

Unit 10

Holding Collection of Data

Arrays : Declaration

❖ An array is a collection of similar types of data

❖ In Java, here is how we can declare an array:

```
dataType[ ] arrayName;
```

- dataType - it can be primitive data types like int, char, double, byte, etc. or Java objects
- arrayName - it is an identifier

❖ To define the number of elements that an array can hold, we have to allocate memory for the array in Java

```
double[] data = new double[10];
```

❖ Here, the array can store 10 elements. We can also say that the size or length of the array is 10.

Arrays : Declaring and Compile Time Initialization

- ❖ We can declare and allocate memory of an array in one single statement. For example

```
double[] data = new double[10];
```

- ❖ In Java, we can initialize arrays during declaration. For example,

```
int[] age = {23,45,52,34,22};
```

- ❖ Note that we have not provided the size of the array in above example. In this case, the Java compiler automatically specifies the size by counting the number of elements in the array (i.e. 5).

Array Indexes

- ❖ In the Java array, each memory location is associated with a number. The number is known as an array index. We can also initialize arrays in Java, using the index number. For example,

```
// declare an array
int[] age = new int[5];

// initialize array
age[0] = 12;
age[1] = 4;
age[2] = 5;
..
```

age[0]	age[1]	age[2]	age[3]	age[4]
12	4	5	2	5

Java Arrays initialization

- ❖ Array indices always start from 0. That is, the first element of an array is at index 0.
- ❖ If the size of an array is n, then the last element of the array will be at index n-1.

Accessing Array Elements : loop

- ❖ We can access the element of an array using the index number or looping through array elements.

```
public class Main {  
    public static void main(String[] args) {  
        int [] ages = {23,56,12,34};  
        for(int i=0;i<ages.length;++i) {  
            System.out.println(ages[i]);  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        int [] ages = {23,56,12,34};  
        for(int age:ages) {  
            System.out.println(age);  
        }  
    }  
}
```

23
56
12
34

- ❖ We are using the length property of the array to get the size of the array.

Multidimensional Array:2D

- ❖ A multidimensional array is an array of arrays. i.e. each element of a multidimensional array is an array itself.

For Example: `int[][] a = new int[3][4];`

- ❖ Here we have created a multidimensional array named a. It is a 2-dimensional array that can hold a maximum of 12 elements.

	Column 1	Column 2	Column 3	Column 4
Row 1	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
Row 2	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
Row 3	<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

Multidimensional Array: Accessing elements

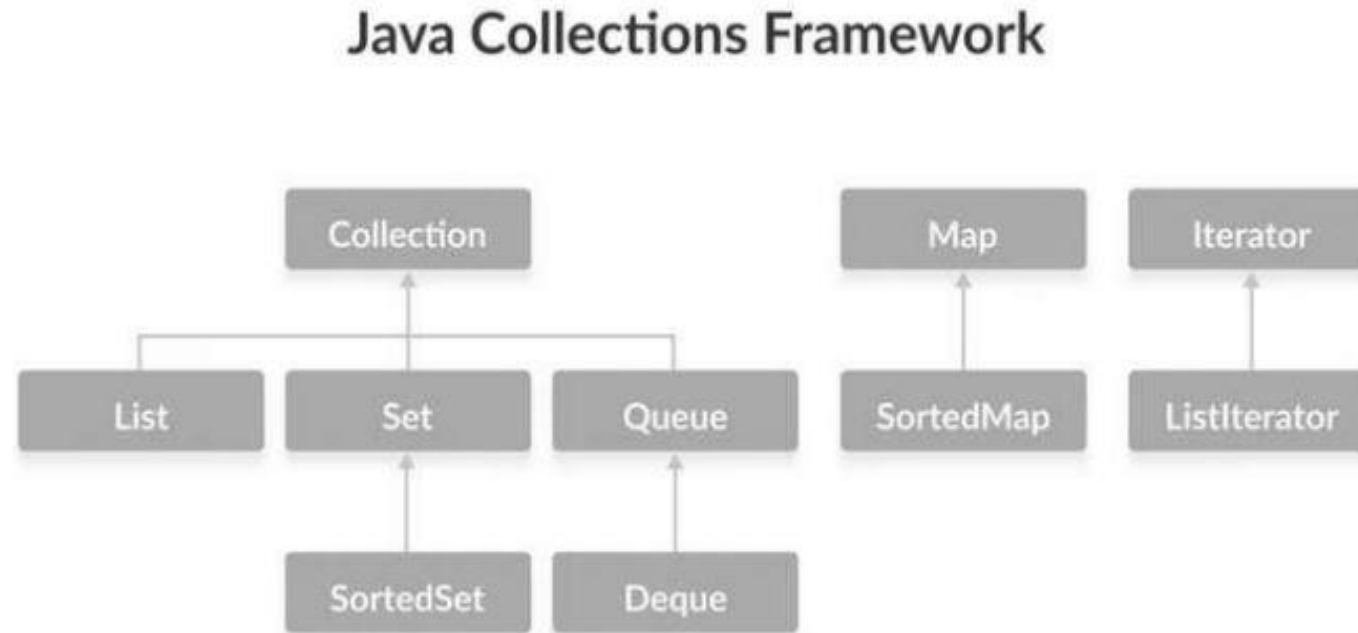
```
public class Main {  
    public static void main(String[] args) {  
        int[][] nums={  
            {11,22,33},  
            {77,88},  
            {-9,-8,-77,-89}  
        };  
        System.out.println("Using For Loop");  
        for(int i=0;i<nums.length;++i) {  
            for(int j=0;j<nums[i].length;++j) {  
                System.out.print(nums[i][j]+" ");  
            }  
            System.out.println();  
        }  
  
        System.out.println("Using For-each Loop");  
        for(int[] numArray:nums) {  
            for(int num:numArray) {  
                System.out.print(num+" ");  
            }  
            System.out.println();  
        }  
    }  
}
```

Output

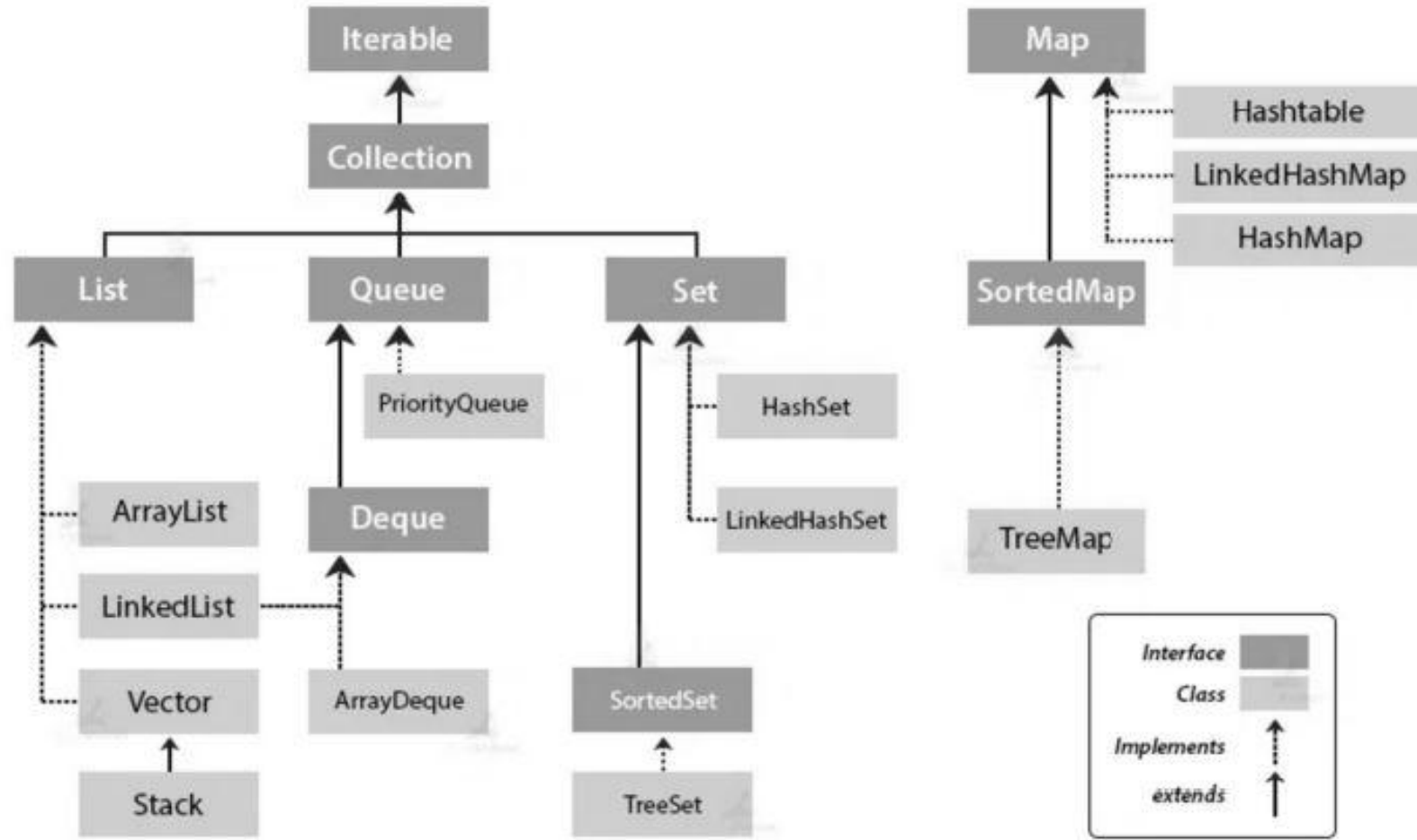
```
Using For Loop  
11 22 33  
77 88  
-9 -8 -77 -89  
Using For-each Loop  
11 22 33  
77 88  
-9 -8 -77 -89
```

Java Collections Framework

- ❖ The Java collections framework provides a set of interfaces and classes to implement various data structures and algorithms.



Collection Framework Hierarchy



Java Collection Interface

- ❖ The Collection interface is the root interface of the collections framework hierarchy.
- ❖ Java does not provide direct implementations of the Collection interface but provides implementations of its sub-interfaces like List, Set, and Queue.

1. List Interface

- The List interface is an ordered collection that allows us to add and remove elements like an array.

2. Set Interface

- The Set interface allows us to store elements in different sets similar to the set in mathematics. **It cannot have duplicate elements.**

3. Queue Interface

- The Queue interface is used when we want to store and access elements in First In, First Out manner.

Java Map and Iterator Interface

- Java Map Interface

- ❖ In Java, the Map interface allows elements to be stored in key/value pairs.
- ❖ Keys are unique names that can be used to access a particular element in a map. And, each key has a single value associated with it.

- Java Iterator Interface

- ❖ In Java, the Iterator interface provides methods that can be used to access elements of collections.

Methods of Collection

- ❖ The Collection interface includes various methods that can be used to perform different operations on objects.
- ❖ These methods are available in all its interfaces.
 - `add()`: inserts the specified element to the collection
 - `size()`: returns the size of the collection
 - `remove()`: removes the specified element from the collection
 - `iterator()`: returns an iterator to access elements of the collection
 - `addAll()`: adds all the elements of a specified collection to the collection
 - `removeAll()`: removes all the elements of the specified collection from the collection
 - `clear()`: removes all the elements of the collection

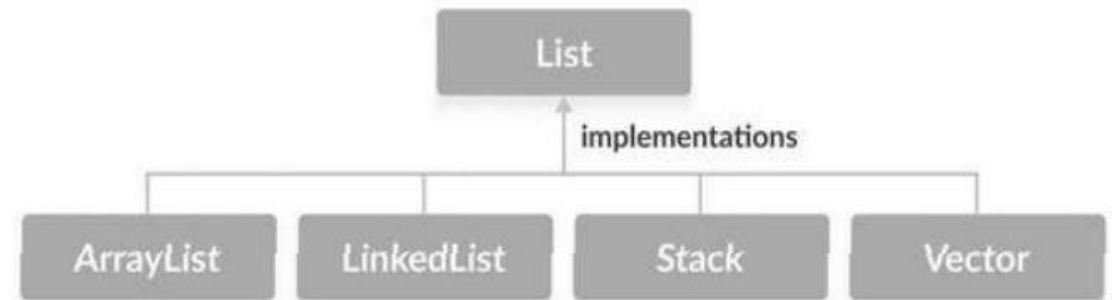
Why the Collections Framework?

- ❖ The Java collections framework provides various data structures and algorithms that can be used directly. This has two main advantages:
 - We do not have to write code to implement these data structures and algorithms manually.
 - Our code will be much more efficient as the collections framework is highly optimized.
- ❖ Moreover, the collections framework allows us to use a specific data structure for a particular type of data
 - **If we want our data to be unique, then we can use the Set interface** provided by the collections framework.
 - To store data in key/value pairs, we can use the Map interface.
 - The Array
 - List class provides the functionality of resizable arrays

Java List Interface

- ❖ In Java, the List interface is an ordered collection that allows us to store and access elements sequentially.
- ❖ It extends the Collection interface.
- ❖ Since List is an interface, we cannot create objects from it.

- ❖ In order to use functionalities of the List interface, we can use these classes: ArrayList, LinkedList, Stack, Vector



Java List Interface

❖ In Java, we must import java.util.List package in order to use List.

```
// ArrayList implementation of List  
List<String> list1 = new ArrayList<>();  
  
// LinkedList implementation of List  
List<String> list2 = new LinkedList<>();
```

- ❖ Here, we have created objects list1 and list2 of classes ArrayList and LinkedList.
- ❖ These objects can use the functionalities of the List interface.

Methods of List

- ❖ `add()` - adds an element to a list
- ❖ `addAll()` - adds all elements of one list to another
- ❖ `get()` - helps to randomly access elements from lists
- ❖ `iterator()` - returns iterator object that can be used to sequentially access elements of lists
- ❖ `set()` - changes elements of lists
- ❖ `remove()` - removes an element from the list
- ❖ `removeAll()` - removes all the elements from the list
- ❖ `clear()` - removes all the elements from the list (more efficient than `removeAll()`)
- ❖ `size()` - returns the length of lists
- ❖ `toArray()` - converts a list into an array
- ❖ `contains()` - returns true if a list contains specified element

Implementation of the List Interface

- ArrayList

- ❖ The ArrayList class of the Java collections framework provides the functionality of resizable arrays
- ❖ Unlike arrays, arraylists can automatically adjust its capacity when we add or remove elements from it. Hence, arraylists are also known as dynamic arrays.

- ❖ Creating an ArrayList

```
// create Integer type arraylist  
ArrayList<Integer> arrayList = new ArrayList<>();  
  
// create String type arraylist  
ArrayList<String> arrayList = new ArrayList<>();
```

- ❖ In the above program, we have used Integer not int. It is because we cannot use primitive types while creating an arraylist. Instead, we have to use the corresponding wrapper classes.

Methods of ArrayList Class

Methods	Descriptions
<code>size()</code>	Returns the length of the arraylist.
<code>sort()</code>	Sort the arraylist elements.
<code>clone()</code>	Creates a new arraylist with the same element, size, and capacity.
<code>contains()</code>	Searches the arraylist for the specified element and returns a boolean result.
<code>ensureCapacity()</code>	Specifies the total element the arraylist can contain.
<code>isEmpty()</code>	Checks if the arraylist is empty.
<code>indexOf()</code>	Searches a specified element in an arraylist and returns the index of the element.

ArrayList Operations

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

```
public class ListExample {
```

```
    public static void main(String[] args) {
```

```
        int i=0;
```

```
        // Creating list using the ArrayList class
```

```
        List myList = new ArrayList();
```

```
        myList.add("Sunday");
```

```
        myList.add("Monday");
```

```
        System.out.println("Elements of myList before adding "+ myList);
```

```
    }
```

```
}
```

tput - ListExample (run) X

run:

Elements of myList before adding [Sunday, Monday]

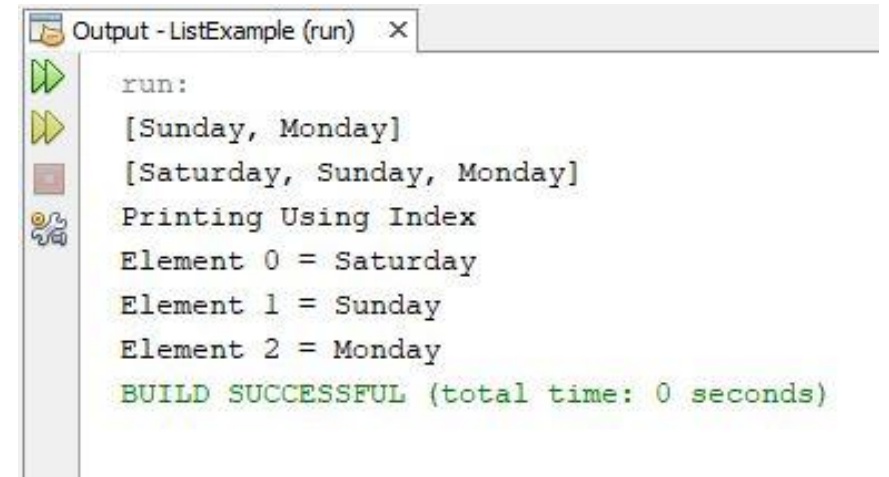
BUILD SUCCESSFUL (total time: 0 seconds)

ArrayList Operations Example

```
import java.util.*;

public class ListExample {

    public static void main(String[] args) {
        int i=0;
        // Creating list using the ArrayList class
        List<String> myList = new ArrayList<String>();
        myList.add("Sunday");
        myList.add("Monday");
        System.out.println(myList);
        myList.add(0, "Saturday");
        System.out.println(myList);
        System.out.println("Printing Using Index");
        for(i=0;i<myList.size();i++){
            System.out.println("Element "+i+" = "+myList.get(i));
        }
    }
}
```



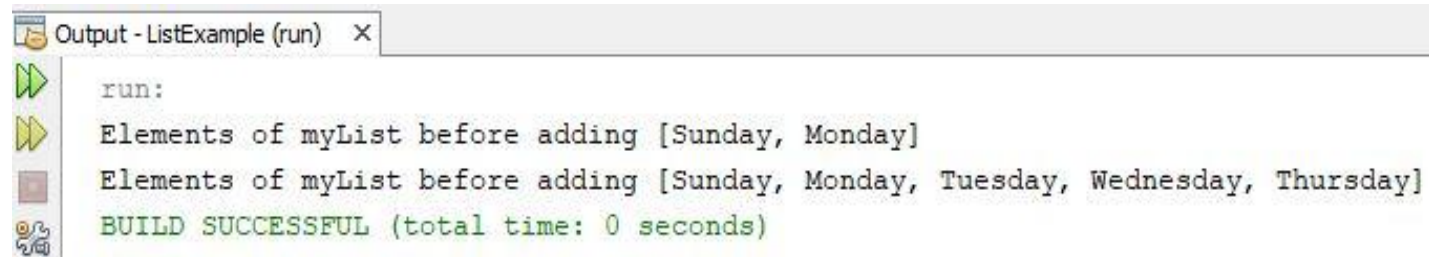
The screenshot shows an IDE's output window titled "Output - ListExample (run)". It contains the following text:

```
run:
[Sunday, Monday]
[Saturday, Sunday, Monday]
Printing Using Index
Element 0 = Saturday
Element 1 = Sunday
Element 2 = Monday
BUILD SUCCESSFUL (total time: 0 seconds)
```

The output window includes standard IDE icons on the left: a green play button, a yellow play button, a red stop button, and a magnifying glass icon.

ArrayList- addAll Operation

```
public class ListExample {  
  
    public static void main(String[] args) {  
        int i=0;  
        // Creating list using the ArrayList class  
        List<String> myList = new ArrayList<String>();  
        List<String> mySecondList = new ArrayList<String>();  
        myList.add("Sunday");  
        myList.add("Monday");  
        System.out.println("Elements of myList before adding "+ myList);  
        mySecondList.add("Tuesday");  
        mySecondList.add("Wednesday");  
        mySecondList.add("Thursday");  
        myList.addAll(mySecondList);  
        System.out.println("Elements of myList before adding "+ myList);  
    }  
}
```



```
run:  
Elements of myList before adding [Sunday, Monday]  
Elements of myList before adding [Sunday, Monday, Tuesday, Wednesday, Thursday]  
BUILD SUCCESSFUL (total time: 0 seconds)
```

ArrayList Example-Array to ArrayList and ArrayList to String

```
//array to ArrayList  
String [] strs = {"Java","Python"};  
ArrayList<String> al_strs = new ArrayList<>(Arrays.asList(strs));  
System.out.println(al_strs);
```

[Java, Python]

```
ArrayList<String>tags= new ArrayList<>();  
tags.add("Nature");  
tags.add("Sunset");  
//ArrayList to String  
String str = tags.toString();  
System.out.println(str);
```

[Nature, Sunset]

Vectors

- ❖ The Vector Class is an implementation of the List Interface that allows us to **create resizable-arrays** similar to the ArrayList class.
- ❖ Creating a vector

`Vector<Type> vector = new Vector<>();`

```
4 import java.util.*;
5
6 public class VectorExample {
7     public static void main(String[] args) {
8         Vector<String> myVector = new Vector<>();
9         myVector.add("January");
10        myVector.add("February");
11        myVector.add("March");
12        System.out.println("Initial Vector "+myVector);
13        String vectorData = myVector.toString();
14        System.out.println("Vector to String "+vectorData);
15        myVector.remove(1);
16        System.out.println("Vector after removing data "+myVector);
17    }
18 }
```

Output - ListExample (run) x

```
run:
Initial Vector [January, February, March]
Vector to String [January, February, March]
Vector after removing data [January, March]
BUILD SUCCESSFUL (total time: 0 seconds)
```


Methods of Vector

`add(element)` - adds an element to vectors

`add(index, element)` - adds an element to the specified position

`addAll(vector)` - adds all elements of a vector to another vector

`remove(index)` - removes an element from specified position

`removeAll()` - removes all the elements

`clear()` - removes all elements. It is more efficient than `removeAll()`

`get(index)` - returns an element specified by the index

`iterator()` - returns an iterator object to sequentially access vector elements

`set()` changes an element of the vector

`size()` returns the size of the vector

`toArray()` converts the vector into an array

`toString()` converts the vector into a String

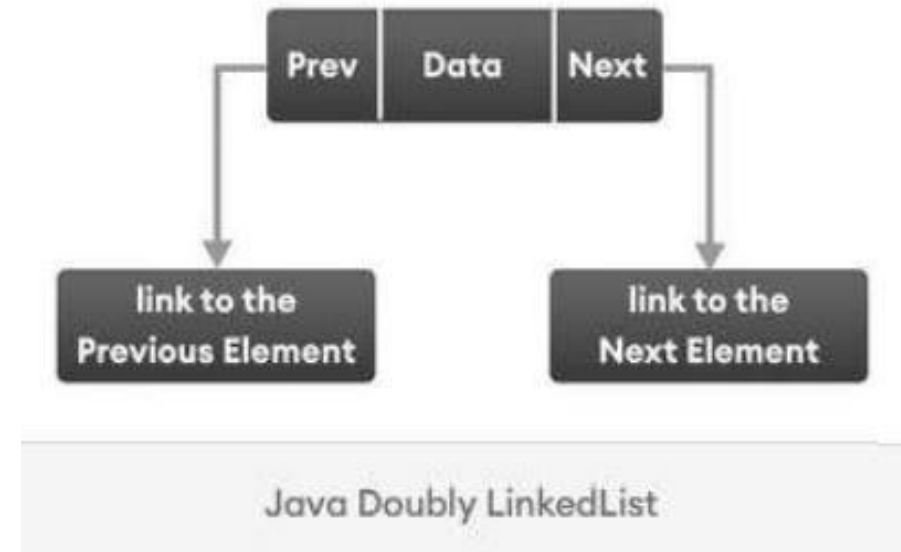
`contains()` searches the vector for specified element and returns a boolean result

Java Vector vs. ArrayList

- ❖ Synchronization : Vector is synchronized, which means only one thread at a time can access the code, while ArrayList is not synchronized, which means multiple threads can work on ArrayList at the same time.
- ❖ Performance: **ArrayList is faster, since it is non-synchronized**, while vector operations give slower performance since they are synchronized (thread-safe).
- ❖ Data Growth: ArrayList and Vector both grow and shrink dynamically to maintain optimal use of storage – but the way they resize is different. ArrayList increments 50% of the current array size if the number of elements exceeds its capacity, while vector increments 100% – essentially doubling the current array size.
- ❖ Traversal: Vector can use both Enumeration and Iterator for traversing over elements of vector while ArrayList can only use Iterator for traversing.

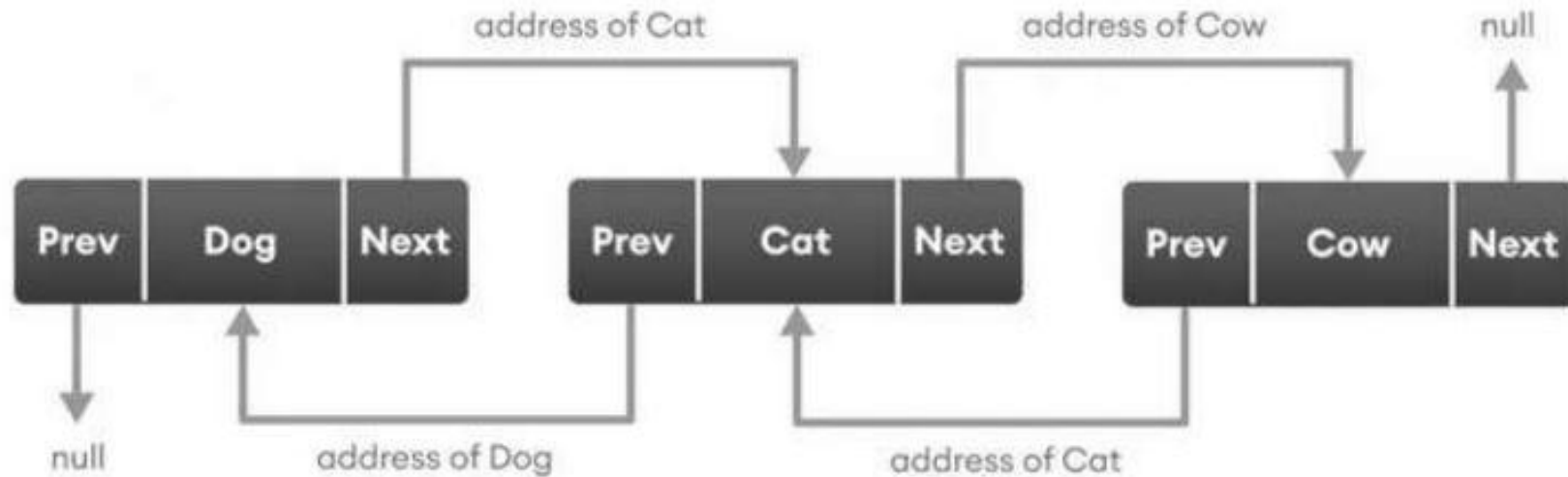
Java LinkedList

- ❖ The LinkedList class of the Java collections framework provides the functionality of the linked list data structure (doubly linkedlist)
- ❖ Each element in a linked list is known as a node.
- ❖ It consists of 3 fields:
 - Prev - stores an address of the previous element in the list. It is null for the first element
 - Next - stores an address of the next element in the list. It is null for the last element
 - Data - stores the actual data



Java LinkedList

- ❖ Elements in linked lists are not stored in sequence.
- ❖ Instead, they are scattered and connected through links (Prev and Next).



Java LinkedList

❖ Creating Linked List in Java

LinkedList<Type> linkedList = new LinkedList<>();

❖ Here Type indicates the type of a linked list

```
// create Integer type linked list
LinkedList<Integer> linkedList = new LinkedList<>();

// create String type linked list
LinkedList<String> linkedList = new LinkedList<>();
```

Java LinkedList Example

```
//create LinkedList
LinkedList<String>frameworks = new LinkedList<>();
//add elements to LinkedList
frameworks.add("Laravel");
frameworks.add("Django");
//view LinkedList
System.out.println(frameworks);
//add elements in a specified position
frameworks.add(0,"Spring");
System.out.println(frameworks);
```

```
[Laravel, Django]
[Spring, Laravel, Django]
```

```
LinkedList<String>frameworks = new LinkedList<>();
frameworks.add("Laravel");
frameworks.add("Django");
//Accessing elements
String framework1=frameworks.get(0);
System.out.println(framework1);
```

```
Laravel
```

Java LinkedList Example

```
LinkedList<String>frameworks = new LinkedList<>();  
frameworks.add("Laravel");  
frameworks.add("Django");  
//Change Element of Linked List  
frameworks.set(0, "Code Igniter");  
System.out.println(frameworks);
```

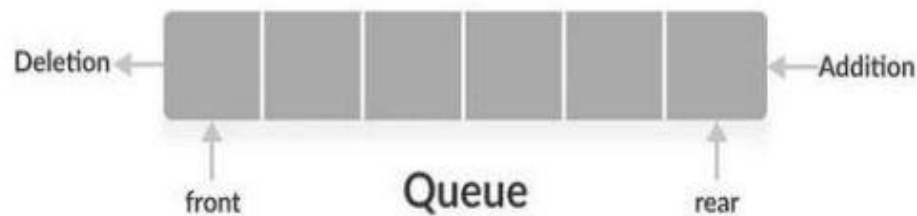
[Code Igniter, Django]

```
LinkedList<String>frameworks = new LinkedList<>();  
frameworks.add("Laravel");  
frameworks.add("Django");  
frameworks.add("Nextjs");  
System.out.println(frameworks);  
//Remove Elements  
String removed = frameworks.remove(2);  
System.out.println("Removed:" + removed);  
System.out.println(frameworks);
```

[Laravel, Django, Nextjs]
Removed:Nextjs
[Laravel, Django]

LinkedList as Deque and Queue

- ❖ Since the LinkedList class also implements the Queue and the Deque interface, it can implement methods of these interfaces as well
- ❖ In queues, elements are stored and accessed in First In, First Out manner. That is, elements are added from the behind and removed from the front.
- ❖ In a deque, we can insert and remove elements from both front and rear



LinkedList as Deque and Queue

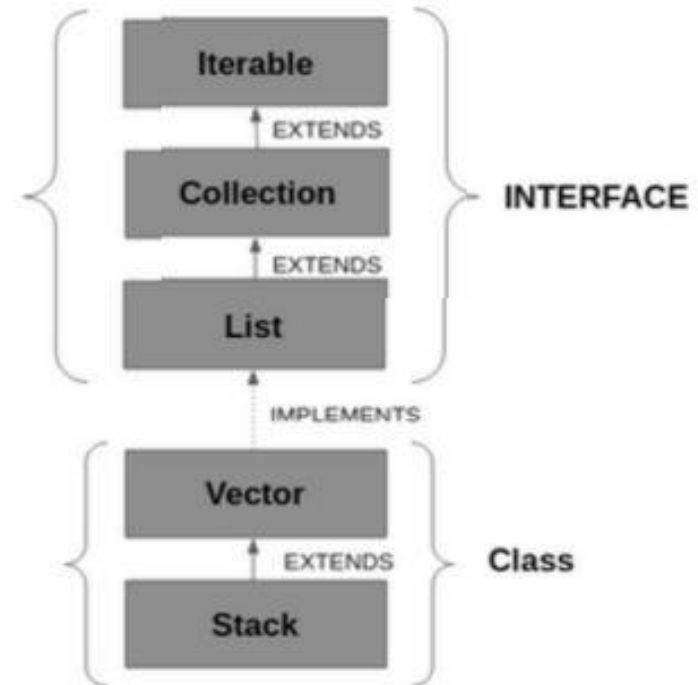
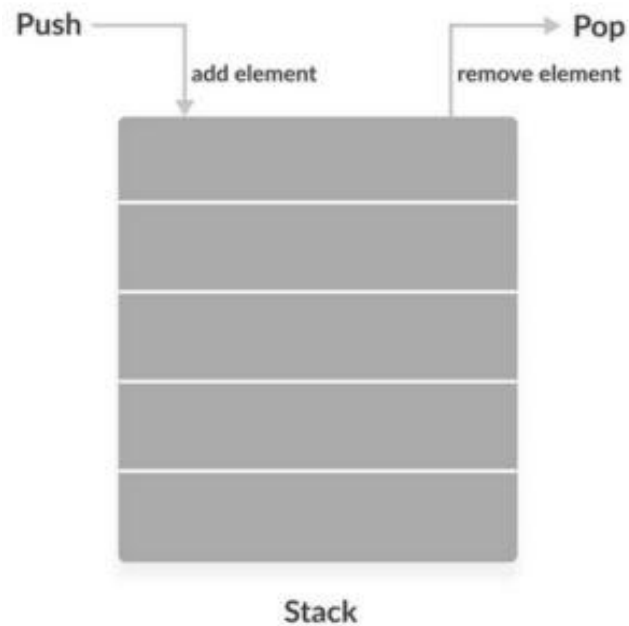
Methods	Descriptions
addFirst()	Adds te specified element at the beginning of the linked list
addLast()	Adds the specified element at the end of the linked list
getFirst()	Returns the first element
getLast()	Returns the last element
removeFirst()	Removes the first element
removeLast()	Removes the last element
poll()	Returns and removes
peek()	Returns the first element (head) of the linked list
Offer()	Adds the specified element at the end of the linked list

LinkedList vs ArrayList

ArrayList	LinkedList
1) ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.
2) Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.
5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contagious.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.	There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) To be precise, an ArrayList is a resizable array.	LinkedList implements the doubly linked list of the list interface.

Stack Class

- ❖ Java Collection framework provides a Stack class that models and implements a Stack data structure.
- ❖ The class is based on the principle of last-in-first-out



Creating a Stack

- ❖ In order to create a stack, we must import the java.util.Stack package first. Once we import the package, here is how we can create a stack in Java.

`Stack<Type> stacks = new Stack<>();`

```
public class StackExample {  
  
    Stack<String> stacks1= new Stack<>();  
    Stack<Integer> stacks2 = new Stack<Integer>();  
}
```

Stack Methods

- ❖ Since Stack extends the Vector class, it inherits all the methods Vector.
- ❖ Besides these methods, the Stack class includes 5 more methods that distinguish it from Vector.
 1. push() :To add an element to the top of the stack
 2. pop() :To remove an element from the top of the stack
 3. peek() :returns an object from the top of the stack
 4. search() :To search an element in the stack. It returns the position of the element from the top of the stack.
 5. empty() :To check whether a stack is empty or not

Stack Class Example

```
import java.util.Stack;

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

        System.out.println("Stack is empty:"+languages.empty());

        languages.push("Java");
        languages.push("Python");
        System.out.println(languages);

        languages.pop();
        System.out.println(languages);

        languages.push("Javascript");
        System.out.println(languages.peek());

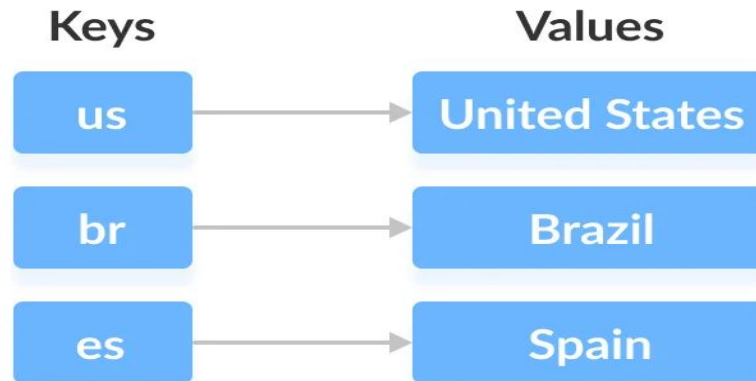
        System.out.println(languages.search("Python"));
        System.out.println(languages.search("Java"));

        System.out.println("Stack is empty:"+languages.empty());
    }
}
```

```
Stack is empty:true
[Java, Python]
[Java]
Javascript
-1
2
Stack is empty:false
```

Java Map Interface

- ❖ The Map interface of the Java collections framework provides the functionality of the map data structure.
- ❖ In Java, element of Map are stored in **key/value** pairs. **Keys** are unique values associated with individual **Values**.
- ❖ A map cannot contain duplicate keys. And, each key is associated with a single value.



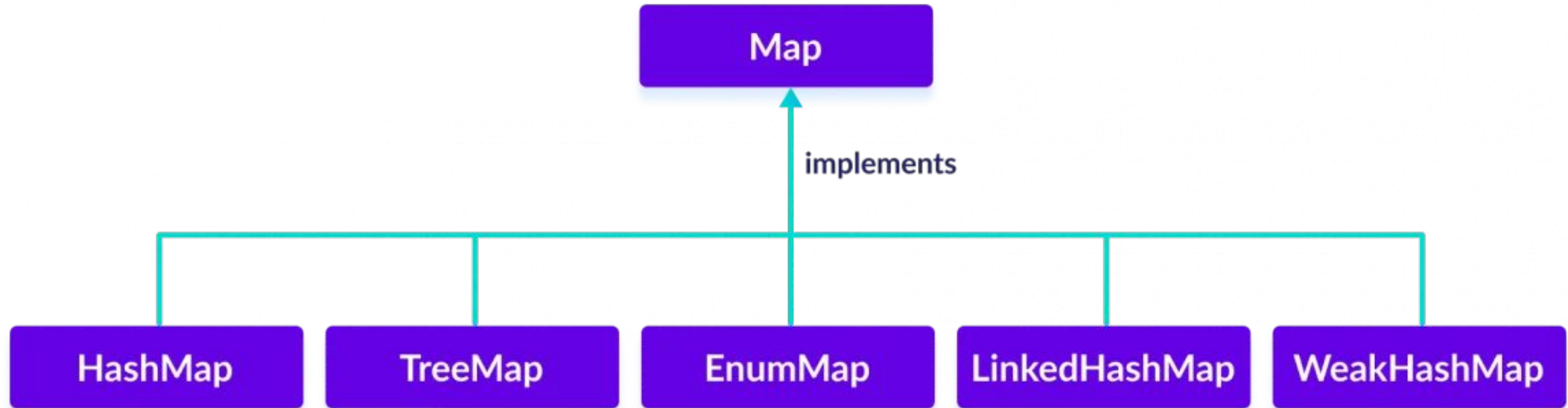
Java Map Interface

- ❖ We can access and modify values using the keys associated with them
- ❖ In the above diagram, we have values: United States, Brazil, and Spain. And we have corresponding keys: us, br, and es.
- ❖ Now, we can access those values using their corresponding keys
- ❖ Note: The Map interface maintains 3 different sets:
 - the set of keys
 - the set of values
 - the set of key/value associations (mapping)
- ❖ Hence we can access keys, values, and associations individually.

```
// Map implementation using HashMap  
Map<Key, Value> numbers = new HashMap<>();
```

Classes that implement Map

Collections Framework



Methods of Map

- ❖ The Map interface includes all the methods of the Collection Interface.
- ❖ Besides methods available in the Collection interface, the Map interface also includes the following methods.

Method	Description
put(K, V)	Inserts the association of a key K and a value V into the map. If the key is already present, the new value replaces the old value.
putAll()	Inserts all the entries from the specified map to this map.
putIfAbsent(K, V)	Inserts the association if the key K is not already associated with the value V.
get(K)	Returns the value associated with the specified key K. If the key is not found, it returns null.
getOrDefault(K, defaultValue)	Returns the value associated with the specified key K. If the key is not found, it returns the defaultValue.

Methods of Map

Method	Description
containsKey(K)	Checks if the specified key K is present in the map or not.
containsValue(V)	Checks if the specified value V is present in the map or not.
replace(K, V)	Replace the value of the key K with the new specified value V.
replace(K, oldValue, newValue)	Replaces the value of the key K with the new value newValue only if the key K is associated with the value oldValue.
remove(K)	Removes the entry from the map represented by the key K.
remove(K, V)	Removes the entry from the map that has key K associated with value V.
values()	Returns a set of all the values present in a map.
keySet()	Returns a set of all the keys present in a map.
entrySet()	Returns a set of all the key/value mapping present in a map.

HashMap

- ❖ The HashMap class of the Java collections framework provides the functionality of the hash table data structure.
- ❖ It stores elements in key/value pairs. Here, keys are unique identifiers used to associate each value on a map.

- ❖ Creating a HashMap

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

- ❖ Example

```
HashMap<String, Integer> numbers = new HashMap<>();
```

Here Key is of String type and value is of Integer type

HashMap Operations

```
//create HashMap
HashMap<String, String> contacts = new HashMap<>();
//add elements
contacts.put("John", "9899999999");
contacts.put("Jane", "9888888888");
//access HashMap elements
System.out.println(contacts.get("John"));
```

9899999999

```
HashMap<String, String> contacts = new HashMap<>();
contacts.put("John", "9899999999");
contacts.put("Jane", "9888888888");
//get KeySet
System.out.println(contacts.keySet());
//get ValueSet
System.out.println(contacts.values());
//get EntrySet
System.out.println(contacts.entrySet());
```

```
[John, Jane]
[9899999999, 9888888888]
[John=9899999999, Jane=9888888888]
```

HashMap Operations

```
HashMap<String, String> contacts = new HashMap<>();
contacts.put("John", "9899999999");
contacts.put("Jane", "9888888888");
contacts.put("Mary", "9866666666");
System.out.println(contacts);
//replacing values
contacts.replace("John", "777777777");
System.out.println(contacts);
//remove HashMap element
contacts.remove("Mary");
System.out.println(contacts);
```

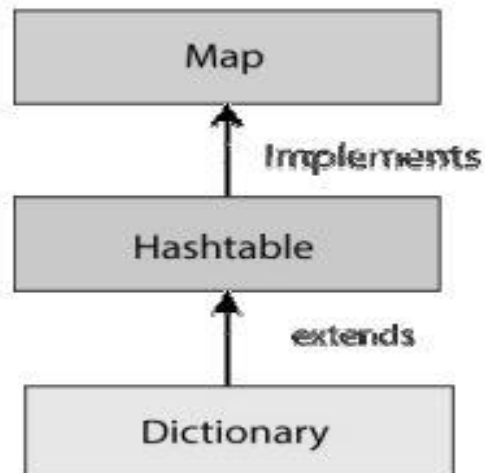
```
{John=9899999999, Jane=9888888888, Mary=9866666666}
{John=777777777, Jane=9888888888, Mary=9866666666}
{John=777777777, Jane=9888888888}
```

Assignment

Differentiate between HashTable and HashMap

Dictionary

- ❖ Dictionary is an abstract class, representing a key-value relation and works similar to a map
- ❖ Given a key you can store values and when needed can retrieve the value back using its key



elements()	returns enumeration of the values: Enumeration<V>
get(Object key)	returns a value for the given key
isEmpty()	returns true if there are no mappings
keys()	returns enumeration of the keys: Enumeration<K>
put(K key, V value)	inserts the key/value mapping
remove(Object key)	removes the mapping for the given key
size()	returns the number of mappings

Java Set Interface

- ❖ The Set interface of the Java Collections framework provides the features of the mathematical set in Java.
- ❖ It extends the Collection interface.
- ❖ Unlike the List interface, sets cannot contain duplicate elements.



Methods of Set

add() - adds the specified element to the set

addAll() - adds all the elements of the specified collection to the set

iterator() - returns an iterator that can be used to access elements of the set sequentially

remove() - removes the specified element from the set

removeAll() - removes all the elements from the set that is present in another specified set

retainAll() - retains all the elements in the set that are also present in another specified set

clear() - removes all the elements from the set

size() - returns the length (number of elements) of the set

toArray() - returns an array containing all the elements of the set

contains() - returns true if the set contains the specified element

containsAll() - returns true if the set contains all the elements of the specified collection

hashCode() - returns a hash code value (address of the element in the set)

Set Operations

- ❖ The Java Set interface allows us to perform basic mathematical set operations like union, intersection, and subset.

Union - to get the union of two sets `x` and `y`, we can use `x.addAll(y)`

Intersection - to get the intersection of two sets `x` and `y`, we can use `x.retainAll(y)`

Subset - to check if `x` is a subset of `y`, we can use `y.containsAll(x)`

Java HashSet Class

- ❖ The HashSet class of the Java Collections framework provides the functionalities of the hash table data structure.
- ❖ HashSet stores the elements by using a mechanism called hashing.
- ❖ As it implements the Set Interface, duplicate values are not allowed.
- ❖ Objects that you insert in HashSet are not guaranteed to be inserted in the same order. Objects are inserted based on their hash code.
- ❖ **NULL elements are allowed in HashSet.**

HashSet

❖ Creating a hashset

```
HashSet<Integer> numbers = new HashSet<>()
```

❖ HashSet can be defined in terms of capacity and loadFactor

❖ Example:

```
HashSet<Integer> numbers = new HashSet<>(8, 0.75);
```

- a. capacity: Capacity defines how many elements particular hashset can store. The capacity of this hash set is 8. Meaning, it can store 8 elements.
- b. loadFactor: The load factor of this hash set is 0.75. This means, whenever our hash set is filled by 75%, the elements are moved to a new hash table of double the size of the original hash table.

HashSet Operations


```
HashSet<Integer> primes = new HashSet<>();  
//add elements  
primes.add(2);  
primes.add(3);  
primes.add(5);  
System.out.println(primes);  
//remove elements  
primes.remove(2);  
System.out.println(primes);
```

```
[2, 3, 5]  
[3, 5]
```

```
HashSet<Integer> evens = new HashSet<>();  
evens.add(2);  
evens.add(4);  
HashSet<Integer> odds = new HashSet<>();  
odds.add(1);  
odds.add(3);  
HashSet<Integer> numbers = new HashSet<>();  
numbers.add(0);  
numbers.addAll(evens);  
numbers.addAll(odds);  
System.out.println(numbers);
```

```
[0, 1, 2, 3, 4]
```

Union of sets




HashSet Operations

```
HashSet<Integer> primes = new HashSet<>();  
primes.add(3);  
primes.add(5);  
HashSet<Integer> odds = new HashSet<>();  
odds.add(1);  
odds.add(3);  
  
odds.retainAll(primes);  
System.out.println(odds);
```

[3]

Intersection of Set



```
HashSet<Integer> primes = new HashSet<>();  
primes.add(3);  
primes.add(5);  
HashSet<Integer> odds = new HashSet<>();  
odds.add(1);  
odds.add(3);  
  
odds.removeAll(primes);  
System.out.println(odds);
```

[1]

Difference of Set



TreeSet

- ❖ The TreeSet class of the Java collections framework provides the functionality of a tree data structure.
- ❖ The elements in TreeSet are sorted naturally (ascending order)

Methods	Description
add()	Inserts the specified element to the set
addAll()	Inserts all the elements of the specified collection to the set
remove()	Removes the specified element from the set
removeAll()	Removes all the elements from the set
first()	Returns the first element of the set
last()	Returns the last element from the set
ceiling() floor()	Returns the ceiling and floor values from the set

TreeSet

Methods	Description
headset()	The headset() method returns a view of the portion of this set whose elements are strictly less than to the element
tailSet()	The tailSet() method returns a view of the portion of this set whose elements are greater than or equal to from element.

TreeSet

```
import java.util.TreeSet;

public class Javaapp {

    public static void main(String[] args) {

        TreeSet<Integer> ts = new TreeSet<Integer>();
        ts.add(20);
        ts.add(40);
        ts.add(60);
        ts.add(80);
        System.out.println("ceiling(15) : "+ts.ceiling(15));
        System.out.println("ceiling(25) : "+ts.ceiling(25));
        System.out.println("floor(65) : "+ts.floor(65));
        System.out.println("floor(85) : "+ts.floor(85));
        System.out.println("headSet(65) : "+ts.headSet(65));
        System.out.println("tailSet(25) : "+ts.tailSet(25));
    }
}
```

Program Output

```
ceiling(15) : 20
ceiling(25) : 40
floor(65) : 60
floor(85) : 80
headSet(65) : [20, 40, 60]
tailSet(25) : [40, 60, 80]
```

Java Iterator Interface

- ❖ The Iterator interface of the Java collections framework allows us to access elements of a collection.
- ❖ All the Java collections include an iterator() method. This method returns an instance of iterator used to iterate over elements of collections.
- ❖ The Iterator interface provides 4 methods that can be used to perform various operations on elements of collections
 - hasNext() - returns true if there exists an element in the collection
 - next() - returns the next element of the collection
 - remove() - removes the last element returned by the next()
 - forEachRemaining() - performs the specified action for each remaining element of the collection

Iterator Option

```
// Creating an ArrayList
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(3);
numbers.add(2);
System.out.println("ArrayList: " + numbers);
// Creating an instance of Iterator
Iterator<Integer> iterate = numbers.iterator();
// Using the next() method
int number = iterate.next();
System.out.println("Accessed Element: " + number);
// Using the remove() method
iterate.remove();
System.out.println("Removed Element: " + number);
System.out.print("Updated ArrayList: ");
// Using the hasNext() method
while(iterate.hasNext()) {
    int number1 = iterate.next();
    System.out.print(number1+" ");
}
```

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

Java ListIterator Interface

- ❖ The ListIterator interface of the Java collections framework provides the functionality to access elements of a list
- ❖ It is bidirectional. This means it allows us to iterate elements of a list in both the direction. It extends the Iterator interface:
 - hasNext() - returns true if there exists an element in the list
 - next() - returns the next element of the list
 - nextIndex() - returns the index of the element that the next() method will return
 - previous() - returns the previous element of the list
 - previousIndex() - returns the index of the element that the previous() method will return
 - remove() - removes the element returned by either next() or previous()
 - set() - replaces the element returned by either next() or previous() with the specified element

List Iterator Operations

```
// Creating an ArrayList
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(3);
numbers.add(2);
System.out.println("ArrayList: " + numbers);
// Creating an instance of ListIterator
ListIterator<Integer> iterate = numbers.listIterator();
// Using the next() method
int number1 = iterate.next();
System.out.println("Next Element: " + number1);
// Using the nextIndex()
int index1 = iterate.nextIndex();
System.out.println("Position of Next Element: " + index1);

// Using the hasNext() method
System.out.println("Is there any next element? " + iterate.hasNext());
}
```

```
ArrayList: [1, 3, 2]
Next Element: 1
Position of Next Element: 1
Is there any next element? true
```

List Iterator Operations

```
// Creating an ArrayList
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(3);
numbers.add(2);
System.out.println("ArrayList: " + numbers);

// Creating an instance of ListIterator
ListIterator<Integer> iterate = numbers.listIterator();
iterate.next();
iterate.next();

// Using the previous() method
int number1 = iterate.previous();
System.out.println("Previous Element: " + number1);

// Using the previousIndex()
int index1 = iterate.previousIndex();
System.out.println("Position of the Previous element: " + index1);
```

```
ArrayList: [1, 3, 2]
Previous Element: 3
Position of the Previous element: 0
```

Comparator

- ❖ A comparator interface is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of two different classes.
- ❖ This interface is present in `java.util` package and contains 2 methods `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 `Collections.Sort()` work?
 - Internally the Sort method does call 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

Comparator-Example

```
// A class to represent a student.
class Student {
    int rollno;
    String name, address;

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

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name + " "
            + this.address;
    }
}
```

```
class Sortbyroll implements Comparator<Student> {
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}
```

```
class Sortbyname implements Comparator<Student> {
    // Used for sorting in ascending order of
    // name
    public int compare(Student a, Student b)
    {
        return a.name.compareTo(b.name);
    }
}
```

```
class Main {
    public static void main(String[] args)
    {
        ArrayList<Student> ar = new ArrayList<Student>();
        ar.add(new Student(111, "bbbb", "london"));
        ar.add(new Student(131, "aaaa", "nyc"));
        ar.add(new Student(121, "cccc", "jaipur"));

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

        Collections.sort(ar, new Sortbyroll());

        System.out.println("\nSorted by rollno");
        for (int i = 0; i < ar.size(); i++)
            System.out.println(ar.get(i));

        Collections.sort(ar, new Sortbyname());

        System.out.println("\nSorted by name");
        for (int i = 0; i < ar.size(); i++)
            System.out.println(ar.get(i));
    }
}
```

Unsorted

111 bbbb london
131 aaaa nyc
121 cccc jaipur

Sorted by rollno

111 bbbb london
121 cccc jaipur
131 aaaa nyc

Sorted by name

131 aaaa nyc
111 bbbb london
121 cccc jaipur

Assignments

- ❖ Write about Hashset and TreeSet in Java with example operations.
- ❖ What are iterators in Java? Explain with examples
- ❖ What is a comparator in Java? Explain with example