements SortedSet so it has the availability of all methods in Collection, Set, and SortedSet interfaces. Following are the methods in the Treeset interface. In the table below, the "?" signifies that the method works with any type of object including user-defined objects.

Method	Description
add(Object o)	This method will add the specified element according to the same sorting order mentioned during the creation of the TreeSet. Duplicate entries will not get added.
addAll(Collection c)	This method will add all elements of the specified Collection to the set. Elements in the Collection should be homogeneous otherwise ClassCastException will be thrown. Duplicate Entries of Collection will not be added to TreeSet.
ceiling(E e)	This method returns the least element in this set greater than or equal to the given element, or null if there is no such element.
clear()	This method will remove all the elements.
clone()	The method is used to return a shallow copy of the set, which is just a simple copied set.
Comparator comparator()	This method will return the Comparator used to sort elements in TreeSet or it will return null if the default natural sorting order is used.
contains(Object o)	This method will return true if a given element is present in TreeSet else it will return false.
descendingIterator()	This method returns an iterator over the elements in this set in descending order.
descendingSet()	This method returns a reverse order

	view of the elements contained in this set.
first()	This method will return the first element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
floor(E e)	This method returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
headSet(Object toElement)	This method will return elements of TreeSet which are less than the specified element.
higher(E e)	This method returns the least element in this set strictly greater than the given element, or null if there is no such element.
isEmpty()	This method is used to return true if this set contains no elements or is empty and false for the opposite case.
Iterator iterator()	Returns an iterator for iterating over the elements of the set.
last()	This method will return the last element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
lower(E e)	This method returns the greatest element in this set strictly less than the given element, or null if there is no such element.
pollFirst()	This method retrieves and removes the first (lowest) element, or returns null if this set is empty.
pollLast()	This method retrieves and removes the last (highest) element, or returns null if this set is empty.
remove(Object o)	This method is used to return a specific element from the set.
size()	This method is used to return the size

	of the set or the number of elements present in the set.
spliterator()	This method creates a late-binding and fail-fast Spliterator over the elements in this set.
subSet(Object fromElement, Object toElement)	This method will return elements ranging from fromElement to toElement. fromElement is inclusive and toElement is exclusive.
tailSet(Object fromElement)	This method will return elements of TreeSet which are greater than or equal to the specified element.

Various Operations over TreeSet in Java

Here we will be performing various operations over the TreeSet object to get familiar with the methods and concepts of TreeSet in java. Let's see how to perform a few frequently used operations on the TreeSet. They are listed as follows:

- Adding elements
- Accessing elements
- Removing elements
- Iterating through elements

Now let us discuss each operation individually one by one later alongside grasping with the help of a clean java program.

Operation 1: Adding Elements

In order to add an element to the TreeSet, we can use the add() method. However, the insertion order is not retained in the TreeSet. Internally, for every element, the values are compared and sorted in ascending order. We need to keep a note that duplicate elements are not allowed and all the duplicate elements are ignored. And also, Null values are not accepted by the TreeSet.

Example:

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Set<String> ts = new TreeSet<>();
        ts.add("Test");
        ts.add("Test1");
        ts.add("Test1");

        System.out.println(ts);
    }
}

Output
[Test, Test1]
```

Operation 2: Accessing the Elements

After adding the elements, if we wish to access the elements, we can use inbuilt methods like contains(), first(), last(), etc.

Example:

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        NavigableSet<String> ts = new TreeSet<>();

        ts.add("Test");
        ts.add("Test1");
        ts.add("Test1");
        ts.add("Test2");

        System.out.println("Tree Set is " + ts);

        String check = "Test1";

        System.out.println("Contains " + check + " "
                           + ts.contains(check));

        System.out.println("First Value " + ts.first());

        System.out.println("Last Value " + ts.last());

        String val = "Test1";
```

```
        System.out.println("Higher " + ts.higher(val));
        System.out.println("Lower " + ts.lower(val));
    }
}

Output
Tree Set is [Test, Test1, Test2]
Contains Test1 true
First Value Test
Last Value Test2
Higher Test2
Lower Test
```

Operation 3: Removing the Values

The values can be removed from the TreeSet using the remove() method. There are various other methods that are used to remove the first value or the last value.

Example:

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        NavigableSet<String> ts = new TreeSet<>();

        ts.add("1");
        ts.add("2");
        ts.add("3");
        ts.add("A");
        ts.add("B");
        ts.add("Z");

        System.out.println("Initial TreeSet " + ts);

        ts.remove("B");
        System.out.println("After removing element " + ts);

        ts.pollFirst();
        System.out.println("After removing first " + ts);

        ts.pollLast();
        System.out.println("After removing last " + ts);
    }
```

```
}

Output
Initial TreeSet [1, 2, 3, A, B, Z]
After removing element [1, 2, 3, A, Z]
After removing first [2, 3, A, Z]
After removing last [2, 3, A]
```

Operation 4: Iterating through the TreeSet

There are various ways to iterate through the TreeSet. The most famous one is to use the enhanced for loop. And mostly you would be iterating the elements with this approach while practicing questions over TreeSet as this is most frequently used when it comes to tree, maps, and graphs problems.

Example:

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        Set<String> ts = new TreeSet<>();

        ts.add("G");
        ts.add("F");
        ts.add("G");
        ts.add("A");
        ts.add("B");
        ts.add("Z");

        for (String value : ts)
            System.out.print(value + ", ");

        System.out.println();
    }
}

Output
A, B, F, G, Z,
```

Features of a TreeSet

- TreeSet implements the SortedSet interface. So, duplicate values are not allowed.
- Objects in a TreeSet are stored in a sorted and ascending order.
- TreeSet does not preserve the insertion order of elements but elements are sorted by keys.
- If we are depending on the default natural sorting order, the objects that are being inserted into the tree should be homogeneous and comparable. TreeSet does not allow the insertion of heterogeneous objects. It will throw a classCastException at Runtime if we try to add heterogeneous objects.
- The TreeSet can only accept generic types which are comparable.
- For example, the StringBuffer class does NOT implement the Comparable interface. Therefore, inserting StringBuffer objects into a TreeSet without a custom Comparator will throw a ClassCastException.

HashSet:

- HashSet in Java implements the Set interface of Collections Framework. It is used to store the unique elements and it doesn't maintain any specific order of elements.
- Can store the Null values.
- Uses HashMap (implementation of hash table data structure) internally.
- Also implements Serializable and Cloneable interfaces.
- HashSet is not thread-safe. So to make it thread-safe, synchronization is needed externally.

Example:

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        HashSet<Integer> hs = new HashSet<>();

        hs.add(1);
        hs.add(2);
        hs.add(1);

        System.out.println("HashSet Size: " + hs.size());
        System.out.println("Elements in HashSet: " + hs);
    }
}
Output
HashSet Size: 2
Elements in HashSet: [1, 2]
```

Declaring a HashSet

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>,
Cloneable, Serializable
```

where E is the type of elements stored in a HashSet.

Before storing an Object, HashSet checks whether there is an existing entry using hashCode() and equals() methods. In the above example, two lists are considered equal if they have the same elements in the same order. When you invoke the hashCode() method on the two lists, they both would give the same hash since they are equal.

Internal Working of a HashSet

All the classes of the Set interface are internally backed up by Map. HashSet uses HashMap for storing its object internally. You must be wondering that to enter a value in HashMap we need a key-value pair, but in HashSet, we are passing only one value.

Storage in HashMap: Actually the value we insert in HashSet acts as a key to the map Object and for its value, java uses a constant variable. So in the key-value pair, all the values will be the same.

Constructors of HashSet class

To create a HashSet, we need to create an object of the HashSet class. The HashSet class consists of various constructors that allow the possible creation of the HashSet. The following are the constructors available in this class.

Constructor	Description	Syntax
HashSet()	This constructor is used to build an empty HashSet object in which the default initial capacity is 16 and the default load factor is 0.75.	HashSet<E> hs = new HashSet<E>();
HashSet(int initialCapacity)	This constructor is used to build an empty HashSet object in which the initialCapacity is specified at the time of object creation. Here, the default loadFactor remains 0.75.	HashSet<E> hs = new HashSet<E>(int initialCapacity);
HashSet(int	This constructor is used	HashSet<E> hs = new

initialCapacity, float loadFactor)	to build an empty HashSet object in which the initialCapacity and loadFactor are specified at the time of object creation.	HashSet<E>(int initialCapacity, float loadFactor);
HashSet(Collection)	This constructor is used to build a HashSet object containing all the elements from the given collection. In short, this constructor is used when any conversion is needed from any Collection object to the HashSet object.	HashSet<E> hs = new HashSet<E>(Collection C);

Methods in Java HashSet

Method	Description
add(E e)	Used to add the specified element if it is not present, if it is present then return false.
clear()	Used to remove all the elements from the set.
contains(Object o)	Used to return true if an element is present in a set.
remove(Object o)	Used to remove the element if it is present in the set.
iterator()	Used to return an iterator over the element in the set.
isEmpty()	Used to check whether the set is empty or not. Returns true for empty and false for a non-empty condition for set.
size()	Used to return the size of the set.
clone()	Used to create a shallow copy of the set.

Performing Various Operations on HashSet

Let’s see how to perform a few frequently used operations on the HashSet.

1. Adding Elements in HashSet

To add an element to the HashSet, we can use the add() method. However, the insertion order is not retained in the HashSet. We need to keep a note that duplicate elements are not allowed and all duplicate elements are ignored.

Example:

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        // Creating an empty HashSet of string entities
        HashSet<String> hs = new HashSet<String>();

        // Adding elements using add() method
        hs.add("Test");
        hs.add("Test1");
        hs.add("Test1");
        hs.add("Test2");

        // Printing all string entries inside the Set
        System.out.println("HashSet : " + hs);
    }
}

Output
HashSet : [Test, Test1,Test2]
```

2. Removing Elements in HashSet

The values can be removed from the HashSet using the remove() method.

Example:

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        HashSet<String> hs = new HashSet<String>();

        hs.add("1");
        hs.add("2");
        hs.add("3");
        hs.add("A");
        hs.add("B");
        hs.add("Z");

        // Printing the elements of HashSet elements
        System.out.println("HashSet : " + hs);

        // Removing the element B
        hs.remove("B");

        // Printing the updated HashSet elements
        System.out.println("HashSet after removing element : " + hs);

        // Returns false if the element is not present
        System.out.println("B exists in Set : " + hs.remove("B"));
    }
}

Output
HashSet : [1, 2, 3, A, B, Z]
HashSet after removing element [1, 2, 3, A, Z]
B exists in Set : false
```

3. Iterating through the HashSet

Iterate through the elements of HashSet using the iterator() method. Also, the most famous one is to use the enhanced for loop.

Example:

```
import java.util.HashSet;
import java.util.Iterator;

public class GFG  {
    public static void main(String[] args) {
        HashSet<String> hs = new HashSet<>();

        // Add elements to the HashSet
        hs.add("A");
        hs.add("B");
        hs.add("1");
        hs.add("3");
        hs.add("2");
        hs.add("Z");

        // Using iterator() method to iterate
        // Over the HashSet
        System.out.print("Using iterator : ");
        Iterator<String> iterator = hs.iterator();

        while (iterator.hasNext())
            System.out.print(iterator.next() + ", ");

        System.out.println();

        System.out.print("Using enhanced for loop : ");
        for (String element : hs)
            System.out.print(element + " , ");

    }
}

Output
Using iterator : A, B, 1, 2, Z,
Using enhanced for loop : A , B , 1 , 2 , Z ,
```

Initial Capacity: The initial capacity means the number of buckets when the hashtable (HashMap internally uses hashtable data structure) is created. The number of buckets will be automatically increased if the current size gets full.

Load Factor: The load factor is a measure of how full the HashSet is allowed to

get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

$$\text{Load Factor} = \frac{\text{Number of stored elements in the table}}{\text{Size of the hash table}}$$

Example: If internal capacity is 16 and the load factor is 0.75 then the number of buckets will automatically get increased when the table has 12 elements in it.

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

LinkedHashSet:

LinkedHashSet in Java implements the Set interface of the Collection Framework. It combines the functionality of a HashSet with a LinkedList to maintain the insertion order of elements.

- Stores unique elements only.
- Maintains insertion order.
- Provides faster iteration compared to HashSet.
- Allows null elements.

Example:

```
import java.util.LinkedHashSet;

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

        // Create a LinkedHashSet of Strings
        LinkedHashSet<String> lh = new LinkedHashSet<>();

        System.out.println("" + lh);
    }
}
```

Details of Queue Interface (Deque(I), PriorityQueue, ArrayDeque)

The Queue Interface is a part of java.util package and extends the Collection interface. It stores and processes the data in order means elements are inserted at the end and removed from the front.

Key Features:

1. Most implementations, like PriorityQueue, do not allow null elements.
2. Implementation Classes:
    a. LinkedList
    b. PriorityQueue
    c. ArrayDeque
    d. ConcurrentLinkedQueue (for thread-safe operations)
3. Commonly used for task scheduling, message passing, and buffer management in applications.
4. It supports iterating through elements. The order of iteration depends on the implementation.

Declaration of Java Queue Interface

The Queue interface is declared as:

```
public interface Queue extends Collection
```

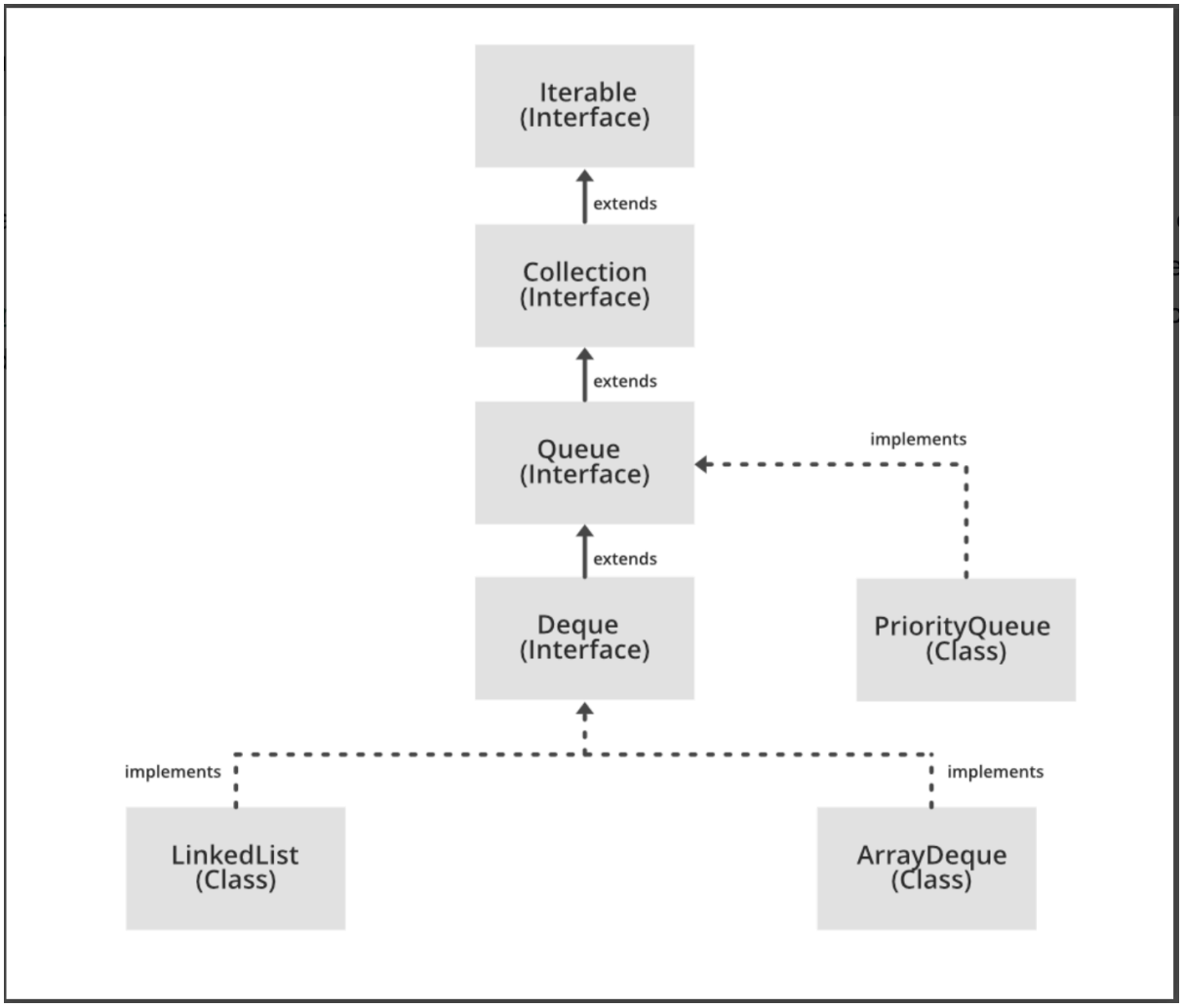
We cannot instantiate a Queue directly as it is an interface. Here, we can use a class like LinkedList or PriorityQueue that implements this interface.

```
Queue<Obj> queue = new LinkedList<Obj>();
```

Example: Basic Queue using LinkedList

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

public class QueueExample {
    public static void main(String args[]) {
        Queue<Integer> q = new LinkedList<>();

        System.out.println("Queue elements: " + q);
    }
}

Output
Queue elements: []
```

Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java. Note that neither of these implementations is thread-safe. PriorityBlockingQueue is one alternative implementation if the thread-safe implementation is needed.



Common Methods

The Queue interface provides several methods for adding, removing, and inspecting elements in the queue. Here are some of the most commonly used methods:

- **add(element):** Adds an element to the rear of the queue. If the queue is full, it throws an exception.
- **offer(element):** Adds an element to the rear of the queue. If the queue is full, it returns false.
- **remove():** Removes and returns the element at the front of the queue. If the queue is empty, it throws an exception.
- **poll():** Removes and returns the element at the front of the queue. If the queue is empty, it returns null.
- **element():** Returns the element at the front of the queue without removing it. If the queue is empty, it throws an exception.

- **peek():** Returns the element at the front of the queue without removing it. If the queue is empty, it returns null.

Example 1: This example demonstrates basic queue operations.

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

public class Main {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // add elements to the queue
        queue.add("apple");
        queue.add("banana");
        queue.add("cherry");

        System.out.println("Queue: " + queue);

        // remove the element at the front of the queue
        String front = queue.remove();
        System.out.println("Removed element: " + front);

        // print the updated queue
        System.out.println("Queue after removal: " + queue);

        // add another element to the queue
        queue.add("date");

        // peek at the element at the front of the queue
        String peeked = queue.peek();
        System.out.println("Peeked element: " + peeked);

        // print the updated queue
        System.out.println("Queue after peek: " + queue);
    }
}

Output
Queue: [apple, banana, cherry]
Removed element: apple
Queue after removal: [banana, cherry]
Peeked element: banana
Queue after peek: [banana, cherry, date]
```

Deque Interface in Java

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 adding or removing 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.

- **Null Handling:** Most implementations do not allow null elements, as null is used as a special return value to indicate the absence of elements.
- **Thread-Safe Alternatives:** Use **ConcurrentLinkedDeque** or **LinkedBlockingDeque** for thread-safe operations and avoid ArrayDeque in concurrent environments as it is not thread-safe.

Example:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        Deque<String> d = new ArrayDeque<>();

        d.addFirst("1");
        d.addLast("2");

        String f = d.removeFirst();
        String l = d.removeLast();

        System.out.println("First: " + f + ", Last: " + l);
    }
}

Output
First: 1, Last: 2
```

Syntax: The deque interface is declared as,

```
public interface Deque extends Queue
```

Creating Deque Objects

Since Deque is an interface, objects cannot be created of the type deque. We always need a class that extends this list in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Deque. This type-safe queue can be defined as:

```
// Obj is the type of the object to be stored in Deque
Deque<Obj> deque = new ArrayDeque<> ();
```

Example: Deque

```
import java.util.*;

public class DequeExample {
    public static void main(String[] args){
        Deque<String> deque = new LinkedList<String>();

        deque.add("Element 1 (Tail)");

        // Add at the first
        deque.addFirst("Element 2 (Head)");

        // Add at the last
        deque.addLast("Element 3 (Tail)");

        // Add at the first
        deque.push("Element 4 (Head)");

        // Add at the last
        deque.offer("Element 5 (Tail)");

        // Add at the first
        deque.offerFirst("Element 6 (Head)");

        System.out.println(deque + "\n");

        // We can remove the first element
        // or the last element.
        deque.removeFirst();
        deque.removeLast();
        System.out.println("Deque after removing "
                           + "first and last: "
                           + deque);
    }
}

Output
[Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail),
Element 3 (Tail), Element 5 (Tail)]

Deque after removing first and last: [Element 4 (Head), Element 2
(Head), Element 1 (Tail)...
```

Methods of Deque Interface

The following are the methods present in the deque interface:

Method	Description
add(element)	This method is used to add an element at the tail of the queue. If the Deque is capacity restricted and no space is left for insertion, it returns an IllegalStateException. The function returns true on successful insertion.
addFirst(element)	This method is used to add an element at the head of the queue. If the Deque is capacity restricted and no space is left for insertion, it returns an IllegalStateException. The function returns true on successful insertion.
addLast(element)	This method is used to add an element at the tail of the queue. If the Deque is capacity restricted and no space is left for insertion, it returns an IllegalStateException. The function returns true on successful insertion.
contains()	This method is used to check whether the queue contains the given object or not.
descendingIterator()	This method returns an iterator for the deque. The elements will be returned in order from last(tail) to first(head).
element()	This method is used to retrieve, but not remove, the head of the queue represented by this deque.
getFirst()	This method is used to retrieve, but not remove, the first element of this deque.
getLast()	This method is used to retrieve, but not remove, the last element of this deque.
iterator()	This method returns an iterator for the deque. The elements will be returned in order from first (head) to last (tail).

offer(element)	This method is used to add an element at the tail of the queue. This method is preferable to the add() method since this method does not throw an exception when the capacity of the container is full since it returns false.
offerLast(element)	This method is used to add an element at the tail of the queue. This method is preferable to the add() method since this method does not throw an exception when the capacity of the container is full since it returns false.
peek()	This method is used to retrieve the element at the head of the deque but doesn't remove the element from the deque. This method returns null if the deque is empty.
peekFirst()	This method is used to retrieve the element at the head of the deque but doesn't remove the element from the deque. This method returns null if the deque is empty.
peekLast()	This method is used to retrieve the element at the tail of the deque but doesn't remove the element from the deque. This method returns null if the deque is empty.
offerFirst(element)	This method is used to add an element at the head of the queue. This method is preferable to the addFirst() method since this method does not throw an exception when the capacity of the container is full since it returns false.
poll()	This method is used to retrieve and remove the element at the head of the deque. This method returns null if the deque is empty.
pollFirst()	This method is used to retrieve and remove the element at the head of the deque. This method returns null if the deque is empty.

pollLast()	This method is used to retrieve and remove the element at the tail of the deque. This method returns null if the deque is empty.
pop()	This method is used to remove an element from the head and return it.
push(element)	This method is used to add an element at the head of the queue.
removeFirst()	This method is used to remove an element from the head of the queue.
removeLast()	This method is used to remove an element from the tail of the queue.
size()	This method is used to find and return the size of the deque.

ArrayDeque in Java

In Java, the ArrayDeque is a resizable array implementation of the Deque interface, which stands for double-ended queue. It allows elements to be added or removed from both ends efficiently. It can be used as a stack (LIFO) or a queue (FIFO).

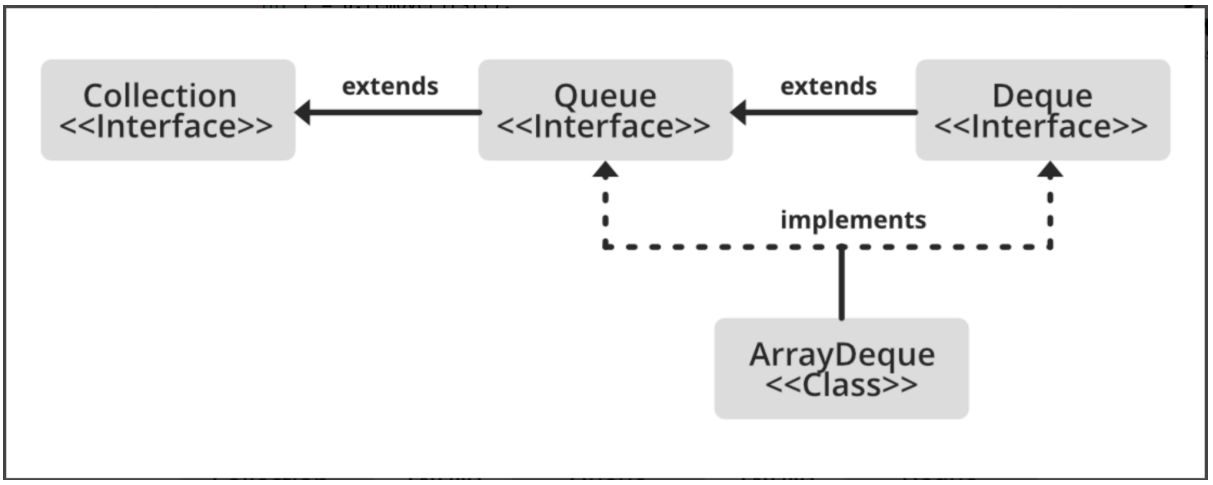
1. ArrayDeque grows dynamically.
2. It generally provides faster operations than LinkedList because it does not have the overhead of node management.
3. Operations like addFirst(), addLast(), removeFirst(), removeLast() are all done in constant time O(1).
4. The ArrayDeque class implements these two interfaces Queue interface and Deque interface. support concurrent access by multiple threads.

Example: This example demonstrates how to use an ArrayDeque to add elements to both ends of the deque and then remove them from both ends using addFirst(), addLast(), removeFirst(), and removeLast() methods.

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        Deque<Integer> d = new ArrayDeque<>();
        d.addFirst(1);
        d.addLast(2);
        int f = d.removeFirst();
        int l = d.removeLast();
        System.out.println("First: " + f + ", Last: " + l);
    }
}

Output
First: 1, Last: 2
```

ArrayDeque Hierarchy

The below image demonstrates the inheritance hierarchy of ArrayDeque, how it implements the Deque interface, which extends the Queue , and how both are part of the Collection interface.



Declaration of ArrayDeque

In Java, the declaration of ArrayDeque can be done as:

```
ArrayDeque<Type> deque = new ArrayDeque<>();
```

Note: Here, "Type" is the type of the element the deque will hold (e.g. Integer, String).

Constructors

Constructor	Description
ArrayDeque()	This constructor is used to create an empty ArrayDeque and by default holds an initial capacity to hold 16 elements
ArrayDeque(Collection<? extends E> c)	This constructor is used to create an ArrayDeque containing all the elements the same as that of the specified collection.
ArrayDeque(int numofElements)	This constructor is used to create an empty ArrayDeque and holds the capacity to contain a specified number of elements

Example 1: This example demonstrates how to create an empty ArrayDeque, add elements to it, and print the deque's contents.

```
import java.util.ArrayDeque;

public class Main {
    public static void main(String[] args) {
        ArrayDeque<Integer> d = new ArrayDeque<>();

        d.add(10);
        d.add(20);
        System.out.println("ArrayDeque: " + d);
    }
}

Output
ArrayDeque: [10, 20]
```

Example 2: This example demonstrates how to initialize an ArrayDeque with elements from a collection and print its contents.

```
import java.util.ArrayDeque;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        ArrayDeque<Integer> d
            = new ArrayDeque<>(Arrays.asList(1, 2, 3, 4));
        System.out.println("ArrayDeque: " + d);
    }
}

Output
ArrayDeque: [1, 2, 3, 4]
```

Example 3: This example demonstrates how to create an ArrayDeque with a specified initial capacity (10) and add elements to it.

```
import java.util.ArrayDeque;

public class Main {
    public static void main(String[] args)
    {
        ArrayDeque<Integer> d = new ArrayDeque<>(10);
        d.add(5);
        d.add(15);
        System.out.println("ArrayDeque: " + d);
    }
}

Output
ArrayDeque: [5, 15]
```

Performing Various Operations on ArrayDeque

1. Adding Element: We can use methods like add(), addFirst(), addLast(), offer(), offerFirst(), offerLast() to insert elements to the ArrayDeque.

Example: This example demonstrates how to use various methods add(), addFirst(), addLast(), offer(), offerFirst(), offerLast() to insert elements in the Deque.

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

public class Main {
    public static void main(String[] args) {
        Deque<String> d = new ArrayDeque<String>();

        // add() method to insert
        d.add("1");
        d.addFirst("2");
        d.addLast("3");

        // offer() method to insert
        d.offer("4");
        d.offerFirst("5");
        d.offerLast("6");

        System.out.println("ArrayDeque : " + d);
    }
}

Output
ArrayDeque : [5, 2, 1, 3, 4, 6]
```

2. Accessing Elements: We can use methods like getFirst(), getLast(), peek(), peekFirst(), peekLast() to access elements of the ArrayDeque.

Example: This example demonstrates how to access the elements of the ArrayDeque using getFirst() and getLast() methods.

```
import java.io.*;
import java.util.*;
public class Main {
    public static void main(String args[]) {
        // Creating an empty ArrayDeque
        ArrayDeque<String> d
                = new ArrayDeque<String>();

        d.add("1");
        d.add("2");
        d.add("3");
        d.add("4");
        d.add("5");

        // Displaying the ArrayDeque
        System.out.println("ArrayDeque: " + d);

        // Displaying the First element
        System.out.println("The first element is: " + d.getFirst());

        // Displaying the Last element
        System.out.println("The last element is: " + d.getLast());
    }
}

Output
ArrayDeque: [1, 2, 3, 4, 5]
The first element is: 1
The last element is: 5
```

3. Removing Elements: We can use various methods like remove(), removeFirst(), removeLast(), poll(), pollFirst(), pollLast(), pop() to remove elements from the ArrayDeque.

Example: This example demonstrates removing elements from the ArrayDeque using pop(), poll() and pollFirst() methods.

```
// Java program to demonstrates
// Removal Elements in Deque
import java.util.*;
public class Main {
    public static void main(String[] args) {

        // Initializing a deque
        Deque<String> d = new ArrayDeque<String>();

        // Adding elements
        d.add("Java");
        d.addFirst("C++");
        d.addLast("Python");

        // Printing initial elements
        System.out.println("Initial Deque: " + d);

        // Removing elements as a stack from top/front
        System.out.println("Removed element using pop(): " + d.pop());

        // Removing an element from the front
        System.out.println("Removed element using poll(): " + d.poll());

        // Removing an element from the front using pollFirst
        System.out.println("Removed element using pollFirst(): " +
d.pollFirst());

        // The deque is empty now
        System.out.println("Final Deque: " + d);
    }
}

Output
Initial Deque: [C++, Java, Python]
Removed element using pop(): C++
Removed element using poll(): Java
Removed element using pollFirst(): Python
Final Deque: []
```

Methods

Method	Description
add(Element e)	The method inserts a particular element at the end of the deque.
addAll(Collection<? extends E> c)	Adds all of the elements in the specified collection at the end of this deque, as if by calling addLast(E) on each one, in the order that they are returned by the collection's iterator.
addFirst(Element e)	The method inserts a particular element at the start of the deque.
addLast(Element e)	The method inserts a particular element at the end of the deque. It is similar to the add() method
clear()	The method removes all deque elements.
clone()	The method copies the deque.
contains(Obj)	The method checks whether a deque contains the element or not
element()	The method returns element at the head of the deque
forEach(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
getFirst()	The method returns first element of the deque
getLast()	The method returns last element of the deque
isEmpty()	The method checks whether the deque is empty or not.
iterator()	Returns an iterator over the elements in this deque.
offer(Element e)	The method inserts elements at the end of deque.
offerFirst(Element e)	The method inserts elements at the front of the deque.

offerLast(Element e)	The method inserts elements at the end of the deque.
peek()	The method returns the head element without removing it.
poll()	The method returns head element and also removes it
pop()	The method pops out an element for stack represented by deque
push(Element e)	The method pushes an element onto stack represented by deque
remove()	The method returns head element and also removes it
remove(Object o)	Removes a single instance of the specified element from this deque.
removeAll(Collection<?> c)	Removes all of this collection's elements that are also contained in the specified collection (optional operation).
removeFirst()	The method returns the first element and also removes it
removeFirstOccurrence(Object o)	Removes the first occurrence of the specified element in this deque (when traversing the deque from head to tail).
removeIf(Predicate<? super Element> filter)	Removes all of the elements of this collection that satisfy the given predicate.
removeLast()	The method returns the last element and also removes it
removeLastOccurrence(Object o)	Removes the last occurrence of the specified element in this deque (when traversing the deque from head to tail).
retainAll(Collection<?> c)	Retains only the elements in this collection that are contained in the specified collection (optional operation).

size()	Returns the number of elements in this deque.
spliterator()	Creates a late-binding and fail-fast Spliterator over the elements in this deque.
toArray()	Returns an array containing all of the elements in this deque in proper sequence (from first to the last element).
toArray(T[] a)	Returns an array containing all of the elements in this deque in proper sequence (from first to the last element); the runtime type of the returned array is that of the specified array.

PriorityQueue in Java

The PriorityQueue class in Java is part of the java.util package. It implements a priority heap-based queue that processes elements based on their priority rather than the FIFO (First-In-First-Out) concept of a Queue.

Key Points:

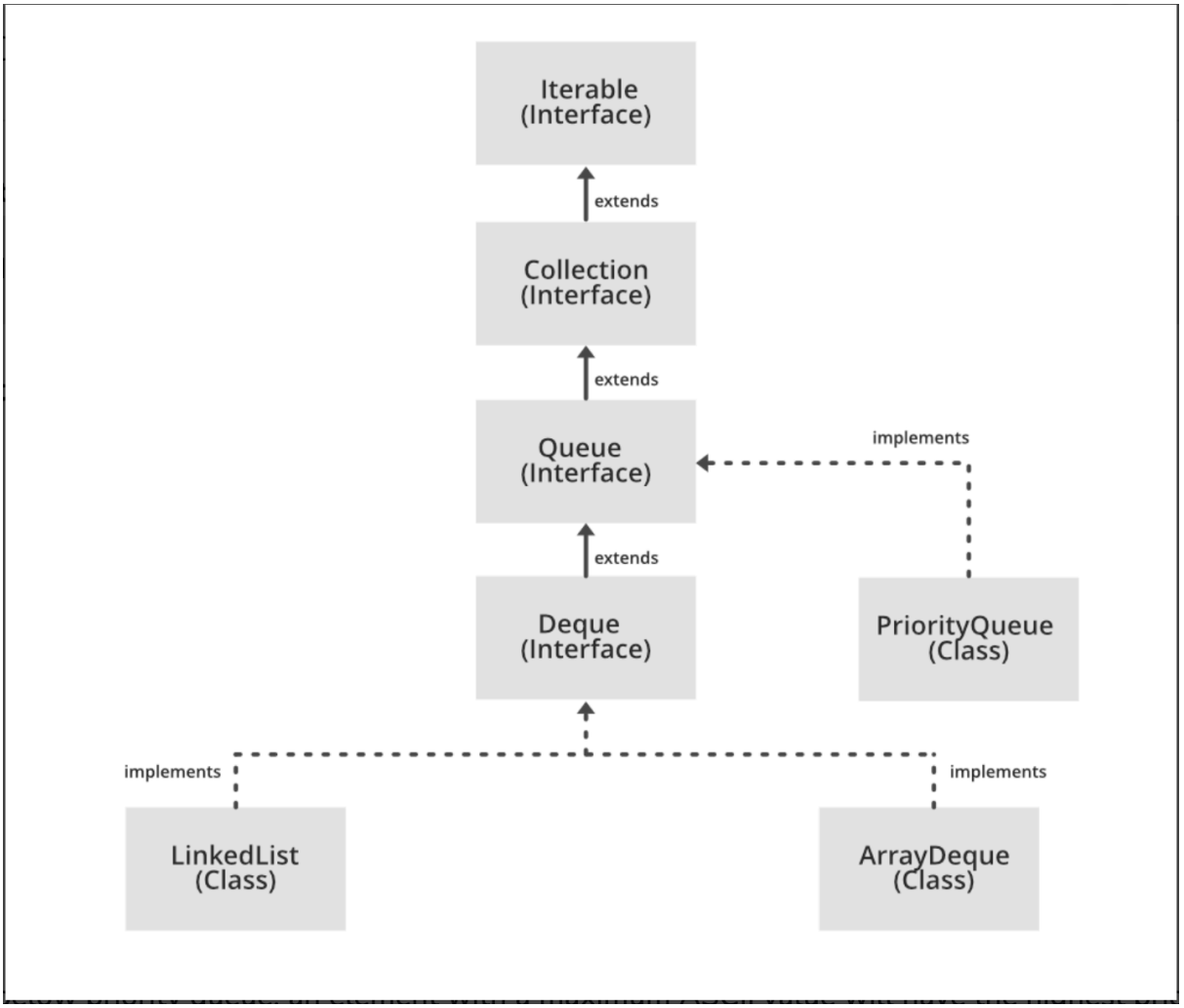
- The PriorityQueue is based on the Priority Heap.
- The elements of the priority queue are ordered according to the natural ordering, and elements must implement Comparable, or by a Comparator provided at queue construction time, depending on which constructor is used.
- The size of the Priority Queue is dynamic, this means it will increase or decrease as per the requirement.

Example:

```
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Integer> p = new PriorityQueue<>();
        p.add(3);
        p.add(10);
        p.add(7);
        p.add(2);
        System.out.println("Head of Queue: " + p.peek());

    }
}
```

```
Output
Head of Queue: 2
```

PriorityQueue Class Hierarchy in Java



Declaration of PriorityQueue

```
public class PriorityQueue<E> extends AbstractQueue<E> implements
Serializable
```

where, E is the type of elements held in this queue. The class implements Serializable, Iterable<E>, Collection<E>, Queue<E> interfaces.

Important Points:

- PriorityQueue does not permit null.
- We cannot create a PriorityQueue of Objects that are non-comparable.
- PriorityQueue are unbound queues.
- The head of this queue is the least element with respect to the specified ordering.
- Since PriorityQueue is not thread-safe, Java provides a PriorityBlockingQueue class that implements the BlockingQueue interface to use in a Java multithreading environment.
- The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
- It provides O(log(n)) time for add and poll methods.
- It inherits methods from AbstractQueue, AbstractCollection, Collection, and Object class.

Constructors

1. PriorityQueue(): This method creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.

```
PriorityQueue<E> pq = new PriorityQueue<E>();
```

2. PriorityQueue(Collection<E> c): This creates a PriorityQueue containing the elements in the specified collection.

```
PriorityQueue<E> pq = new PriorityQueue<E>(Collection<E> c);
```

3. PriorityQueue(int initialCapacity): This creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.

```
PriorityQueue<E> pq = new PriorityQueue<E>(int initialCapacity);
```

4. PriorityQueue(int initialCapacity, Comparator<E> comparator): This creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.

```
PriorityQueue<E> pq = new PriorityQueue(int initialCapacity,
Comparator<E> comparator);
```

5. PriorityQueue(PriorityQueue<E> c): This creates a PriorityQueue containing the elements in the specified priority queue.

```
PriorityQueue<E> pq = new PriorityQueue(PriorityQueue<E> c)
```

6. PriorityQueue(SortedSet<E> c): This creates a PriorityQueue containing the elements in the specified sorted set.

```
PriorityQueue<E> pq = new PriorityQueue<E>(SortedSet<E> c);
```

7. PriorityQueue(Comparator<E> comparator): This creates a PriorityQueue with the default initial capacity and whose elements are ordered according to the specified comparator.

```
PriorityQueue<E> pq = new PriorityQueue<E>(Comparator<E> c);
```

Example:

The example below explains the following basic operations of the priority queue.

- **boolean add(E element):** This method inserts the specified element into this priority queue.
- **public peek():** This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
- **public poll():** This method retrieves and removes the head of this queue, or returns null if this queue is empty.

```
import java.util.*;

class Main {
    public static void main(String args[]) {
        // Creating empty priority queue
        PriorityQueue<Integer> p = new PriorityQueue<Integer>();

        // Adding items to the pQueue using add()
        p.add(10);
        p.add(20);
        p.add(15);

        // Printing the top element of PriorityQueue
        System.out.println(p.peek());

        // Printing the top element and removing it
        // from the PriorityQueue container
        System.out.println(p.poll());

        // Printing the top element again
        System.out.println(p.peek());
    }
}
Output
10
10
15
```

Different Operations on PriorityQueue
Let's see how to perform a few frequently used operations on the Priority Queue class.

1. Adding Elements
To add an element in a priority queue, we can use the add() method. The insertion order is not retained in the PriorityQueue. The elements are stored based on the priority order which is ascending by default.

Example:

```
// Java Program to add elements in a PriorityQueue
import java.util.*;

public class Main {
    public static void main(String args[]) {
        // Creating PriorityQueue
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        for(int i=0; i<3; i++){
            pq.add(i);
            pq.add(1);
        }

        System.out.println(pq);
    }
}

Output
[0, 1, 1, 1, 2, 1]
```

Time and Space Complexities:
Time Complexity: O(log n)
Space Complexity: O(n), where n is the number of elements in the priority queue.

Note: We will not get sorted elements by printing PriorityQueue. Below is an example for this,

Example:

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

public class Main {

    public static void main(String args[]) {
        PriorityQueue<String> pq = new PriorityQueue<>();

        pq.add("3");
        pq.add("1");
        pq.add("2");

        System.out.println(pq);
    }
}

Output
[1, 2, 3]
```

2. Removing Elements

To remove an element from a priority queue, we can use the remove() method. If there are multiple such objects, then the first occurrence of the object is removed. Apart from that, the poll() method is also used to remove the head and return it.

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

public class Main {
    public static void main(String args[]) {
        PriorityQueue<String> pq = new PriorityQueue<>();
        pq.add("2");
        pq.add("1");
        pq.add("3");

        System.out.println("Initial PriorityQueue " + pq);

        pq.remove("2");
        System.out.println("After Remove: " + pq);
        System.out.println("Poll Method: " + pq.poll());
        System.out.println("Final PriorityQueue: " + pq);
    }
}
```

```
Output
Initial PriorityQueue [1, 2, 3]
After Remove: [1, 3]
Poll Method: 1
Final PriorityQueue: [3]
```

Time and Space Complexities:
Time Complexity: O(n)
Space Complexity: O(n), where n is the number of elements in the priority queue.

3. Accessing the Elements
Queue follows the First In First Out principle, we can access only the head of the queue. To access elements from a priority queue, we can use the peek() method.

Example:

```
import java.util.*;

class Main {
    public static void main(String[] args) {
        // Creating a priority queue
        PriorityQueue<String> pq = new PriorityQueue<>();
        pq.add("3");
        pq.add("1");
        pq.add("2");
        System.out.println("PriorityQueue: " + pq);

        // Using the peek() method
        String element = pq.peek();
        System.out.println("Accessed Element: " + element);
    }
}

Output
PriorityQueue: [1, 2, 3]
Accessed Element: 1
```

Time and Space Complexities:

- **Time Complexity:** O(1)
- **Space Complexity:** O(n), where n is the number of elements in the priority queue.

4. Iterating the PriorityQueue

There are multiple ways to iterate through the PriorityQueue. The most famous way is converting the queue to the array and traversing using an iterator.

Example:

```
import java.util.*;

public class Main {
    public static void main(String args[]) {
        PriorityQueue<String> pq = new PriorityQueue<>();

        pq.add("3");
        pq.add("1");
        pq.add("2");

        Iterator iterator = pq.iterator();

        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }
}

Output
1 2 3
```

Time and Space Complexities:

- **Time Complexity:** O(n)
- **Space Complexity:** O(1) per iteration, where n is the number of elements in the priority queue.

Methods in PriorityQueue Class

Method	Description
add(E e)	Inserts the specified element into this priority queue.
clear()	Removes all of the elements from this priority queue.
comparator()	Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements.
contains?(Object o)	Returns true if this queue contains the specified element.
forEach?(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
iterator()	Returns an iterator over the elements in this queue.
offer(E e)	Inserts the specified element into this priority queue.
remove(Object o)	Removes a single instance of the specified element from this queue, if it is present.
removeAll(Collection<?> c)	Removes all of this collection's elements that are also contained in the specified collection (optional operation).
removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.
retainAll(Collection<?> c)	Retains only the elements in this collection that are contained in the specified collection (optional operation).
spliterator()	Creates a late-binding and fail-fast Spliterator over the elements in this queue.
toArray()	Returns an array containing all of the elements in this queue.
toArray?(T[] a)	Returns an array containing all of the

	elements in this queue; the runtime type of the returned array is that of the specified array.

Methods Declared in Class java.util.AbstractQueue

Method	Description
addAll(Collection<? extends E> c)	Adds all of the elements in the specified collection to this queue.
element()	Retrieves, but does not remove, the head of this queue.
remove()	Retrieves and removes the head of this queue.

Methods Declared in Class java.util.AbstractCollection

Method	Description
containsAll(Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
isEmpty()	Returns true if this collection contains no elements.
toString()	Returns a string representation of this collection.

Methods Declared in Interface java.util.Collection

Method	Description
containsAll(Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
equals(Object o)	Compare the specified object with this collection for equality.
toString()	Returns a string representation of this collection.
hashCode()	Returns the hash code value for this collection.
isEmpty()	Returns true if this collection contains no elements.
parallelStream()	Returns a possibly parallel Stream

	with this collection as its source.
stream()	Returns a sequential Stream with this collection as its source.
toArray(IntFunction<T[]> generator)	Returns an array containing all of the elements in this collection, using the provided generator function to allocate the returned array.
size()	Returns the number of elements in this collection.

Methods Declared in Interface java.util.Queue

Method	Description
peek()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
poll()	Retrieves and removes the head of this queue, or returns null if this queue is empty.

Details of Map Interface (HashMap, LinkedHashMap, TreeMap)

In Java, the Map Interface is part of the java.util package and represents a mapping between a key and a value. The Java Map interface is not a subtype of the Collections interface. So, it behaves differently from the rest of the collection types.

Key Features:

- **No Duplicates in Keys:** Keys should be unique, but values can be duplicated.
- **Null Handling:** It allows one null key in implementations like HashMap and LinkedHashMap, and allows multiple null values in most implementations.
- **Thread-Safe Alternatives:** Use ConcurrentHashMap for thread-safe operations. Also, wrap an existing map using Collections.synchronizedMap() for synchronized access.

The Map data structure in Java is implemented by two interfaces:

- Map Interface
- SortedMap Interface

The three primary classes that implement these interfaces are,

- HashMap
- TreeMap
- LinkedHashMap

Now, let us go through a simple example first to understand the concept.

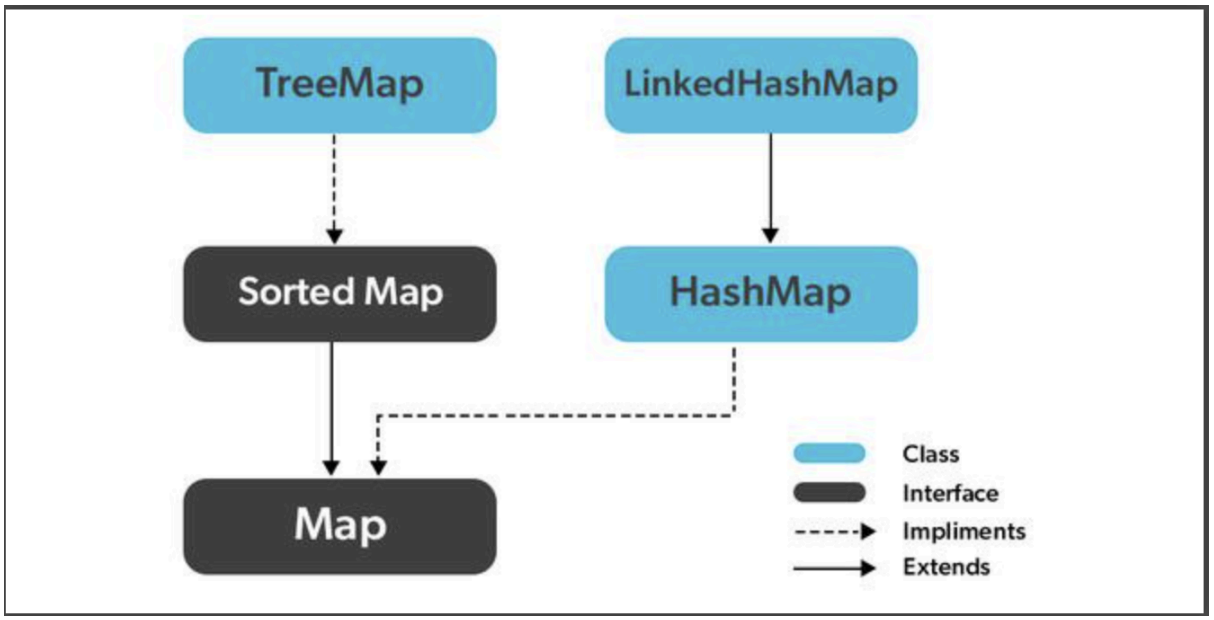
Example:

```
import java.util.HashMap;
import java.util.Map;

public class MapExample {
    public static void main(String[] args) {
        // Create a Map using HashMap
        Map<String, Integer> m = new HashMap<>();

        // Adding key-value pairs to the map
        m.put("1", 1);
        m.put("2", 2);
        m.put("3", 3);
        System.out.println("Map elements: " + m);
    }
}
Output
Map elements: {3=3, 2=2, 1=1}
```

We must know why and when to use Maps. Maps are perfect to use for key-value mapping such as dictionaries. Some common scenarios are as follows:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).

Hierarchy of Map Interface in Java



Creating Map Objects

Since Map is an interface, objects cannot be created of the type map. We always need a class that implements this map interface in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Map.

Syntax: Defining Type-safe Map:

```
Map<String, Integer> hm = new HashMap<>();  // Type-safe map storing
String keys and Integer values
```

Characteristics of a Map Interface

- 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.

Methods in Java Map Interface

Methods	Action Performed
clear()	This method is used in Java Map Interface to clear and remove all of the elements or mappings from a specified Map collection.
containsKey(Object)	This method is used in Map Interface in Java to check whether a particular key is being mapped into the Map or not. It takes the key element as a parameter and returns True if that element is mapped in the map.
containsValue(Object)	This method is used in Map Interface to check whether a particular value is being mapped by a single or more than one key in the Map. It takes the value as a parameter and returns True if that value is mapped by any of the keys in the map.
entrySet()	This method is used in Map Interface in Java to create a set out of the same elements contained in the map. It basically returns a set view of the map or we can create a new set and store the map elements into them.
equals(Object)	This method is used in Java Map Interface to check for equality between two maps. It verifies whether the elements of one map passed as a parameter is equal to the elements of this map or not.
get(Object)	This method is used to retrieve or fetch the value mapped by a particular key mentioned in the parameter. It returns NULL when map contains no such mapping for the key.

hashCode()	This method is used in Map Interface to generate a hashCode for the given map containing keys and values.
isEmpty()	This method is used to check if a map has any entry for key and value pairs. If no mapping exists, then this returns true.
keySet()	This method is used in Map Interface to return a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.
put(Object, Object)	This method is used in Java Map Interface to associate the specified value with the specified key in this map.
putAll(Map)	This method is used in Map Interface in Java to copy all of the mappings from the specified map to this map.
remove(Object)	This method is used in Map Interface to remove the mapping for a key from this map if it is present in the map.
size()	This method is used to return the number of key/value pairs available in the map.
values()	This method is used in Java Map Interface to create a collection out of the values of the map. It basically returns a Collection view of the values in the HashMap.
getOrDefault(Object key, V defaultValue)	Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associate it with the given non-null value.

putIfAbsent(K key, V value)	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current associate value.

Example:

```
import java.util.*;

class Main{
    public static void main(String args[]) {
        // Creating an empty HashMap
        Map<String, Integer> hm
            = new HashMap<String, Integer>();

        hm.put("a", new Integer(100));
        hm.put("b", new Integer(200));
        hm.put("c", new Integer(300));
        hm.put("d", new Integer(400));

        // Traversing through Map using for-each loop
        for (Map.Entry<String, Integer> me :
             hm.entrySet()) {

            System.out.print(me.getKey() + ":");
            System.out.println(me.getValue());
        }
    }
}

Output
a:100
b:200
c:300
d:400
```

HashMap in Java

In Java, HashMap is part of the Java Collections Framework and is found in the java.util package. It provides the basic implementation of the Map interface in Java. HashMap stores data in (key, value) pairs. Each key is associated with a value, and you can access the value by using the corresponding key.

- Internally uses Hashing (similar to Hashtable in Java).
- Not synchronized (unlike Hashtable in Java) and hence faster for most of the cases.
- Allows to store the null keys as well, but there should be only one null key object, and there can be any number of null values.
- Duplicate keys are not allowed in HashMap, if you try to insert the duplicate key, it will replace the existing value of the corresponding key.
- HashMap uses keys in the same way as an Array uses an index.
- HashMap allows for efficient key-based retrieval, insertion, and removal with an average O(1) time complexity.

Example:

```
// Create a HashMap
 HashMap<String, Integer> hashMap = new HashMap<>();

// Add elements to the HashMap
 hashMap.put("John", 25);
 hashMap.put("Jane", 30);
 hashMap.put("Jim", 35);

Output
25
false
2
```

HashMap Declaration

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>,
Cloneable, Serializable
```

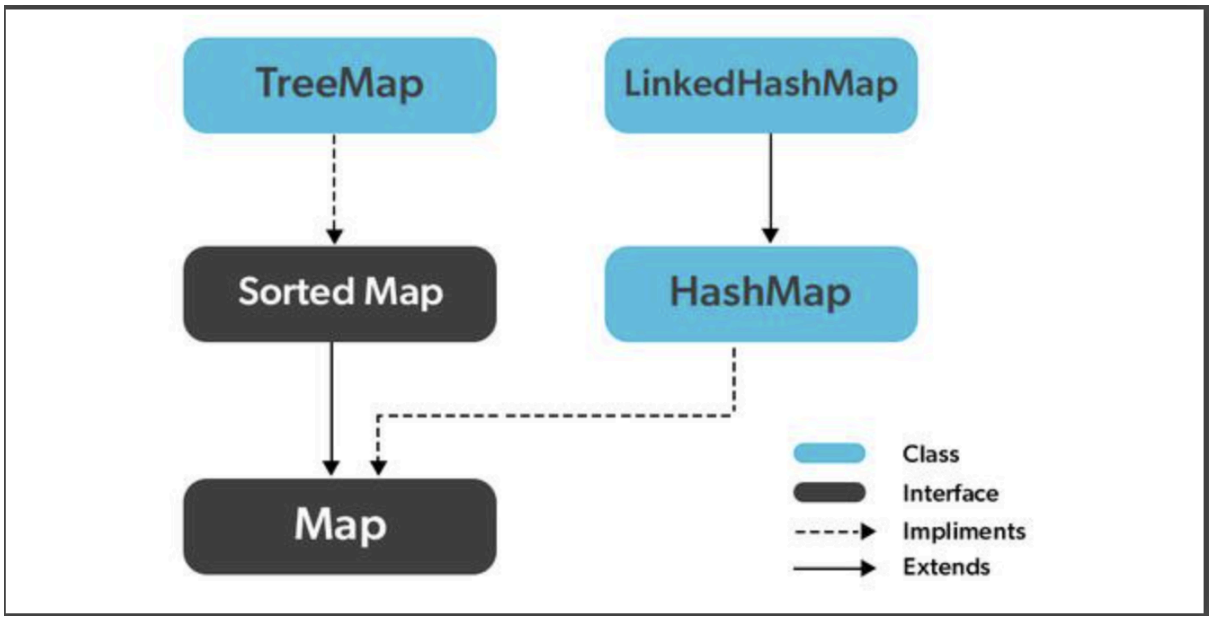
It takes two parameters namely as follows:

- The type of keys maintained by this map
- The type of mapped values

Note: Keys and values can't be primitive datatypes. The key in Hashmap is valid if it implements hashCode() and equals() method , it should also be immutable (immutable custom object) so that hashcode and equality remains constant. Value in hashmap can be any wrapper class, custom objects, arrays, any reference type or even null . For example, Hashmap can have an array as value but not as key.

HashMap in Java implements Serializable, Cloneable, Map<K, V> interfaces.Java HashMap extends AbstractMap<K, V> class. The direct subclasses are LinkedHashMap and PrinterStateReasons.

Hierarchy of HashMap in Java



Characteristics of HashMap

A HashMap is a data structure that is used to store and retrieve values based on keys. Some of the key characteristics of a hashmap include:

- **Not ordered:** HashMaps are not ordered, which means that the order in which elements are added to the map is not preserved. However, LinkedHashMap is a variation of HashMap that preserves the insertion order.
- **Thread-unsafe:** HashMaps are not thread-safe, which means that if multiple threads access the same hashmap simultaneously, it can lead to data inconsistencies. If thread safety is required, ConcurrentHashMap can be used.
- **Capacity and load factor:** HashMaps have a capacity, which is the number of elements that it can hold, and a load factor, which is the measure of how full the hashmap can be before it is resized.

Java HashMap Constructors

HashMap provides 4 constructors and the access modifier of each is public which are listed as follows:

1. HashMap()
2. HashMap(int initialCapacity)
3. HashMap(int initialCapacity, float loadFactor)
4. HashMap(Map map)

Now discuss the above constructors one by one alongside implementing the same with the help of clean Java programs.

1. HashMap()
It is the default constructor which creates an instance of HashMap with an initial capacity of 16 and a load factor of 0.75.

Syntax:

```
HashMap<K, V> hm = new HashMap<K, V>();
```

2. HashMap(int initialCapacity)
It creates a HashMap instance with a specified initial capacity and load factor of 0.75.

Syntax:

```
HashMap<K, V> hm = new HashMap<K, V>(int initialCapacity);
```

3. HashMap(int initialCapacity, float loadFactor)
It creates a HashMap instance with a specified initial capacity and specified load factor.

Syntax:

```
HashMap<K, V> hm = new HashMap<K, V>(int initialCapacity, float
loadFactor);
```

4. HashMap(Map map)
It creates an instance of HashMap with the same mappings as the specified map.

Syntax:

```
HashMap<K, V> hm = new HashMap<K, V>(Map map);
```

1. Adding Elements in HashMap in Java

To add an element to the map, we can use the put() method. However, the insertion order is not retained in the Hashmap. Internally, for every element, a separate hash is generated and the elements are indexed based on this hash to make it more efficient.

```
// Java program to add elements
// to the HashMap
import java.io.*;
import java.util.*;

class AddElementsToHashMap {
    public static void main(String args[])
    {
        // No need to mention the
        // Generic type twice
        HashMap<Integer, String> hm1 = new HashMap<>();

        // Initialization of a HashMap
        // using Generics
        HashMap<Integer, String> hm2
            = new HashMap<Integer, String>();

        // Add Elements using put method
        hm1.put(1, "TEST");
        hm1.put(2, "TEST1");
        hm1.put(3, "TEST2");

        hm2.put(1, "TEST");
        hm2.put(2, "TEST1");
        hm2.put(3, "TEST2");

        System.out.println("Mappings of HashMap hm1 are : "
                           + hm1);
        System.out.println("Mapping of HashMap hm2 are : "
                           + hm2);
    }
}

Output
Mappings of HashMap hm1 are : {1=TEST, 2=TEST1, 3=TEST2}
Mapping of HashMap hm2 are : {1=TEST, 2=TEST1, 3=TEST2}
```

2. Changing Elements in HashMap in Java

After adding the elements, if we wish to change the element, it can be done by again adding the element with the put() method. Since the elements in the map are indexed using the keys, the value of the key can be changed by simply inserting the updated value for the key for which we wish to change.

```
// Java program to change
// elements of HashMap

import java.io.*;
import java.util.*;
class ChangeElementsOfHashMap {
    public static void main(String args[])
    {

        // Initialization of a HashMap
        HashMap<Integer, String> hm
            = new HashMap<Integer, String>();

        // Change Value using put method
        hm.put(1, "Test");
        hm.put(2, "Test3");
        hm.put(3, "Test2");

        System.out.println("Initial Map " + hm);

        hm.put(2, "Test1");

        System.out.println("Updated Map " + hm);
    }
}

Output
Initial Map {1=Test, 2=Test3, 3=Test2}
Updated Map {1=Test, 2=Test1, 3=Test2}
```

3. Removing Element from Java HashMap

To remove an element from the Map, we can use the remove() method. This method takes the key value and removes the mapping for a key from this map if it is present in the map.

```
// Java program to remove
// elements from HashMap

import java.io.*;
import java.util.*;
class RemoveElementsOfHashMap {
    public static void main(String args[])
    {
        // Initialization of a HashMap
        Map<Integer, String> hm
            = new HashMap<Integer, String>();

        // Add elements using put method
        hm.put(1, "Test");
        hm.put(2, "Test1");
        hm.put(3, "Test2");
        hm.put(4, "Test3");

        // Initial HashMap
        System.out.println("Mappings of HashMap are : "
                           + hm);

        // remove element with a key
        // using remove method
        hm.remove(4);

        // Final HashMap
        System.out.println("Mappings after removal are : "
                           + hm);
    }
}

Output
Mappings of HashMap are : {1=Test, 2=Test1, 3=Test2, 4=Test3}
Mappings after removal are : {1=Test, 2=Test1, 3=Test2}
```

4. Traversal of Java HashMap

We can use the Iterator interface to traverse over any structure of the Collection Framework. Since Iterators work with one type of data we use Entry< ? , ? > to resolve the two separate types into a compatible format. Then using the next() method we print the entries of HashMap.

```
// Java program to traversal a
// Java.util.HashMap

import java.util.HashMap;
import java.util.Map;

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

        // Add elements using put method
        map.put("Test", 10);
        map.put("Test1", 30);
        map.put("Test2", 20);

        // Iterate the map using
        // for-each loop
        for (Map.Entry<String, Integer> e : map.entrySet())
            System.out.println("Key: " + e.getKey()
                               + " Value: " + e.getValue());
    }
}

Output
Key: Test2 Value: 20
Key: Test Value: 10
Key: Test1 Value: 30
```

Time and Space Complexity

Methods	Time Complexity	Space Complexity
Adding Elements in HashMap	O(1)	O(N)
Removing Element from HashMap	O(1)	O(N)
Extracting Element from HashMap	O(1)	O(N)

Methods in HashMap

K – The type of the keys in the map.

V – The type of values mapped in the map.

Methods	Description
clear()	Removes all of the mappings from this map.
clone()	Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
computeIfAbsent(K key, Function<?super K,? extends V> mappingFunction)	If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
containsKey(Object key)	Returns true if this map contains a mapping for the specified key.
containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
isEmpty()	Returns true if this map contains no key-value mappings.
entrySet()	Returns a Set view of the mappings contained in this map.
keySet()	Returns a Set view of the keys contained in this map.
merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associate it with the given non-null value.

put(K key, V value)	Associates the specified value with the specified key in this map.
putAll(Map<? extends K,? extends V> m)	Copies all of the mappings from the specified map to this map.
remove(Object key)	Removes the mapping for the specified key from this map if present.
size()	Returns the number of key-value mappings in this map.
values()	Returns a Collection view of the values contained in this map.

Methods inherited from class java.util.AbstractMap

Methods	Description
equals()	Compare the specified object with this map for equality.
hashCode()	Returns the hash code value for this map.
toString()	Returns a string representation of this map.

Methods inherited from interface java.util.Map

Methods	Description
equals()	Compare the specified object with this map for equality.
forEach(BiConsumer<? super K, ? super V> action)	Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
getOrDefault(Object key, V defaultValue)	Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
hashCode()	Returns the hash code value for this map.

putIfAbsent(K key, V value)	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
remove(Object key, Object value)	Removes the entry for the specified key only if it is currently mapped to the specified value.
replace(K key, V value)	Replaces the entry for the specified key only if it is currently mapped to some value.
replace(K key, V oldValue, V newValue)	Replaces the entry for the specified key only if currently mapped to the specified value.
replaceAll(BiFunction<? super K,? super V,? extends V> function)	Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

Advantages of HashMap

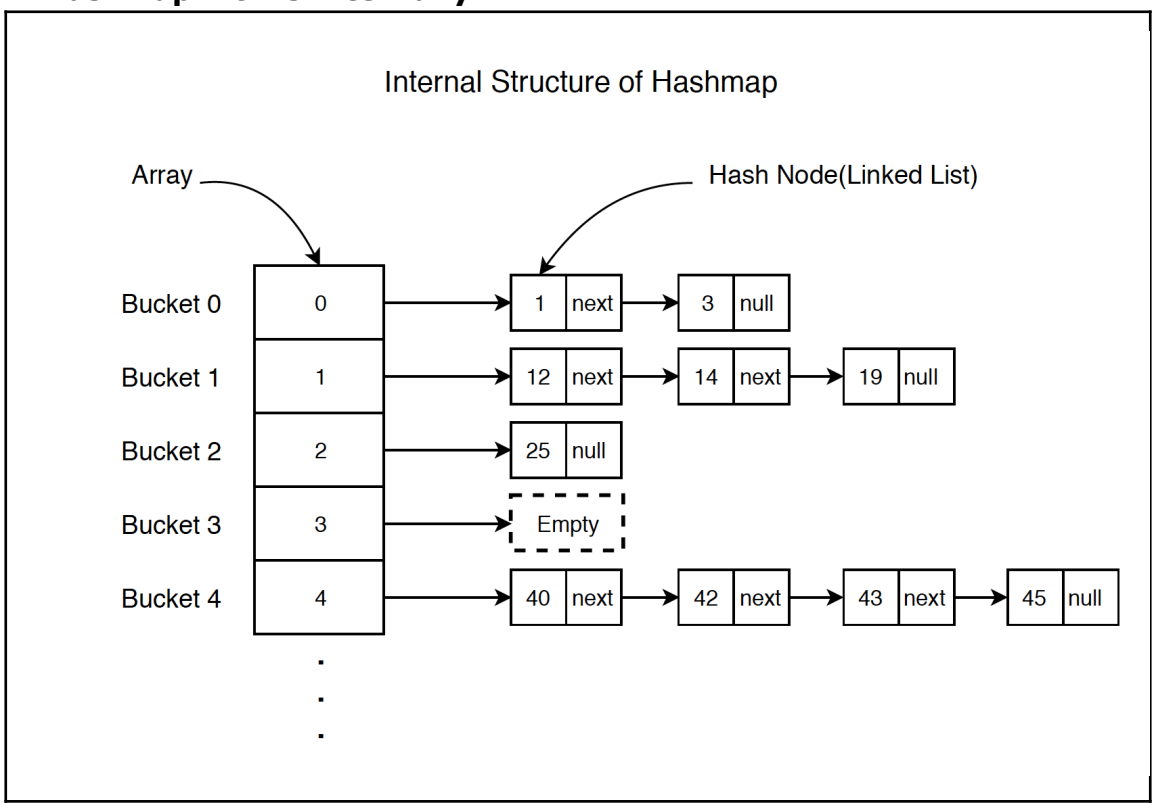
- Fast retrieval: HashMaps provide constant time access to elements, which means that retrieval and insertion of elements is very fast.
- Efficient storage: HashMaps use a hashing function to map keys to indices in an array. This allows for quick lookup of values based on keys, and efficient storage of data.
- Flexibility: HashMaps allow for null keys and values, and can store key-value pairs of any data type.
- Easy to use: HashMaps have a simple interface and can be easily implemented in Java.
- Suitable for large data sets: HashMaps can handle large data sets without slowing down.

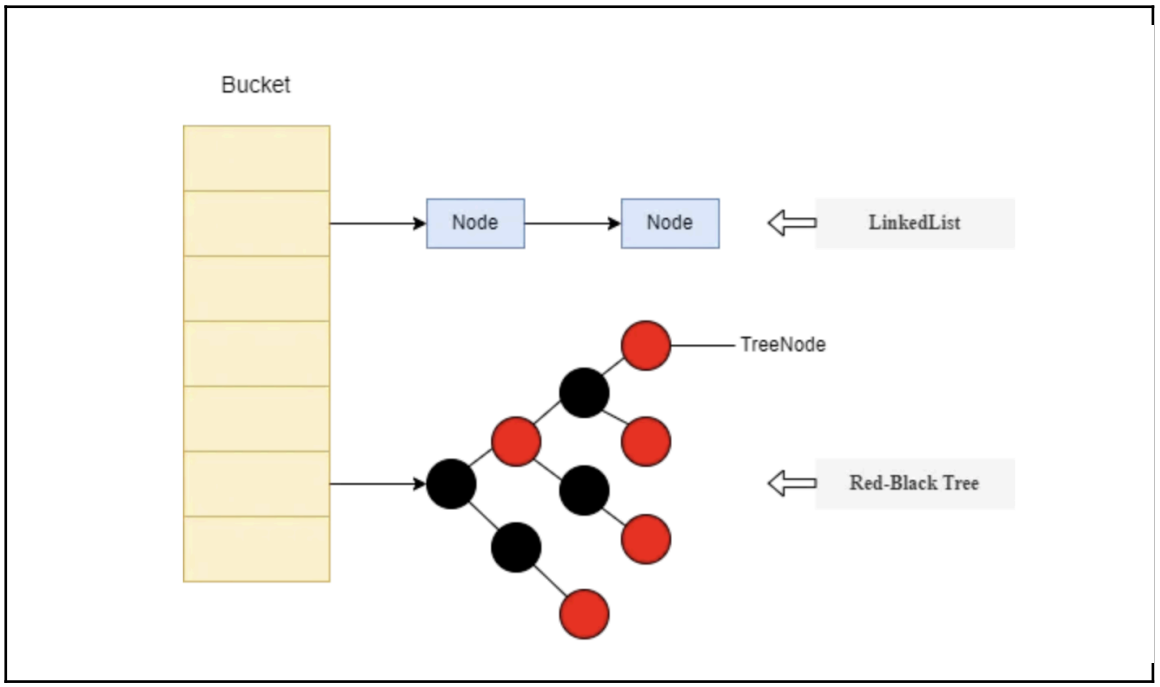
Disadvantages of HashMap

- Unordered: HashMaps are not ordered, which means that the order in which elements are added to the map is not preserved.
- Not thread-safe: HashMaps are not thread-safe, which means that if multiple threads access the same hashmap simultaneously, it can lead to data inconsistencies.
- Performance can degrade: In some cases, if the hashing function is not properly implemented or if the load factor is too high, the performance of a HashMap can degrade.
- More complex than arrays or lists: HashMaps can be more complex to understand and use than simple arrays or lists, especially for beginners.

- Higher memory usage: Since HashMaps use an underlying array, they can use more memory than other data structures like arrays or lists. This can be a disadvantage if memory usage is a concern.

How HashMap works internally:





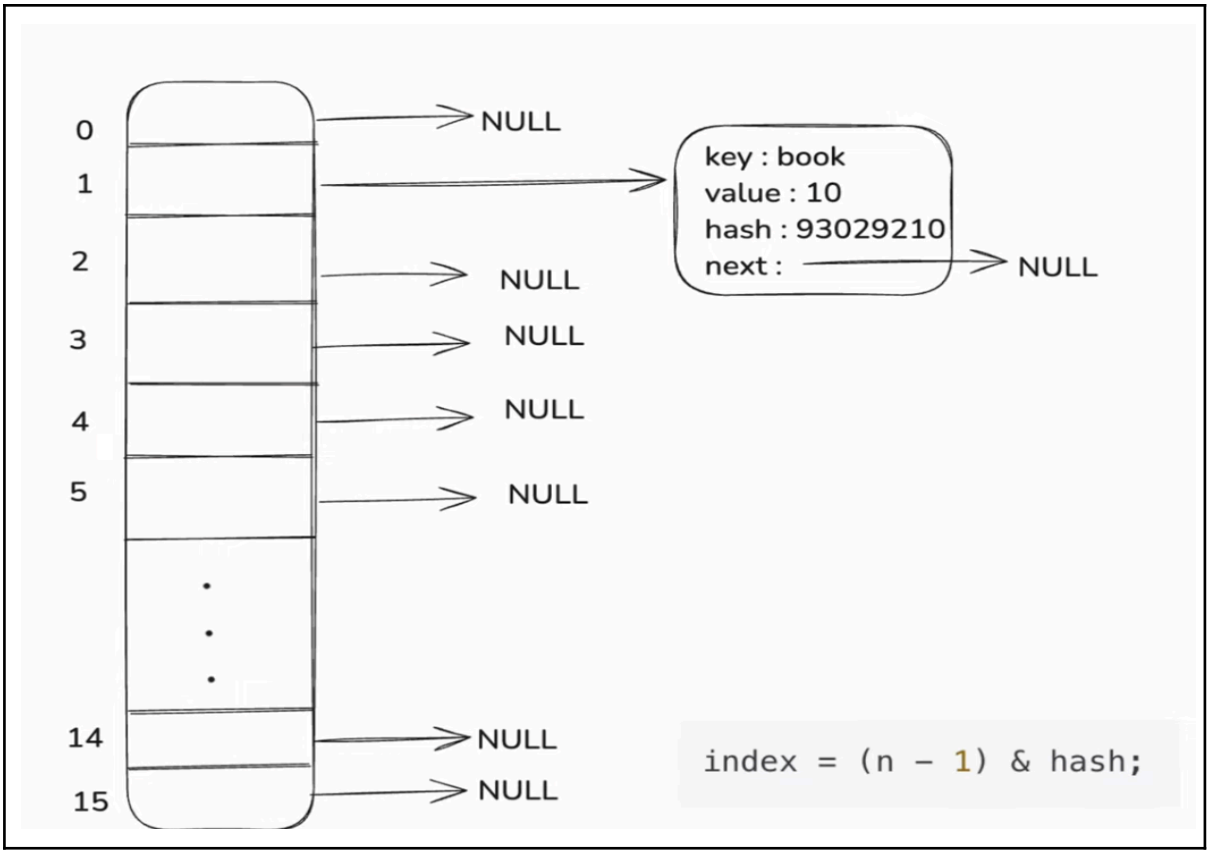
When Java creates new HashMap objects in memory:

- The following fields are initialized with default values.
    - capacity = 16
    - load factor = .75f
    - threshold = 12
- capacity * load factor = 16 * 75 = 12. So the table resizes when 12 entries are added.
-
- At this point no internal array is being created. To save memory java does not create an array when you create an instance of HashMap. This is called Lazy Initialization.

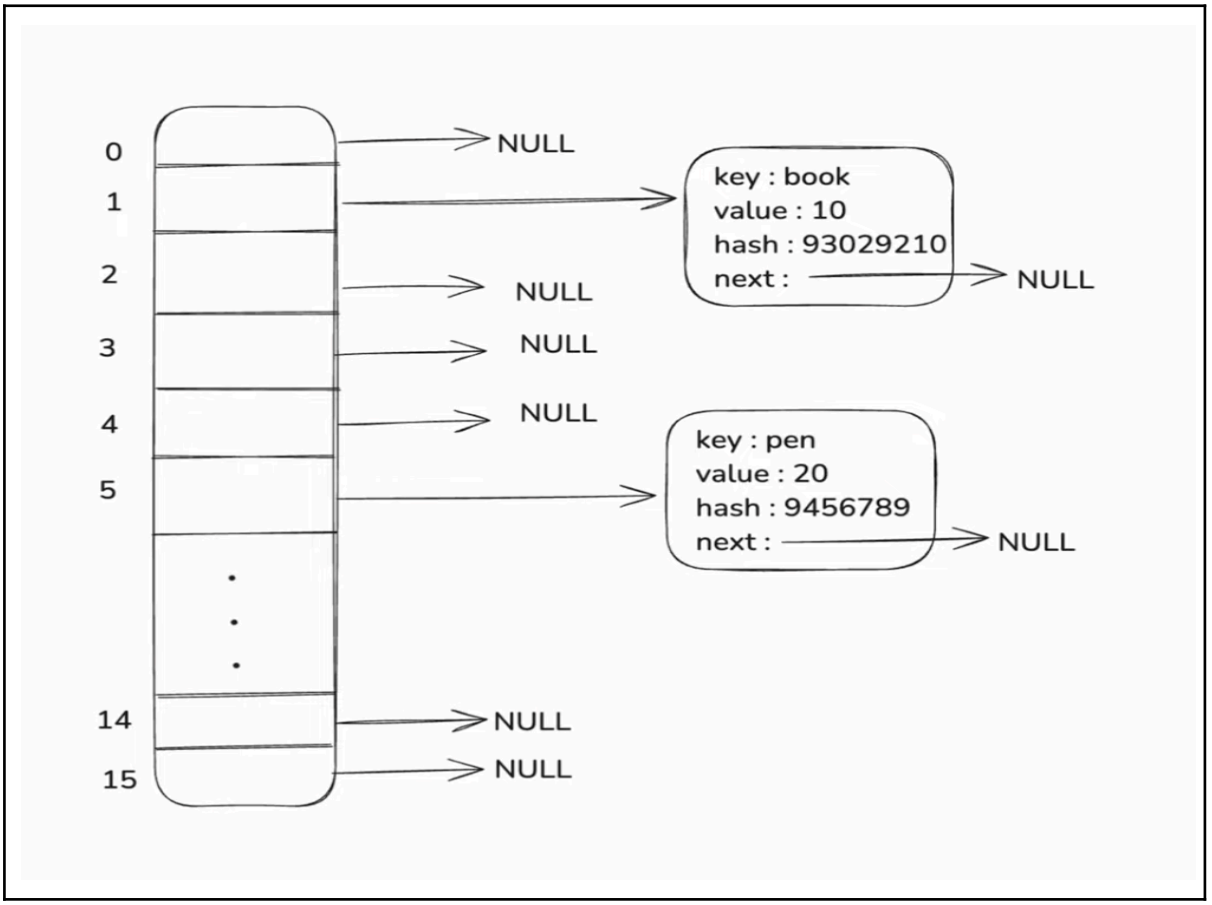
How put() method works:

productCount.put("book",10);

1. The array is initialized with the size of 16.
2. Compute hash code
   hashCode ("book") = 93029210
3. Calculate index
   index = (16-1) & 93029210
   index = 15 & 93029210
   index = 1

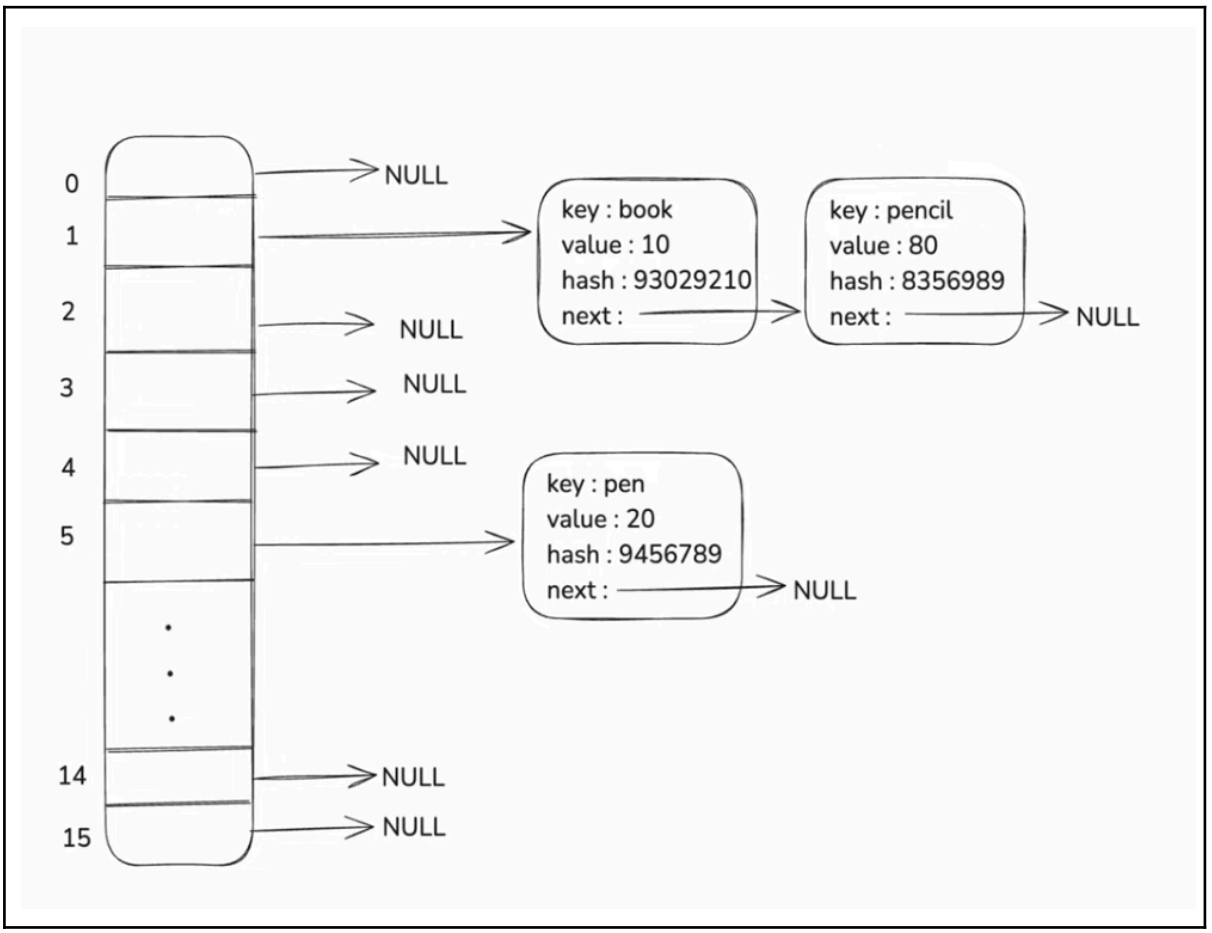


productCount.put("pen",20);



productCount.put("pencil", 80);

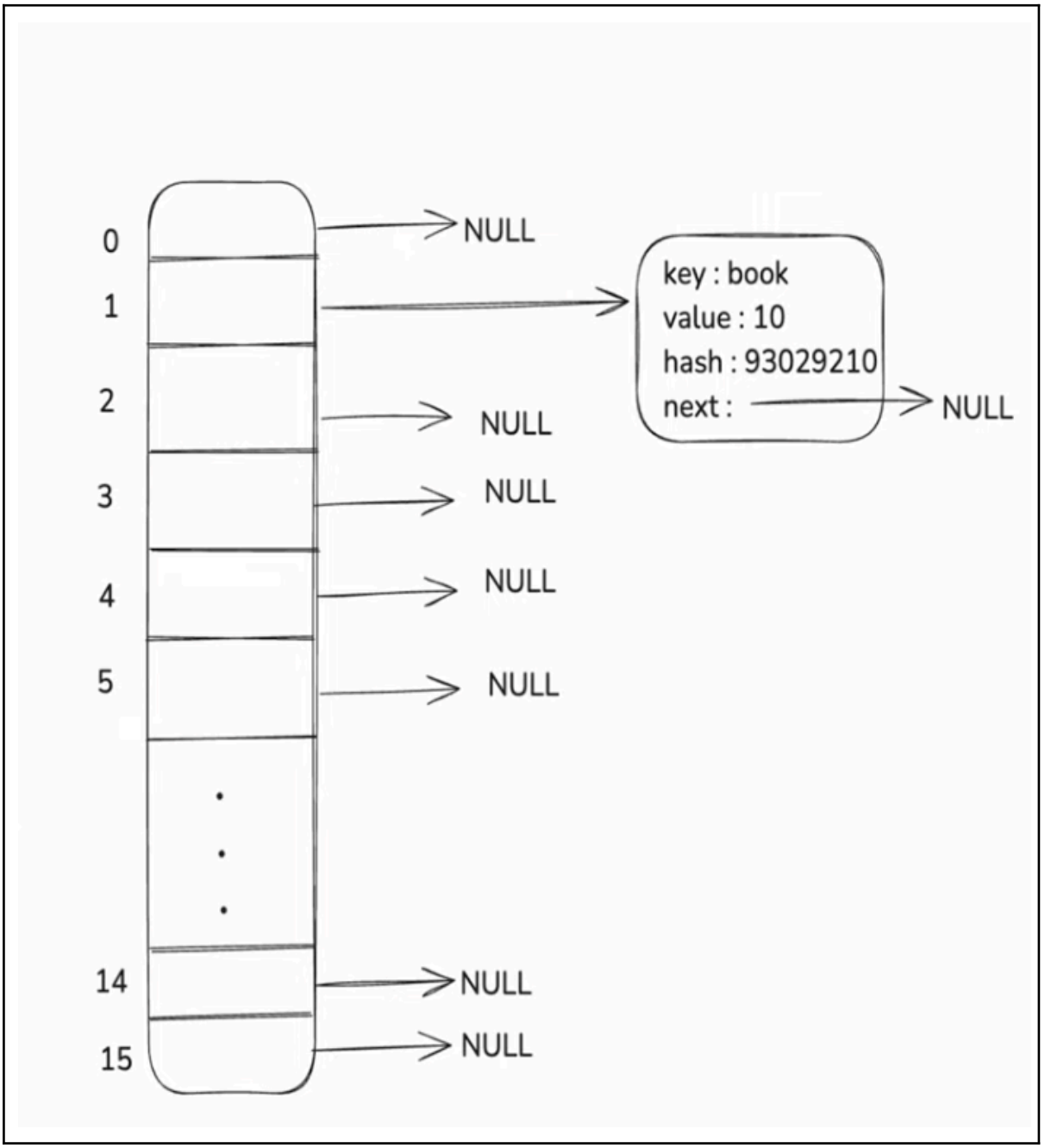
- Compute hash code
  - hashcode("pencil"= 8356989)
- Calculate index
  - Index = (16-1) & 8356989
  - Index = 1
- The key is new; it chains a new node at the end.



How get() method works internally:

productCount.get("book).

- Compute the hash code of the key.
- Calculate the index
- Go to the bucket.
- If no node is found, return null.
- If node is found:
    - node check if node key equals to key
    - If yes, return value.
    - If not, traverse the list/tree until a match is found.
- Return the value or return null if no match



Java LinkedHashMap

LinkedHashMap in Java implements the Map interface of the Collections Framework. It stores key-value pairs while maintaining the insertion order of the entries. It maintains the order in which elements are added.

- Stores unique key-value pairs.
- Maintains insertion order.
- Allows one null key and multiple null values.
- Fast performance for basic operations.

Example:

```
import java.util.LinkedHashMap;

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

        // Create a LinkedHashMap of
        // Strings (keys) and Integers (values)
        LinkedHashMap<String, Integer> lhm = new LinkedHashMap<>();

        // Displaying the LinkedHashMap
        System.out.println("" + lhm);
    }
}

Output
{}
```

Declaration of LinkedHashMap

```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>
```

Here, K is the key Object type and V is the value Object type

- K: The type of the keys in the map.
- V: The type of values mapped in the map.

The LinkedHashMap Class is just like HashMap with an 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.

Internal Working of LinkedHashMap

A LinkedHashMap is an extension of the HashMap class and it implements the Map interface. Therefore, the class is declared as:

```
public class LinkedHashMap
    extends HashMap
       implements Map
```

In this class, the data is stored in the form of nodes. The implementation of the LinkedHashMap is very similar to a doubly-linked list. Therefore, each node of the LinkedHashMap is represented as:

LinkedHashMap Node Representation in Java

- **Key:** Since this class extends HashMap, the data is stored in the form of a key-value pair. Therefore, this parameter is the key to the data.
- **Value:** For every key, there is a value associated with it. This parameter stores the value of the keys. Due to generics, this value can be of any form.
- **Next:** Since the LinkedHashMap stores the insertion order, this contains the address to the next node of the LinkedHashMap.
- **Previous:** This parameter contains the address to the previous node of the LinkedHashMap.

Synchronized LinkedHashMap

The LinkedHashMap class is not synchronized. If it is used in a multi-threaded environment where structural modifications like adding or removing elements are made concurrently then external synchronization is needed. This can be done by wrapping the map using Collections.synchronizedMap() method.

```
Map<K, V> synchronizedMap = Collections.synchronizedMap(new
LinkedHashMap<>());
```

Constructors of LinkedHashMap Class

LinkedHashMap class provides various constructors for different use cases:

1. LinkedHashMap(): This is used to construct a default LinkedHashMap constructor.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<>();
```

2. LinkedHashMap(int capacity): It is used to initialize a particular LinkedHashMap with a specified capacity.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<>(int capacity);
```

3. LinkedHashMap(Map<? extends K,? extends V> map): It is used to initialize a particular LinkedHashMap with the elements of the specified map.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<K, V>(Map<? extends K,?
extends V> map);
```

4. LinkedHashMap(int capacity, float fillRatio): It is used to initialize both the capacity and fill ratio for a LinkedHashMap. A fillRatio also called as loadFactor is a metric that determines when to increase the size of the LinkedHashMap automatically. By default, this value is 0.75 which means that the size of the map is increased when the map is 75% full.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<K, V>(int capacity, float
fillRatio);
```

5. LinkedHashMap(int capacity, float fillRatio, boolean Order): This constructor is also used to initialize both the capacity and fill ratio for a LinkedHashMap along with whether to follow the insertion order or not.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<K, V>(int capacity, float
fillRatio, boolean Order);
```

Here, for the Order attribute, true is passed for the last access order and false is passed for the insertion order.

Methods of LinkedHashMap

Below are some commonly used methods of the LinkedHashMap class:

Method	Description
containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
entrySet()	Returns a Set view of the mappings contained in this map.

get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
keySet()	Returns a Set view of the keys contained in this map.
removeEldestEntry(Map.Entry<K,V> eldest)	Returns true if this map should remove its oldest entry.
values() .	Returns a Collection view of the values contained in this map

Performing Various Operations on LinkedHashMap

Let's see how to perform a few frequently used operations on the LinkedHashMap class instance.

1. Adding Elements in LinkedHashMap

In order to add an element to the LinkedHashMap, we can use the put() method. This is different from HashMap because in HashMap, the insertion order is not retained but it is retained in the LinkedHashMap.

Example:

```
import java.util.*;

class Main {
    public static void main(String args[]) {
        LinkedHashMap<Integer, String> lhm
            = new LinkedHashMap<Integer, String>();

        // Add mappings to Map using put() method
        lhm.put(3, "Test3");
        lhm.put(2, "Test2");
        lhm.put(1, "Test1");

        // Printing mappings to the console
        System.out.println("" + lhm);
    }
}

Output
{3=Test3, 2=Test2, 1=Test1}
```

2. Updating Elements in LinkedHashMap

After adding elements, if we wish to change the element, it can be done by again adding the element using the put() method. Since the elements in the LinkedHashMap are indexed using the keys, the value of the key can be changed by simply re-inserting the updated value for the key for which we wish to change.

Example:

```
import java.util.*;

class Main {
    public static void main(String args[]) {
        LinkedHashMap<Integer, String> lhm
            = new LinkedHashMap<Integer, String>();
        lhm.put(3, "Test");
        lhm.put(2, "Test");
        lhm.put(1, "Test");

        // Printing mappings to the console
        System.out.println("" + lhm);

        // Updating the value with key 2
        lhm.put(2, "Test2");

        // Printing the updated Map
        System.out.println("Updated Map: " + lhm);
    }
}

Output
{3=Test, 2=Test, 1=Test}
Updated Map: {3=Test, 2=Test2, 1=Test}
```

3. Removing Element in LinkedHashMap

In order to remove an element from the LinkedHashMap, we can use the remove() method. This method takes the value of key as input, searches for the existence of such key and then removes the mapping for the key from this LinkedHashMap if it is present in the map. Apart from that, we can also remove the first entered element from the map if the maximum size is defined.

Example:

```
import java.util.*;

class Main {
    public static void main(String args[]) {
        LinkedHashMap<Integer, String> lhm
            = new LinkedHashMap<Integer, String>();
        lhm.put(3, "Test3");
        lhm.put(2, "Test2");
        lhm.put(1, "Test1");
        lhm.put(4, "Test4");

        // Printing the mappings to the console
        System.out.println("" + lhm);

        // Removing the mapping with Key 4
        lhm.remove(4);

        // Printing the updated map
        System.out.println("" + lhm);
    }
}

Output
{3=Test3, 2=Test2, 1=Test1, 4=Test4}
{3=Test3, 2=Test2, 1=Test1}
```

4. Iterating through the LinkedHashMap

There are multiple ways to iterate through the LinkedHashMap. The most famous way is to use a for-each loop over the set view of the map (fetched using map.entrySet() instance method). Then for each entry (set element) the values of key and value can be fetched using the getKey() and the getValue() method.

Example:

```
import java.util.*;

class Main {
    public static void main(String args[]) {
        LinkedHashMap<Integer, String> lhm
            = new LinkedHashMap<Integer, String>();

        lhm.put(3, "Test3");
        lhm.put(2, "Test2");
        lhm.put(1, "Test1");

        for (Map.Entry<Integer, String> mapElement :
             lhm.entrySet()) {
            Integer k = mapElement.getKey();
            String v = mapElement.getValue();

            System.out.println(k + " : " + v);
        }
    }
}

Output
3 : Test3
2 : Test2
1 : Test1
```

Advantages of LinkedHashMap

1. It maintains insertion order.
2. Faster iteration.
3. Allows null values.

Disadvantages of LinkedHashMap

1. Higher memory usage.
2. Slower insertion.
3. Less efficient for large datasets.

TreeMap in Java

TreeMap is a part of the Java Collection Framework. It implements the Map and NavigableMap interface and extends the AbstractMap class. It stores key-value pairs in a sorted order based on the natural ordering of keys or a custom Comparator. It uses a Red-Black Tree for efficient operations (add, remove, retrieve) with a time complexity of O(log n).

- The keys in a TreeMap are always sorted.
- Most operations, such as get, put, and remove have a time complexity of O(log n).
- TreeMap does not allow null as a key, it allows null as a value. Attempting to insert a null key will result in NullPointerException.
- TreeMap is not Synchronized. For thread-safe operations, we need to use Collections.synchronized map.
- Entry pairs returned by the methods in this class and their views represent snapshots of mappings at the time they were produced. They do not support the Entry.setValue method.

Let us start with a simple Java code snippet that demonstrates how to create and use a TreeMap in Java.

```
// Java Program to create a TreeMap
import java.util.Map;
import java.util.TreeMap;

public class TreeMapCreation {
    public static void main(String args[])
    {
        // Create a TreeMap of Strings
        // (keys) and Integers (values)
        TreeMap<String, Integer> tm = new TreeMap<>();

        System.out.println("TreeMap elements: " + tm);
    }
}

Output
TreeMap elements: {}
```

Constructors in TreeMap:
In order to create a TreeMap, we need to create an object of the TreeMap class. The TreeMap class consists of various constructors that allow the possible creation of the TreeMap. The following are the constructors available in this class:

Constructor	Description
TreeMap()	This constructor is used to build an empty TreeMap that will be sorted by using the natural order of its keys.
TreeMap(Comparator comp)	This constructor is used to build an empty TreeMap object in which the elements will need an external specification of the sorting order.
TreeMap(Map M)	This constructor is used to initialize a TreeMap with the entries from the given map M which will be sorted by using the natural order of the keys.
TreeMap(SortedMap sm)	This constructor is used to initialize a TreeMap with the entries from the given sorted map which will be stored in the same order as the given sorted map

Example 1: This example demonstrates how to create a TreeMap with the default constructor, adding elements and printing the sorted key-value pairs.

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        TreeMap<Integer, String> tm
            = new TreeMap<Integer, String>();

        tm.put(10, "Test1");
        tm.put(15, "Test3");
        tm.put(20, "Test3");
        System.out.println("TreeMap: " + tm);
    }
}

Output
TreeMap: {10=Test1, 15=Test2, 20=Test3}
```

Example 2: This example demonstrates using the TreeMap(Comparator comp) constructor to sort Student Objects by their roll number with a custom Comparator.

```
import java.util.*;

class Student {
    int rollNo;
    String name, address;

    public Student(int rollNo, String name, String address) {
        this.rollno = rollNo;
        this.name = name;
        this.address = address;
    }

    public String toString() {
        return this.rollno + " " + this.name + " " + this.address;
    }
}

class SortByRoll implements Comparator<Student> {
    public int compare(Student a, Student b) {
        return a.rollno - b.rollno;
    }
}

public class Main {
    public static void main(String[] args) {
        TreeMap<Student, Integer> tm = new TreeMap<>(new SortByRoll());

        tm.put(new Student(111, "Test1", "New York"), 1);
        tm.put(new Student(131, "Test2", "London"), 2);
        tm.put(new Student(121, "Test3", "Paris"), 3);

        System.out.println("TreeMap sorted by roll number: " + tm);
    }
}

Output
TreeMap sorted by roll number: {111 Test1 New York=1, 121 Test3 Paris=3,
131 Test2 London=2}
```

Example 3: This example demonstrates using the TreeMap(Map M) constructor to create a TreeMap from an existing HashMap.

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

public class Main {
    public static void main(String[] args) {
        Map<Integer, String> m
                = new HashMap<Integer, String>();

        m.put(10, "Test1");
        m.put(20, "Test2");
        m.put(30, "Test3");

        TreeMap<Integer, String> tm
                = new TreeMap<Integer, String>(m);

        System.out.println("TreeMap: " + tm);
    }
}

Output
TreeMap: {10=Test1, 20=Test2, 30=Test3}
```

Example 4: Demonstrating TreeMap using the SortedMap Constructor.

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

public class Main {
    public static void main(String[] args) {
        SortedMap<Integer, String> sm
                = new ConcurrentSkipListMap<Integer,
                String>();

        sm.put(10, "Test1");
        sm.put(15, "Test2");
        sm.put(20, "Test3");
        sm.put(25, "Test4");
        sm.put(30, "Test5");
        TreeMap<Integer, String> tm = new TreeMap<Integer, String>(sm);
        System.out.println("TreeMap: " + tm);
    }
}
Output
TreeMap: {10=Test1, 15=Test2, 20=Test3, 25=Test4, 30=Test5}
```

Performing Various Operations on TreeMap

1. Adding Elements: We can use the put() method to insert elements to a TreeMap. However, the insertion order is not retained in the TreeMap. Internally, for every element, the keys are compared and sorted in ascending order.

Example: This example demonstrates how to use the put() method to insert elements into a TreeMap both with and without generics.

```
// Java Program to insert elements into a TreeMap
import java.util.*;

class Main {
    public static void main(String args[])  {
        TreeMap<Integer, String> tm = new TreeMap<>();
        tm.put(3, "Test3");
        tm.put(2, "Test2");
        tm.put(1, "Test1");

        System.out.println("TreeMap with raw type: " + tm);

        TreeMap<Integer, String> tm1 = new TreeMap<>();
        tm1.put(3, "Language");
        tm1.put(2, "Programming");
        tm1.put(1, "Java");

        System.out.println("TreeMap with generics: " + tm1);
    }
}

Output
TreeMap with raw type: {1=Test1, 2=Test2, 3=Test3}
TreeMap with generics: {1=Java, 2=Programming, 3=Language}
```

2. Changing Elements: To change the element in a TreeMap, simply use the put() method again with the same key and the new value.

Example: This example demonstrates how to update an existing value in a TreeMap by using the put() method with an existing key.

```
import java.util.*;

class Main {
    public static void main(String args[]) {
        TreeMap<Integer, String> tm = new TreeMap<Integer, String>();

        tm.put(3, "Test3");
        tm.put(2, "Test2");
        tm.put(1, "Test1");
        System.out.println(tm);

        tm.put(2, "Test4");
        System.out.println(tm);
    }
}

Output
{1=Test1, 2=Test2, 3=Test3}
{1=Test1, 2=Test4, 3=Test3}
```

3. Removing Element: We can use the remove() method to remove elements from the TreeMap.

Example: This example demonstrates how to remove an element from a TreeMap using the remove() method.

```
import java.util.*;

class Main {
    public static void main(String args[]) {
        TreeMap<Integer, String> tm = new TreeMap<Integer, String>();
        tm.put(3, "Java");
        tm.put(2, "C++");
        tm.put(1, "Pyhton");
        tm.put(4, "JS");
        System.out.println(tm);
        tm.remove(4);
        System.out.println(tm);
    }
}
```

```
Output
{1=Pyhton, 2=C++, 3=Java, 4=JS}
{1=Pyhton, 2=C++, 3=Java}
```

4. Iterating Elements: There are multiple ways to iterate through the Map. The most famous way is to use a for-each loop and get the keys. The value of the key is found by using the getValue() method.

Example: This example demonstrates iterating over a TreeMap using a for-each loop to access and print the key-value pairs.

```
import java.util.*;

class Main {
    public static void main(String args[]) {
        TreeMap<Integer, String> tm = new TreeMap<>();

        tm.put(3, "Test3");
        tm.put(2, "Test2");
        tm.put(1, "Test1");

        // For-each loop for traversal over entrySet()
        for (Map.Entry<Integer, String> e : tm.entrySet()) {
            int k = e.getKey();
            String v = e.getValue();

            // Printing the key and value
            System.out.println(k + " : " + v);
        }
    }
}

Output
1 : Test1
2 : Test2
3 : Test3
```

Methods in TreeMap:

Method	Action Performed
clear()	The method removes all mappings from this TreeMap and clears the map.
clone()	The method returns a shallow copy of this TreeMap.
containsKey(Object key)	Returns true if this map contains a mapping for the specified key.
containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
entrySet()	Returns a set view of the mappings contained in this map.
firstKey()	Returns the first (lowest) key currently in this sorted map.
get(Object key)	Returns the value to which this map maps the specified key.
headMap(Object key_value)	The method returns a view of the portion of the map strictly less than the parameter key_value.
keySet()	The method returns a Set view of the keys contained in the treemap.
lastKey()	Returns the last (highest) key currently in this sorted map.
put(Object key, Object value)	The method is used to insert a mapping into a map.
putAll(Map map)	Copies all of the mappings from the specified map to this map.
remove(Object key)	Removes the mapping for this key from this TreeMap if present.
size()	Returns the number of key-value mappings in this map.
subMap((K startKey, K endKey)	The method returns the portion of this map whose keys range from startKey, inclusive, to endKey, exclusive.
values()	Returns a collection view of the values contained in this map.

Advantages

- The TreeMap provides a sorted order of its elements, based on the natural order of its keys or a custom Comparator passed to the constructor. This makes it useful in situations where you need to retrieve elements in a specific order.
- Because the elements in a TreeMap are stored in a sorted order, you can predict the order in which they will be returned during iteration, making it easier to write algorithms that process the elements in a specific order.
- The TreeMap provides an efficient implementation of the Map interface, allowing you to retrieve elements in logarithmic time, making it useful in search algorithms where you need to retrieve elements quickly.
- The TreeMap is implemented using a Red-Black tree, which is a type of self-balancing binary search tree. This provides efficient performance for adding, removing, and retrieving elements, as well as maintaining the sorted order of the elements.

Disadvantages

- Inserting elements into a TreeMap can be slower than inserting elements into a regular Map, as the TreeMap needs to maintain the sorted order of its elements.
- The keys in a TreeMap must implement the java.lang.Comparable interface, or a custom Comparator must be provided. This can be a restriction if you need to use custom keys that do not implement this interface.

Cursors interfaces of Collection (Enumeration, Iterator, ListIterator)

If we want to get Objects One by One from the Collection then we should go for Cursors. There are 3 Types of Cursors Available in Java.

1. Enumeration
2. Iterator
3. ListIterator

Enumeration

- We can Use Enumeration to get Objects One by One from the Collection.
- We can Create Enumeration Objects by using elements().

Interface definition:

```
public interface Enumeration<E>
```

Important Features

- Enumeration is Synchronized.
- It does not support adding, removing, or replacing elements.
- Elements of legacy Collections can be accessed in a forward direction using Enumeration.
- Legacy classes have methods to work with enumeration and return Enumeration objects.

Example:

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationClass {
    public static void main(String args[]) {
        Enumeration months;
        Vector<String> monthNames = new Vector<>();

        monthNames.add("January");
        monthNames.add("February");
        monthNames.add("March");
        monthNames.add("April");
        monthNames.add("May");
        monthNames.add("June");
        monthNames.add("July");
        monthNames.add("August");
        monthNames.add("September");
        monthNames.add("October");
```

```
        monthNames.add("November");
        monthNames.add("December");
        months = monthNames.elements();

        while (months.hasMoreElements()) {
            System.out.println(months.nextElement());
        }
    }
}

Output
January
February
March
April
May
June
July
August
September
October
November
December
```

Java Enumeration Interface With SequenceInputStream

```
import java.io.*;
import java.util.*;
class Main {
    public static void main(String args[]) throws IOException {
        // creating the FileInputStream objects for all the files
        FileInputStream fin = new FileInputStream("file1.txt");
        FileInputStream fin2 = new FileInputStream("file2.txt");
        FileInputStream fin3 = new FileInputStream("file3.txt");
        FileInputStream fin4 = new FileInputStream("file4.txt");

        // creating Vector object to all the stream
        Vector v = new Vector();
        v.add(fin);
        v.add(fin2);
        v.add(fin3);
        v.add(fin4);
```

```
        // creating enumeration object by calling the elements method
        Enumeration e = v.elements();

        // passing the enumeration object in the constructor
        SequenceInputStream bin = new SequenceInputStream(e);

        int i = 0;
        while ((i = bin.read()) != -1) {
            System.out.print((char)i);
        }

        bin.close();
        fin.close();
        fin2.close();
    }
}
```

Creation Of Custom Enumeration

```
import java.util.Enumeration;
import java.lang.reflect.Array;

public class EnumerationClass implements Enumeration {
    private int size;
    private int cursor;
    private final Object array;
    public EnumerationClass(Object obj)
    {
        Class type = obj.getClass();
        if (!type.isArray()) {
            throw new IllegalArgumentException(
                "Invalid type: " + type);
        }
        size = Array.getLength(obj);
        array = obj;
    }
    public boolean hasMoreElements()
    {
        return (cursor < size);
    }
    public object nextElements()
    {
        return Array.get(array, cursor++);
    }
}
```

Creation of Java Enumeration using String Array

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

public class EnumerationExample {
    public static void main(String args[])
    {
        // String Array Creation
        String str[] = { "apple", "facebook", "google" };

        // Array Enumeration Creation
        ArrayEnumeration aenum = new ArrayEnumeration(str);

        // usage of array enumeration
        while (aenum.hasMoreElements()) {
            System.out.println(aenum.nextElement());
        }
    }
}
```

Methods Of Enumeration Interface

- E - type of elements

Modifier And Return Type	Method	Explanation
default Iterator<E>	asIterator()	This method returns an Iterator which traverses all the remaining elements covered by this enumeration.
boolean	hasMoreElements()	On implementation, it returns the boolean value if there are more elements to extract or not and returns false when all the elements have been enumerated.
E	nextElement()	This method returns the next element of the enumeration. It throws NoSuchElementExceptio n when there are no more elements.

Limitations of Enumeration:

- Enumeration Concept is Applicable Only for Legacy Classes and it is not a universal Cursor.
- By using Enumeration we can Perform Read Operation and we can't Perform Remove Operation.

Iterator

- We can Use Iterator to get Objects One by One from Collection.
- We can Apply Iterator Concept for any Collection Object. Hence it is a Universal Cursor.
- By using Iterator we can Able to Perform Both Read and Remove Operations.
- We can create an Iterator Object by using iterator() of Collection Interface.

Interface definition:

```
public interface Iterator<E>
```

Example:

```
import java.util.*;

class IteratorDemo {
    public static void main(String[] args) {
        ArrayList l = new ArrayList();

        for (int i=0; i<=10; i++) {
             l.add(i);
             System.out.println(l);
        }

        Iterator itr = l.iterator();

        while(itr.hasNext()) {
            Integer I = (Integer)itr.next();
            if(I%2 == 0)
                System.out.println(I);
            else
                itr.remove();
            System.out.println(l);
        }
    }
}
```

Limitations:

- By using Enumeration and Iterator we can Move Only towards Forward Direction and we can’t Move Backward Direction. That is, these are Single Direction Cursors but NotBi- Direction.
- By using Iterator we can Perform Only Read and Remove Operations and we can't Perform Addition of New Objects and Replacing Existing Objects.

ListIterator:

- ListIterator is the Child Interface of Iterator.
- By using ListIterator we can Move Either to the Forward Direction OR to the Backward Direction. That is it is a Bi-Directional Cursor.
- By using ListIterator we can Able to Perform Addition of New Objects and Replacing existing Objects. In Addition to Read and Remove Operations.
- We can Create ListIterator Objects by using listIterator().

Interface definition:

```
public interface ListIterator<E> extends Iterator<E>
```

Some Important points about ListIterator

- It is useful for list implemented classes.
- Available since java 1.2.
- It supports bi-directional traversal. i.e both forward and backward directions.
- It supports all the four CRUD operations(Create, Read, Update, Delete) operations.

Methods of ListIterator

Method	Description
add(E e)	This method inserts the specified element into the list.
hasNext()	This returns true if the list has more elements to traverse.
hasPrevious()	This returns true if the list iterator has more elements while traversing the list in the backward direction.
next()	This method returns the next element and increases the cursor by one position.
nextIndex()	This method returns the index of the element which would be returned on calling the next() method.
previous()	This method returns the previous element of the list and shifts the cursor one position backward

previousIndex()	This method returns the index of the element which would be returned on calling the previous() method.
remove()	This method removes the last element from the list that was returned on calling next() or previous() method element from.
set(E e)	This method replaces the last element that was returned on calling next() or previous() method with the specified element.

Methods declared in interface java. util.Iterator

Method	Description
default void forEachRemaining(Consumer<? super E> action)	Performs the given action for each remaining element until all elements have been processed or the action throws an exception.

Example:

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> names = new LinkedList<>();
        names.add("Learn");
        names.add("from");
        names.add("pathshala360");

        ListIterator<String> listIterator = names.listIterator();

        System.out.println("Forward Direction Iteration:");
        while (listIterator.hasNext()) {
            System.out.print(listIterator.next() + " ");
        }
        System.out.println();
        System.out.println("Backward Direction Iteration:");
        while (listIterator.hasPrevious()) {
            System.out.print(listIterator.previous() + " ");
        }
        System.out.println();
    }
}
Output:
Forward Direction Iteration:
Learn from pathshala36
Backward Direction Iteration:
pathshala36 from Learn
```

Sorting interfaces of Collection (Comparable, Comparator)

Java Comparator Interface
The Comparator interface in Java is used to sort the objects of user-defined classes. The Comparator interface is present in the java.util package. This interface allows us to define custom comparison logic outside of the class for which instances we want to sort. The comparator interface is useful when,

- We need multiple sorting strategies for a class.
- When we want to keep the sorting logic separate from the class.

A comparator object is capable of comparing two objects of the same class. The following function compares obj1 with obj2.

Syntax:

```
public int compare(Object obj1, Object obj2);
```

- It will return a negative integer if obj1 < obj2.
- It will return 0 if both objects are equal.
- It will return a positive integer if obj1 > obj2.

When to Use a Comparator?
Suppose we have a list of Student objects, containing fields like roll no, name, address, DOB, etc, and we need to sort them based on roll no or name. We use Comparator here to define that logic separately.

Methods to Implement Comparator Interface

- **Method 1:** One obvious approach is to write our sort() function using one of the standard algorithms. This solution requires rewriting the whole sorting code for different criteria, like Roll No. and Name.
- **Method 2:** Using comparator interface: The Comparator interface is used to order the objects of a user-defined class. This interface contains 2 methods that are, compare(Object obj1, Object obj2) and equals(Object element). Using a comparator, we can sort the elements based on data members. For instance, it may be on roll no, name, age, or anything else.

How does the sort() Method of the Collections Class Work?
The sort() method of the Collections class is used to sort the elements of a List by the given comparator.

```
public void sort(List list, ComparatorClass c)
```

To sort a given List, ComparatorClass must implement a Comparator interface. Internally the sort() method does call the Compare method of the classes it is sorting. To compare two elements, it asks “Which is greater?” Compare method

returns -1, 0, or 1 to say if it is less than, equal, or greater to the other. It uses this result to then determine if they should be swapped for their sort.

Sort Collections by One Field

Example: Sorting By Roll Number

```
import java.util.*;

class Student {
    int rollNo;
    String name;

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

    @Override
    public String toString() {
        return rollNo + ": " + name;
    }
}

class SortbyRoll implements Comparator<Student> {
    public int compare(Student a, Student b) {
        return a.rollno - b.rollno;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();

        students.add(new Student(111, "Test1"));
        students.add(new Student(131, "test3"));
        students.add(new Student(121, "Test2"));
        students.add(new Student(101, "Test0"));

        Collections.sort(students, new SortbyRoll());
        System.out.println("Sorted by Roll Number ");

        for (int i = 0; i < students.size(); i++)
            System.out.println(students.get(i));

    }
}
```

```
Output
Sorted by Roll Number
101: Aggarwal
111: Mayank
121: Solanki
131: Anshul
```

By changing the return value inside the compare method, you can sort in any order that you wish to. For example: For descending order just change the positions of "a" and "b" in the above compare method.

Note: We can use lambda expression in place of helper function by following the statement as mentioned below:

```
students.sort((p1, p2) -> Integer.compare(p1.age, p2.age));
```

Sort Collection by More than One Field

In the previous example, we have discussed how to sort the list of objects on the basis of a single field using the Comparable and Comparator interface But, what if we have a requirement to sort ArrayList objects in accordance with more than one field like firstly, sort according to the student name and secondly, sort according to student age.

Example: Sorting By Multiple Fields (Name, then Age)

```
import java.util.*;

class Student {
    String name;
    Integer age;

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

    public String getName() {
        return name;
    }

    public Integer getAge() {
        return age;
    }
```

```
    @Override
    public String toString() {
        return name + " : " + age;
    }
}

class CustomerSortingComparator implements Comparator<Student> {
    // Compare first by name, then by age
    public int compare(Student customer1, Student customer2) {
        // Compare by name first
        int NameCompare = customer1.getName().compareTo(
                                customer2.getName());

        // If names are the same, compare by age
        int AgeCompare = customer1.getAge().compareTo(
                                customer2.getAge());

        // Return the result: first by name, second by age
        return (NameCompare == 0) ? AgeCompare : NameCompare;
    }
}

public class ComparatorHelperClassExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();

        students.add(new Student("Ajay", 27));
        students.add(new Student("Sneha", 23));
        students.add(new Student("Simran", 37));
        students.add(new Student("Ankit", 22));
        students.add(new Student("Anshul", 29));
        students.add(new Student("Sneha", 22));

        // Original List
        System.out.println("Original List ");

        // Iterating List
        for (Student it : students) {
            System.out.println(it);
        }

        System.out.println();
```

```
        Collections.sort(students, new CustomerSortingComparator());

        // Display message only
        System.out.println("After Sorting ");
        for (Student it : students) {
            System.out.println(it);
        }
    }
}

Output
Original List
Ajay : 27
Sneha : 23
Simran : 37
Ankit : 22
Anshul : 29
Sneha : 22

After Sorting
Ajay : 27
Ankit : 22
Anshul : 29
Simran : 37
Sneha : 22
Sneha : 23
```

Alternative Method: Using Comparator with Lambda
Java 8 introduced a more simple way to write comparators using lambda expressions. We can use the method mentioned below for achieving same result:

```
students.sort(Comparator.comparing(Student::getName).thenComparing(Stude
nt::getAge));
```

Example:

```
import java.util.*;

class Student {
    String name;
    Integer age;

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

    public String getName() {
        return name;
    }

    public Integer getAge() {
        return age;
    }

    @Override
    public String toString() {
        return name + " : " + age;
    }
}

public class ComparatorHelperClassExample {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();

        students.add(new Student("Ajay", 27));
        students.add(new Student("Sneha", 23));
        students.add(new Student("Simran", 37));
        students.add(new Student("Ankit", 22));
        students.add(new Student("Anshul", 29));
        students.add(new Student("Sneha", 22));

        // Original List
        System.out.println("Original List:");

        // Iterating List
        for (Student it : students) {
            System.out.println(it);
        }
        System.out.println();
```

```
        // Sort students by name, then by age
students.sort(Comparator.comparing(Student::getName).thenComparing(Stude
nt::getAge));

        // Display message after sorting
        System.out.println("After Sorting:");

        // Iterating using enhanced for-loop after sorting ArrayList
        for (Student it : students) {
            System.out.println(it);
        }
    }
}

Output
Original List:
Ajay : 27
Sneha : 23
Simran : 37
Ankit : 22
Anshul : 29
Sneha : 22

After Sorting:
Ajay : 27
Ankit : 22
Anshul : 29
Simran : 37
Sneha : 22
Sneha : 23
```

Java Comparable Interface

The Comparable interface in Java is used to define the natural ordering of objects for a user-defined class. It is part of the java.lang package and it provides a compareTo() method to compare instances of the class. A class has to implement a Comparable interface to define its natural ordering.

Example 1: Here, we will use the Comparable interface to sort integers.

```
import java.util.*;

class Number implements Comparable<Number> {
    int v;

    public Number(int v) {
        this.v = v;
    }

    @Override
    public String toString() {
        return String.valueOf(v);
    }

    @Override
    public int compareTo(Number o) {
        return this.v - o.v;
    }

    public static void main(String[] args) {
        Number[] n = { new Number(4), new Number(1),
                       new Number(7), new Number(2) };

        System.out.println("Before Sorting: " + Arrays.toString(n));
        Arrays.sort(n);
        System.out.println("After Sorting: " + Arrays.toString(n));
    }
}

Output
Before Sorting: [4, 1, 7, 2]
After Sorting: [1, 2, 4, 7]
```

Explanation: In the above example, the compareTo() method is overridden to define the ascending order logic by comparing the v fields of Number objects. Then the Arrays.sort() method sorts the array by using this logic.

Declaration of Comparable Interface

```
public interface Comparable<T> {
    int compareTo(T obj);
}
```

where, T is the type of object which to be compared.

- It compares the current object with the specified object.
- It returns:
    - Negative, if currentObj < specifiedObj.
    - Zero, if currentObj == specifiedObj.
    - Positive, if currentObj > specifiedobj.

Use of Comparable Interface

- In this method, we are going to implement the Comparable interface from java.lang Package in the Pair class.
- The Comparable interface contains the method compareTo to decide the order of the elements.
- Override the compareTo method in the Pair class.
- Create an array of Pairs and populate the array.
- Use the Arrays.sort() function to sort the array.

Example 2: Sorting Pairs with String and Integer Fields
Given an array of Pairs consisting of two fields of type string and integer. Now, we have to sort the array in ascending Lexicographical order and if two strings are the same, sort it based on their integer value.

```
import java.util.*;

class Pair implements Comparable<Pair> {
    String s; // String
    int v;    // Integer

    public Pair(String s, int v) {
        this.s = s;
        this.v = v;
    }

    @Override
    public String toString() {
        return "(" + s + ", " + v + ")";
    }

    @Override
    public int compareTo(Pair p) {
        if (this.s.compareTo(p.s) != 0) {
            return this.s.compareTo(p.s);
        }
        return this.v - p.v;
    }
```

```
    public static void main(String[] args) {
        Pair[] p = {
            new Pair("abc", 3),
            new Pair("a", 4),
            new Pair("bc", 5),
            new Pair("a", 2)
        };

        System.out.println("Before Sorting:");
        for (Pair p1 : p) {
            System.out.println(p1);
        }

        // Sort the array of pairs
        Arrays.sort(p);

        System.out.println("\nAfter Sorting:");
        for (Pair p1 : p) {
            System.out.println(p1);
        }
    }
}

Output
Before Sorting:
(abc, 3)
(a, 4)
(bc, 5)
(a, 2)

After Sorting:
(a, 2)
(a, 4)
(abc, 3)
(bc, 5)
```

Note: if two strings are the same then the comparison is done based on the value.