Today’s Agenda 17-04-2024

|  |
| --- |
| **Java Collections Framework and Generics** |
| Arrays: Declaration, Initialization, and Usage |
| Introduction to List interface. |
| Exploring ArrayList and LinkedList. |
| Understanding the Set interface. |
| Working with HashSet, LinkedHashSet, and TreeSet. |
| Differentiating between various Set implementations. |
| Introduction to Map interface.// 19-03-2024 |
| Exploring HashMap, LinkedHashMap, and TreeMap. |
| Key-value pair management and use cases. |
| Iterators and Comparators |

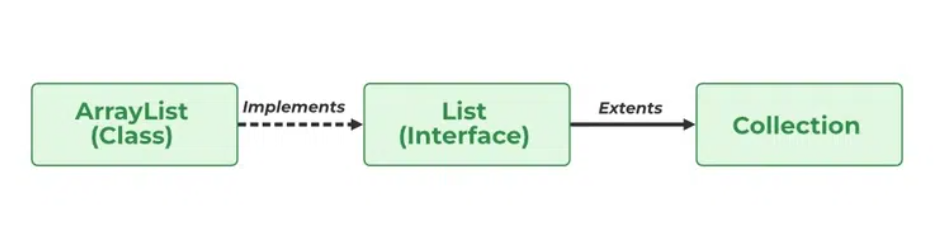
* Arrays: Declaration, Initialization, and Usage:

How to Initialize an Array in Java?

DONE IN SECOND CLASS.

* Introduction to List interface.

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



Declaration of Java List Interface:

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

List is an ‘interface’, implemented by the ArrayList class, pre-defined in java.util package.

Int,char,double, Boolean….wapper….Interger,Character,Boolean

Syntax of Java List

This type of safelist can be defined as:

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

Obj is the type of the object to be stored in List.

import java.util.\*;

class ListDemo {

public static void main(String[] args)

{

// Creating an object of List interface

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

l1.add(0, 1);

l1.add(1, 2);

System.out.println(l1);

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

l2.add(1);

l2.add(2);

l2.add(3);

// Will add list l2 from 1 index

l1.addAll(1, l2);

System.out.println(l1);

// Removes element from index 1

l1.remove(1);

// Printing the updated List 1

System.out.println(l1);

// Prints element at index 3 in list 1

System.out.println(l1.get(3));

// Replace 0th element with 5

// in List 1

l1.set(0, 5);

System.out.println(l1);

}

for(Integer i : list1) {

System.***out***.print(i+ " dfgh");

}

list1.forEach(x->System.***out***.println(x));

}

Operations in a Java List Interface

Since List is an interface, it can be used only with a class that implements this interface. Now, let’s see how to perform a few frequently used operations on the List.

Operation 1: Adding elements to List class using add() method

Operation 2: Updating elements in List class using set() method

Operation 3: Searching for elements using indexOf(), lastIndexOf methods

Operation 4: Removing elements using remove() method

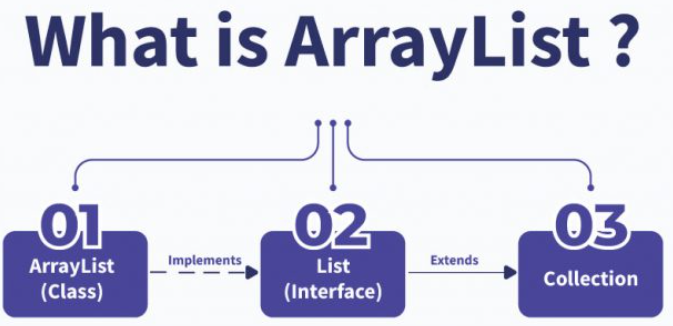
Operation 5: Accessing Elements in List class using get() method

Operation 6: Checking if an element is present in the List class using contains() method

Exploring ArrayList and LinkedList.

ArrayList uses an array, which allows for fast random access but slow insertion and deletion. While LinkedList uses a doubly linked list, which

allows for fast insertion and deletion but slow random access. Also one of the major difference lies in the access time. Node ->



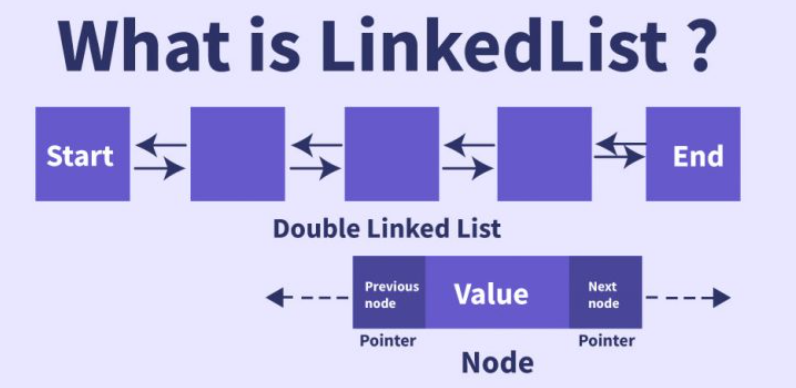
Arraylist is a class which is a part of the Collections Frameworks in java. It implements the List interface hence all the methods of the List class can be used here.

What is LinkedList?

LinkedList are also linear data structures like ArrayList. Here we are not required to specify any size. The objects stored here are not stored in contiguous memory locations like ArrayList. A LinkedList is made up of ‘nodes’. These nodes are the building blocks of the LinkedList just like the cells of an array. A node stores the object data as well as the next node’s address in its p address field.

There are 2 types of Linked List

Singly Linked List – Here every node stores the address of only the next node in the list.



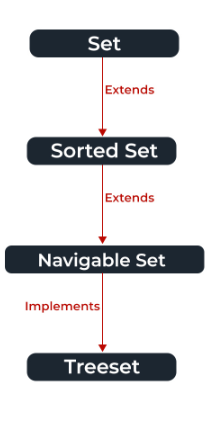
Doubly Linked List – Here every node stores the address of the previous as well as next node in the list. We can traverse the list only in both directions( start to end and end to start ).

**Difference Between ArrayList and LinkedList**

| **Parameters** | **ArrayList** | **LinkedList** |
| --- | --- | --- |
| Underlying Data Structure | Array | Doubly linked list |
| Insertion | Slower compared to LikedListTime Complexity : O(N) | Faster than ArrayListTime Complexity : O(1) |
| Deletion | Slower compared to LikedListTime Complexity : O(N) | Faster than ArrayListTime Complexity : O(1) |
| Traverse | Bidirectional | Bidirectional |
| Searching | Fast searching Time Complexity : O(N) | Fast searching Time Complexity : O(N) |
| Random Access | Fast | Slow |
| Memory Usage | Memory efficient. It only stores the object in the list. | Memory inefficient. It stores the object and the pointers to next and previous nodes. |
| Memory Allocation | Contiguous memory is allocated to all the objects. | Non Contiguous memory is not allocated. |

Understanding the set interface.

The set interface is present in java.util package and extends the Collection interface. It is an unordered collection of objects in which duplicate values cannot be stored.



Since a set doesn’t retain the insertion order, the navigable set interface provides the implementation to navigate through the Set.

Creating Set Objects

Set is an interface, objects cannot be created of the typeset.

Set<Obj> set = new HashSet<Obj> ();

| **Method** | **Description** |
| --- | --- |
| [add(element)](https://www.geeksforgeeks.org/set-add-method-in-java-with-examples/) | 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)](https://www.geeksforgeeks.org/set-addall-method-in-java-with-examples/) | 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()](https://www.geeksforgeeks.org/set-clear-method-in-java-with-examples/) | 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)](https://www.geeksforgeeks.org/set-contains-method-in-java-with-examples/) | This method is used to check whether a specific element is present in the Set or not. |
| [containsAll(collection)](https://www.geeksforgeeks.org/set-containsall-method-in-java-with-examples/) | 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()](https://www.geeksforgeeks.org/set-hashcode-method-in-java-with-examples/) | 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()](https://www.geeksforgeeks.org/set-iterator-method-in-java-with-examples/) | This method is used to return the [iterator](https://www.geeksforgeeks.org/iterators-in-java/) of the set. The elements from the set are returned in a random order. |
| [remove(element)](https://www.geeksforgeeks.org/set-remove-method-in-java-with-examples/) | 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)](https://www.geeksforgeeks.org/set-removeall-method-in-java-with-examples/) | 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)](https://www.geeksforgeeks.org/set-retainall-method-in-java-with-example/) | 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()](https://www.geeksforgeeks.org/set-size-method-in-java-with-example/) | This method is used to get the size of the set. This returns an integer value which signifies the number of elements. |
| [toArray()](https://www.geeksforgeeks.org/set-toarray-method-in-java-with-example/) | This method is used to form an array of the same elements as that of the Set. |

Let us see the output of the following program which try to add duplicate elements in a HashSet.

class Test

{

public static void main(String args[])

{

HashSet hs = new HashSet();

boolean b1 = hs.add("Hi");

boolean b2 = hs.add("world");

boolean b3 = hs.add("Hi");

System.out.println("b1 = "+b1);

System.out.println("b2 = "+b2);

System.out.println("b3 = "+b3);

System.out.println(hs);

}

}

Output:

b1 = true

b2 = true

b3 = false

[ Hiworld , Hi]

package com.wipro.basics.collection;

import java.util.HashSet;

import java.util.Set;

public class SetExample {

public static void main(String[] args) {

Set<Integer> setEx = new HashSet<>();

boolean addCheck = setEx.add(3);

boolean addCheck1 = setEx.add(3);

System.out.println(addCheck + "\*\*\*\*\*\*\*\*\*\*" + addCheck1);

setEx.add(10);

setEx.add(234);

System.out.println(setEx);

Set<Integer> setEx2 = new HashSet<>();

setEx2.add(7);

setEx2.add(4);

setEx.addAll(setEx2);

System.out.println(setEx);

boolean containCheck = setEx.contains(11);

System.out.println(containCheck);

boolean checkALl = setEx.containsAll(setEx2);

System.out.println(checkALl);

System.out.println("in for each");

for(Integer k : setEx) {

System.out.print(k + " ,");

}

setEx.forEach(System.out :: print);

setEx.forEach(y-> System.out.print(y + " ,"));

Set<Employee> sEmp = new HashSet<>();

System.out.println(sEmp.add(new Employee("Tan", 12, 50)));

System.out.println( sEmp.add(new Employee("Tan", 12, 50)));

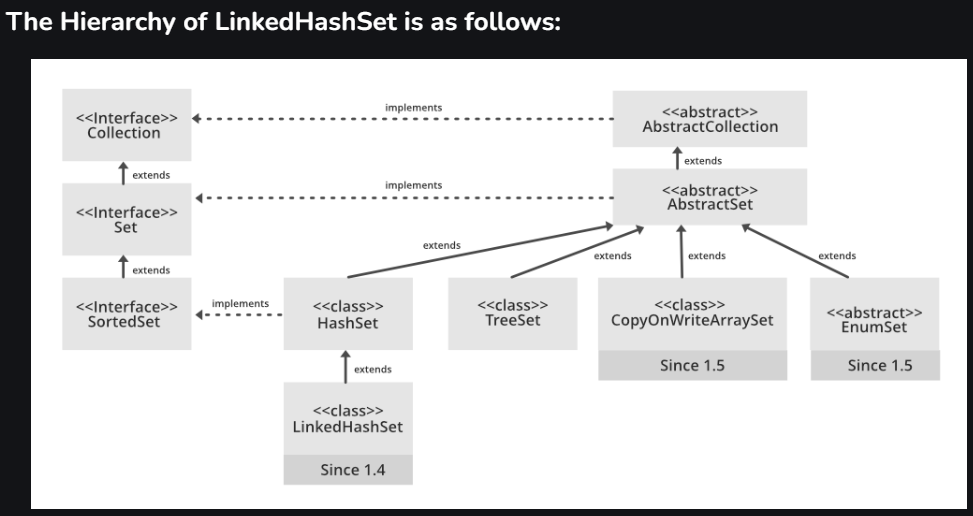
System.out.println(sEmp);

}

}

LINKHASHED SET:

The LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements. When the iteration order is needed to be maintained this class is used.



**All Implemented Interfaces are as listed below:**

Serializable

Cloneable,

Iterable<E>

Collection<E>

Set<E>

Syntax: Declaration

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

Contains unique elements only like HashSet. It extends the HashSet class and implements the Set interface.

Maintains insertion order.

Constructors of LinkedHashSet Class

1. LinkedHashSet(): This constructor is used to create a default HashSet

Set<Interger> set = new HashSet<Interger>();

set.add(8);

set.add(90);

LinkedHashSet<E> hs = new LinkedHashSet<E>(30);

2. LinkedHashSet(Collection C): Used in initializing the HashSet with the elements of the collection C.

LinkedHashSet<E> hs = new LinkedHashSet<E>(Collection c);

3. LinkedHashSet(int size): Used to initialize the size of the LinkedHashSet with the integer mentioned in the parameter.

LinkedHashSet<E> hs = new LinkedHashSet<E>(int size);

What is TreeSet

The TreeSet is a class of the Java Collection framework used to store the tree data structure. It uses a tree data structure to store and maintain the objects. The TreeSet class is the implementing class of the Set interface.

It is a useful class for finding the comparison between the elements, such as greater than, less than, etc., between the available elements of the tree.

Declaration:

The TreeSet class can be declared as follows:

Consider the below example to understand the behaviour of the TreeSet:

import java.util.TreeSet;

public class TreeSetDemo {

// Java program to understand the behavior of the TreeSet

public static void main(String[] args)

{

Set<String> t = new TreeSet<String>(); //implementation of the TreeSet

t.add("Java"); //Adding elements

t.add("Spring");

t.add("Hibernate");

// Duplicates elements are not allowed

t.add("Hibernate");

// By default, the elements will be sorted

// in Ascending order

System.out.println(t);

}

}

Differentiating between various Set implementations.

There are three general-purpose Set implementations — HashSet , TreeSet , and LinkedHashSet . Which of these three to use is generally straightforward. HashSet is much faster than TreeSet (constant-time versus log-time for most operations) but offers no ordering guarantees

How they work internally?

 HashSet uses HashMap internally to store it’s elements.

 LinkedHashSet uses LinkedHashMap internally to store it’s elements.

 TreeSet uses TreeMap internally to store it’s elements.

Performance

 HashSet gives better performance than the LinkedHashSet and TreeSet.

 The performance of LinkedHashSet is between HashSet and TreeSet. It’s performance is

almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList

internally to maintain the insertion order of elements.

 TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the

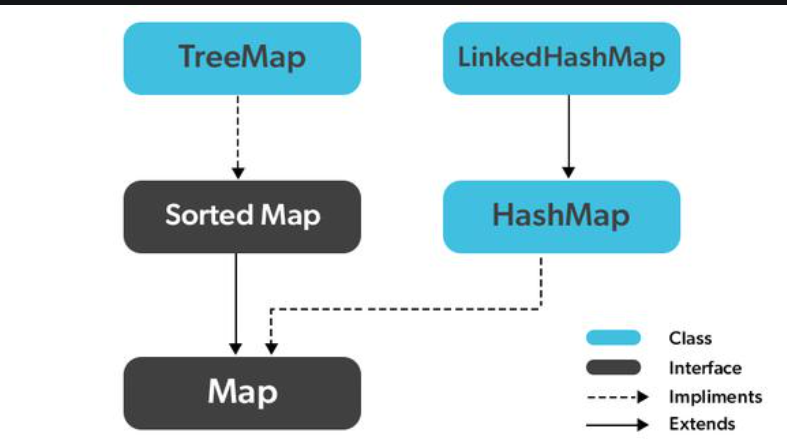
elements after each insertion and removal operations.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Introduction to Map interface:

Map Interface is present in java.util package represents a mapping between a key and a value.

Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys.



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.

There are two interfaces for implementing Map in Java. They are Map and SortedMap, and three classes: HashMap, TreeMap, and LinkedHashMap.

public class JavaExample{

public static void main(String args[]){

HashMap<Integer,String> hMap=new HashMap<>();

hMap.put(101,"Cricket");

hMap.put(105,"Hockey");

hMap.put(111,"Basketball");

System.out.println("HashMap elements: ");

for(Map.Entry mEntry : hMap.entrySet()){

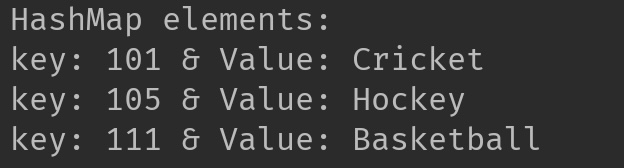
System.out.print("key: "+ mEntry.getKey() + " & Value: ");

System.out.println(mEntry.getValue());

}

}

}

**Output:**  


Checking duplicate key insertion in HashMap:

public class JavaExample{

public static void main(String args[]){

HashMap<Integer,String> hMap=new HashMap<>();

hMap.put(101,"Cricket");

hMap.put(105,"Hockey");

hMap.put(111,"Basketball");

hMap.put(111,"Karate"); //adding element with duplicate key

System.out.println("HashMap elements: ");

for(Map.Entry mEntry : hMap.entrySet()){

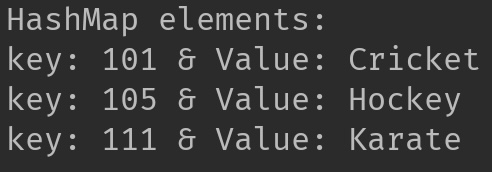
System.out.print("key: "+ mEntry.getKey() + " & Value: ");

System.out.println(mEntry.getValue());

}

}

}

**Output:**  


HashMap, LinkedHashMap, and TreeMap:

Three of the most commonly used java.util.Map interfaces in Java are TreeMap, HashMap, and LinkedHashMap. Each of these have their own unique characteristics and use cases.

TreeMap

TreeMap in Java is all about order. It stores keys in a sorted (ascending) order. The time complexity for insertion and retrieval operations is O(logN). A TreeMap does not permit null keys but can have null values.

Understanding HashMap

Unlike TreeMap, a HashMap does not maintain any order. It offers a constant time performance, O(1), for basic operations like get and put. A HashMap allows one null key and multiple null values.

LinkedHashMap

A LinkedHashMap, similar to HashMap, provides a constant time performance, O(1), for basic operations. However, it also maintains the insertion order. Like HashMap, it permits one null key and multiple null values.

**Comparing TreeMap, HashMap and LinkedHashMap in Java**

Now, let's compare TreeMap, HashMap, and LinkedHashMap based on different parameters.

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameters** | **TreeMap** | **HashMap** | **LinkedHashMap** |
| Performance | TreeMap offers O(logN) time complexity for basic operations. | HashMap provides O(1) time complexity for basic operations. | LinkedHashMap also provides O(1) time complexity for basic operations. |
| Null Keys | TreeMap does not permit null keys. | HashMap allows one null key. | LinkedHashMap permits one null key. |
| Null Values | TreeMap allows null values. | HashMap permits multiple null values. | LinkedHashMap allows multiple null values. |
| Order | TreeMap maintains keys in a sorted order. | HashMap does not maintain any order. | LinkedHashMap maintains the insertion order. |

Comparators and Iterators:

Comparable … Single comparision …. Comparator ---multiple comparision

▸ Interface with a single method that we need to implement:

public int compareTo(T that){

if(this.age> T.getAge(){

return 1;

}else if(this.age< T.getAge(){

Return -1;

}

Else{ return 0;

}

▸ Implement it so that v.compareTo(w):

▸ Returns >0 if v is greater than w.//1

▸ Returns <0 if v is smaller than w.//-1

▸ Returns 0 if v is equal to w.//0

▸ Corresponds to natural ordering.

How to make your class T comparable?

1. Implement Comparable<T> interface.

2. Implement compareTo(T that) method to compare

this T object to that based on natural ordering.

package com.basic.mypackage;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Comparator;

import java.util.List;

import java.util.stream.Collectors;

public class LinkListExmp {

public static void main(String[] args) {

List<Integer> list1 = new ArrayList<>();

list1.add(12);

list1.add(13);

System.***out***.println(list1);

list1.add(0, 11);

System.***out***.println(list1+"after using index");

list1.set(2, 50);

System.***out***.println(list1+"after using set");

System.***out***.println(list1.indexOf(50));

list1.remove(0);

System.***out***.println(list1+"after using remove");

System.***out***.println(list1.get(1));

System.***out***.println(list1.contains(12));

List<Integer> list12 = new ArrayList<>();

list12.add(62);

list12.add(16);

list1.addAll(list12);

System.***out***.println(list1);

list1.forEach(x->System.***out***.println(x));

list1.sort(Comparator.*comparing*(Integer::intValue));

List<Integer> listOutput = list1.stream()

.sorted(Comparator.*comparing*(Integer::intValue))

.collect(Collectors.*toList*());

System.***out***.println(listOutput);

}

}

Iterators:

Iterator();…this method is used to get an Iterator for any collection.

next() … the data ..(Datatype will be of collection datatype)

hasNext()…tell you if the object of data is present at the next index from where the pointer

remove()…..to remove a element or data from the collection that is being called upon.

ArrayList<String> cars = new ArrayList<String>();

cars.add(“volvo”);

cars.add(“BMW”);

Iterator<String> it = cars.iterator();

While(it.hasNext()){

Syso(it.next());

If(it.next()==”volvo”){

it.remove();

}

}

Output : volvo, bmw…