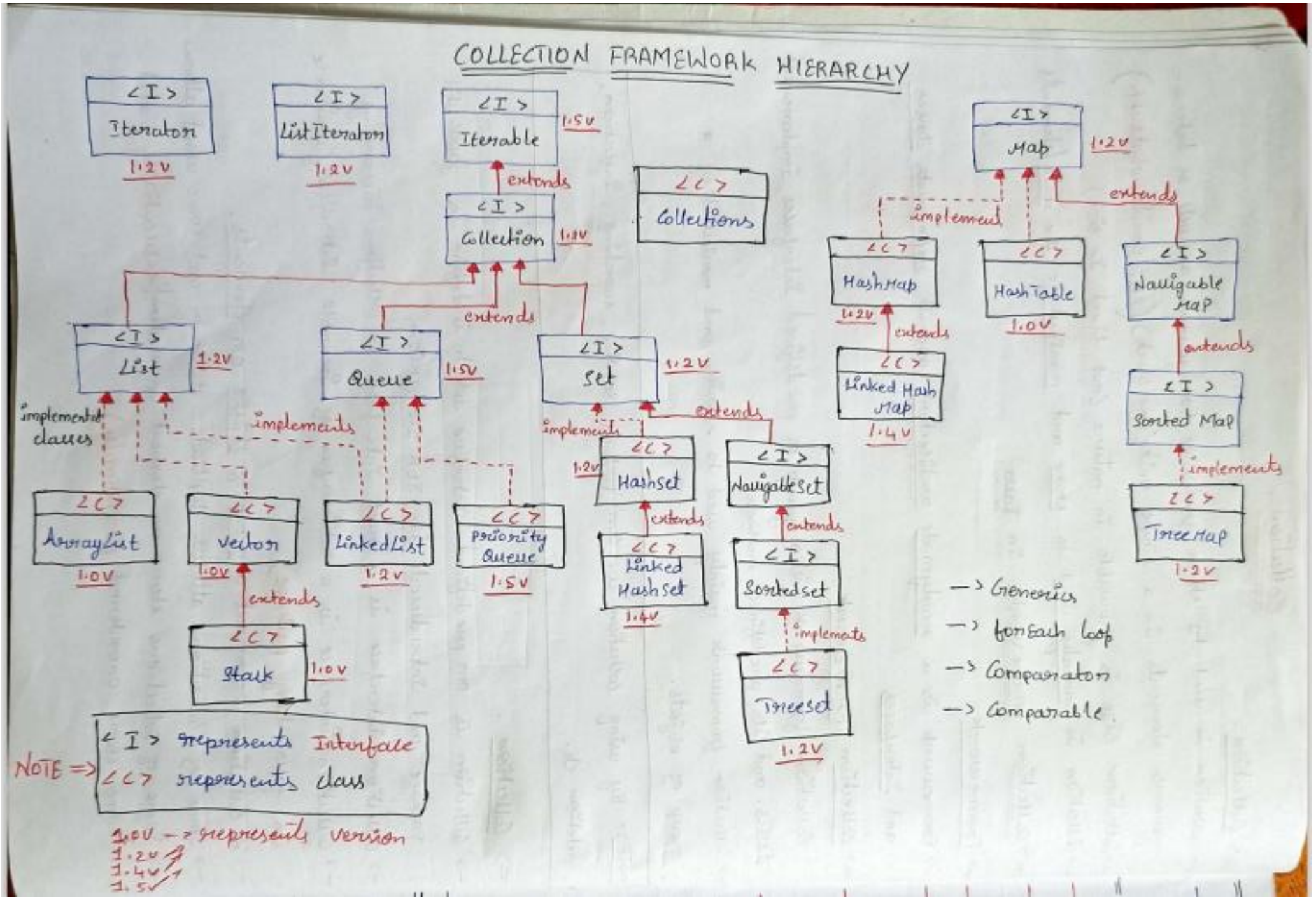
Collections

- Collection is used to store group of homogeneous as well as heterogeneous elements in a single unit (one block)
- Collection size is growable in nature. (not fixed in size)
- Collection is mainly used to store and manipulate the data/ elements.
- COLLECTION FRAMEWORK:
- Framework is a readymade architecture and it represents classes and interfaces.
- Collection framework is a group of pre-defined interfaces, implemented classes and its pre-defined methods.
- Collection framework mainly used to store and manipulate a group of objects.
- By using collection, we can perform sorting, searching, insertion, deletion etc.
- Collection is a pre-defined which is define in java.util.
- Collection interface is a root interface in collection hierarchy
- A collection represents a group of elements or objects.
- Some of the collection will allow duplicate elements and some won't.
- Some will store sequentially and some will store randomly.
- https://youtu.be/K1iu1kXkVoA?si=1Ihuqf_S4V6IBt21
-

COLLECTION FRAMEWORK HIERARCHY



Collection Interface

- Collection interface which contains 3 sub interfaces, they are:
  - List
  - Queue
  - Set
- Pre-defined Methods of collection interface
  - **add (Object e): return type is Boolean; used to add an element to the collection interface.**
  - **addAll (): return type is Boolean; used to add all added collection element to the collection again.**
  - **clear (): return type is void, to remove all the elements from the collection.**
  - **contains (Object o): return type is Boolean; it is used to fetch the given object element. if the element is present, it returns true else return false.**
  - **equals (Object o): return type is Boolean; it is used to compare Object elements if it is same return true else false.**
  - **isEmpty (): Boolean; to check the given collection is empty or not?**
  - **iterator (): Iterator method is used to traverse element by element (Only in forward direction)**
  - **remove (Object o): Boolean; to remove the given element, If the element is removed it returns true else false.**
  - **removeAll (Collection c): Boolean; to remove all the elements from the collection (it works similarly top the clear ()).**
  - **size (): return type is int; to fetch the length/ size of the collection.**

■ LIST

- It is a sub-interface of collection interface and it is defined in java.util.
- It allows duplicates and allows NULL.
- It stores the elements based on index (Starts from zero)
- List will store an element in two ways: Sequential list and non-sequential list
- SEQUENTIAL LIST
    - Elements will be stored in sequentially or orderly.
    - Examples are Array and ArrayList.
- NON-SEQUENTIAL LIST
    - Element will be stored randomly
    - Example is Linked list
- **List interface which contains three implemented classes**, they are
    - ArrayList
    - Vector
    - LinkedList
- **ArrayList Class:**
- ArrayList is an implemented class which implements from list interface and is defined in java.util.
- ArrayList allows duplicates and NULL value.
- ArrayList is not Synchronized.
- It stores elements sequentially based on index.
- ArrayList initial capacity is 10 and grow based on initial capacital formula
- Initial capacital formula: ((current capacity*3)/2) +1
- Once the capacity of the ArrayList is completed based on the ICF new memory get created and old memory will be copied to new memory.
- Old memory reference variable will be referenced to the new memory and old memory removed by GC (Garbage Collection).
- **It stores both homogenous and heterogenous types and has no size limit unlike arrays.**

```
import java.util.ArrayList;
import java.util.List;
public class Demo27{
public static void main(String []args){
ArrayList l = new ArrayList();
l.add(10);
l.add("Java");
l.add(10);
l.add('a');
l.add(45.67);
l.add(true);
System.out.println(l);
System.out.println(l.size());
System.out.println(l.contains(20));
System.out.println(l.get(0));
System.out.println(l);
System.out.println(l.indexOf("Java"));
l.add(10);
System.out.println(l);
System.out.println(l.remove(l.indexOf(10)));
System.out.println(l);
}
}
```

-

```
C:\Users\ganieswar\Desktop\java_programs>javac Demo27.java
Note: Demo27.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

-

- The warning you are encountering is due to **unchecked or unsafe operations** when using a raw type (ArrayList) without specifying the type of elements it stores. In Java, it's good practice to use **generics** to specify the type of objects the list will contain. This helps avoid runtime type issues and eliminates the warning.
- **Solution:**
  You can modify the code to use generics. If you want a list that can store elements of different types, use ArrayList<Object>:

```
import java.util.ArrayList;
import java.util.List;
public class Demo27{
public static void main(String []args){
ArrayList<Object> l = new ArrayList<>();
l.add(10);
l.add("Java");
l.add(10);
l.add('a');
l.add(45.67);
l.add(true);
System.out.println(l);//[10, Java, 10, a, 45.67, true]
System.out.println(l.size());//6
System.out.println(l.contains(20));//false
System.out.println(l.get(0));//10
l.remove(l.get(0));
System.out.println(l);//[Java, 10, a, 45.67, true]
System.out.println(l.indexOf("Java"));//0
l.add(10);
System.out.println(l);//[Java, 10, a, 45.67, true, 10]
System.out.println(l.remove(l.indexOf(10)));//10
System.out.println(l);//[Java, a, 45.67, true, 10]
}
}
```

- **Pre-defined method of List:**
  - **get (int index) – return type is Object – available in List - it is used to fetch the element while providing index.**
  - **indexOf (Object o) – int – List – it will provide index while accepting elements. (returns -1 if element is not present)**
  - **lastIndexOf (Object o) – int - List - when collection contain duplicates and to fetch last duplicate element index.**
  - **remove (int index) – Object – List – It is used to remove an element based on the given index.**
- We can only use index number in remove () method, if you know the index we can use directly, if we know the element and don't know the index use indexOf(element) and the result of this as the input for remove ():
- al.remove(al.indexOf (element));
- remove () not only removes the element from the collection but also returns the element that is removed.

```
import java.util.ArrayList;
import java.util.List;
public class Demo28{
public static void main(String []args){
int arr[] = {10,20,30,40,50};
ArrayList<Object> al = new ArrayList<>();
for(int i=0;i<arr.length;i++)
{
al.add(arr[i]);
}
System.out.println(al);//[10, 20, 30, 40, 50]
int element=40;
al.remove(al.indexOf(element));
System.out.println(al);//[10, 20, 30, 50]
}
}
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
public class Demo28{
public static void main(String []args){
ArrayList<Object> al = new ArrayList<>();
Scanner sc= new Scanner(System.in);
System.out.println("enter the size: ");
int size = sc.nextInt();
System.out.println("enter the elements: ");
for(int i=0;i<size;i++)
{
al.add(sc.nextInt());
}
System.out.println(al);
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo28.java
enter the size:
5
enter the elements:
10 11 12 13 14
[10, 11, 12, 13, 14]
```

■ Vector

- Vector is an implemented class which implements from list interface.
- It is defined in java.util package and it allows both duplicates and NULL values.
- Vector class is synchronized and thread safe.
- Vector class initial capacity is 10 and it will grow double of this current capacity.
- Initial Capacity = Current Capacity * 2
- Vector class is a legacy old class, we use it only when we want to achieve.
- ArrayList is replica of Vector (same)

```
import java.util.Vector;
import java.util.List;
public class Demo27{
public static void main(String []args){
Vector<Object> l =new Vector<>();
l.add(10);
l.add("vector");
l.add(20);
l.add('a');
l.add(null);
System.out.println(l);//[10, vector, 20, a, null]
System.out.println(l.size());//5
System.out.println(l.contains(20));//true
System.out.println(l.get(0));//10
l.remove(l.get(0));
System.out.println(l);//[vector, 20, a, null]
System.out.println(l.indexOf("vector"));//0
l.add(10);
System.out.println(l);//[vector, 20, a, null, 10]
System.out.println(l.remove(l.indexOf('a')));//a
System.out.println(l);//[vector, 20, null, 10]
System.out.println(l.capacity());//10
System.out.println(l.firstElement());//vector
System.out.println(l.lastElement());//10
System.out.println(l.set(2,"java"));//null
System.out.println(l.set(3,11));//10
System.out.println(l);//[vector, 20, java, 11]
}
}
```

-
- We can also add null value into the collection.
- capacity () method is used to return the capacity of the collection.
- **set (index, override_value): It overrides with the override_value in the index given and also returns the present value in the index.**

■ STACK

- Stack is a subclass of vector class which is defined in java.util package.
- It allows duplicates and null values.
- Stack follows Last in first out (LIFO) or First in last out.
- If there is no element in stack that is known as empty stack and stack initial capacity is 10 and grows double of its current capacity.
- Vector class and stack class mainly we use for reservation (like who reserved seat first) by using setsize () method.
- Stack class is synchronized and threadsafe.
- Stack class is a legacy class.
- **Pre-defined methods of stack and vector class: (return type, present in)**
- **push (Object e): (object, stack)**
  It is used to add an element to the stack (at last)
- **Pop (): (object, stack)**
  It is used to remove the topmost or last element from stack
  When there is no element in the stack if we try to perform pop operation JVM will throw EmptyStackException.
- **peek (): (object, stack)**
  It is used to fetch the topmost or last element from the stack
  When there is no element in the stack if we try to perform peek operation JVM will throw EmptyStackException.
- **capacity (): (int, vector)**
  It is used to fetch the initial/ current capacity.
- **setsize (int size): (void, vector)**
  It is used to preserve the stack memory based on given index.
- **set (int index, Object e): (Object, vector)**
  It is used to set/ update an element in stack based on given index.
- **setElementAt (Object e, int index): (void, vector)**
  It is similar to set method.

- In stack class we can add or remove or update an element based on index because stack class inherits index properties from vector class.

```
import java.util.Stack;
public class Demo29
{
public static void main(String []args){
Stack<Object> s = new Stack<>();
System.out.println(s.isEmpty());
System.out.println(s.size());
System.out.println(s.capacity());
s.setSize(3);
System.out.println(s.size());
System.out.println(s.capacity());
s.set(0,"ganesh");
s.push("eswar");
s.push("venu");
s.push("janu");
s.push("naidu");
System.out.println(s);
System.out.println(s.peek());
s.pop();
System.out.println(s);
s.pop();
s.pop();
s.pop();
s.pop();
s.pop();
System.out.println(s);
}
}
```

-

```
C:\Users\ganieswar\Desktop\java_programs>java Demo29.java
true
0
10
3
10
[ganesh, null, null, eswar, venu, janu, naidu]
naidu
[ganesh, null, null, eswar, venu, janu]
[ganesh]
```

-

- Here when we setsize (3) the stack is filled with 3 null values and whenever we try to push another element it will be in index 3 after 3 null values
- And while we do set (0, “ganesh”) the null value in index 0 is overridden by the ganesh.
- Size is how many values the stack contains and it is different from the capacity.

- **REVERSING A STRING USING STACK:**

```
import java.util.Stack;
import java.util.Scanner;
public class Demo30{
public static void reverse(String str){
Stack<Object> s = new Stack<>();
for(int i=0;i<str.length();i++)
{
s.push(str.charAt(i));
}
String rev="";
for(int j=0;!s.isEmpty();j++)
{
rev=rev+s.pop();
}
System.out.print(rev);
}
public static void main(String []args){
Scanner sc = new Scanner(System.in);
System.out.print("enter the String:");
String str = sc.nextLine();
reverse(str);
}
}
```

-

```
C:\Users\ganieswar\Desktop\java_programs>java Demo30.java
enter the String:ganesh
hsenag
```

-

```
import java.util.Stack;
import java.util.Scanner;
import java.util.Arrays;
public class Demo30{
public static int [] reverse(int a[]){
Stack<Object> s = new Stack<>();
for(int i=0;i<a.length;i++){
s.push(a[i]);
}
for(int i=0;!s.isEmpty();i++){
a[i]=(int)s.pop();
}
return a;
}
public static void main(String []args){
Scanner sc = new Scanner(System.in);
int a[] = {10,20,30,40};
//System.out.println(reverse(a)); it will give the address of the array
System.out.println(Arrays.toString(reverse(a)));
}
}
//C:\Users\ganieswar\Desktop\java_programs>java Demo30.java
//[40, 30, 20, 10]
```

-

■ LinkedList:

- LinkedList is an implemented class which implements from list and Queue interface.
- It is defined in java.util package and it allows duplicates and null values.
- LinkedList will grow one node at a time.
- A **LinkedList** is a **sequential data structure**, meaning its elements are stored in a linear order. However, unlike an array, a LinkedList does not store elements in contiguous memory locations.
- LinkedList elements are accessed sequentially by following links (or pointers) from one node to the next.
- You cannot access elements directly by index in constant time, unlike an array.
- It also follows first in first out.
- By using linked list, we can easily perform insertion and deletion.
- In linked list deletion and insertion is easy because the elements are not disturbed, only the link will be disconnected and reconnected.

```
import java.util.LinkedList;
public class Demo31{
public static void main(String []args){
LinkedList<Object> ll =new LinkedList<>();
ll.add(10);
ll.add(20);
ll.add(30);
ll.add(40);
ll.add(50);
ll.add(60);
System.out.println(ll);
System.out.println(ll.size());
System.out.println(ll.remove(ll.get(2)));
System.out.println(ll.isEmpty());
System.out.println(ll);
ll.clear();
System.out.println(ll.isEmpty());
System.out.println(ll);
}
}
```

-

```
C:\Users\ganieswar\Desktop\java_programs>java Demo31.java
[10, 20, 30, 40, 50, 60]
6
true
false
[10, 20, 40, 50, 60]
true
[]
```

-

■ Iterable Interface:

- It is defined in java.lang and allows to traverse/ iterate element by element by using enhanced for loop or foreach loop.
- generics: used to specify the type to the object elements.
- These are used to create the general method which accepts any type and we can mention the type we required at the time of object creation.
- We can use generics for classes, methods and while object creating
- Public List<Type> method name (ArrayList<Type>) {}
- Classname <Type> var_name = new Classname<>()

```
import java.util.ArrayList;
import java.util.List;
public class Demo32{
public static List<Integer> numbers(ArrayList<Integer>l){
l.add(10);
l.add(20);
return l;
}
public static void main(String []args){
ArrayList<Integer> al = new ArrayList<>();
List<Integer> list = numbers(al);
System.out.println(list);
}
}
//[10, 20]
```

-

- FOREACH LOOP:

- It is advanced version of for loop and used to traverse element by element in the given array or collection.
- foreach will traverse only in forward direction and it cannot traverse in reverse order.

```
public class Demo33{
public static void main(String []args){
int a[] = {1,2,3,4,5};
for(int i:a){
//System.out.println(a[i]);//here i represents the elements in a
System.out.print(i+" ");
}
}
}
//1 2 3 4 5
```

-

- For ArrayList:

```
import java.util.ArrayList;
public class Demo33{
public static void main(String []args){
ArrayList<Integer> al=new ArrayList<>();
al.add(10);
al.add(20);
al.add(30);
for(int i:al){
System.out.print(i+" ");
}
}
}
//10 20 30
```

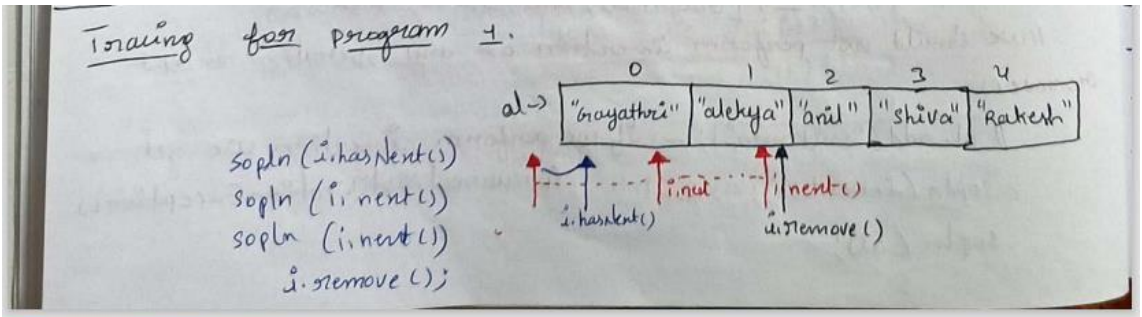
- Here we can't use object in generics because we use int i in the for loop, because we are storing elements of ArrayList to i which need to be the same type.

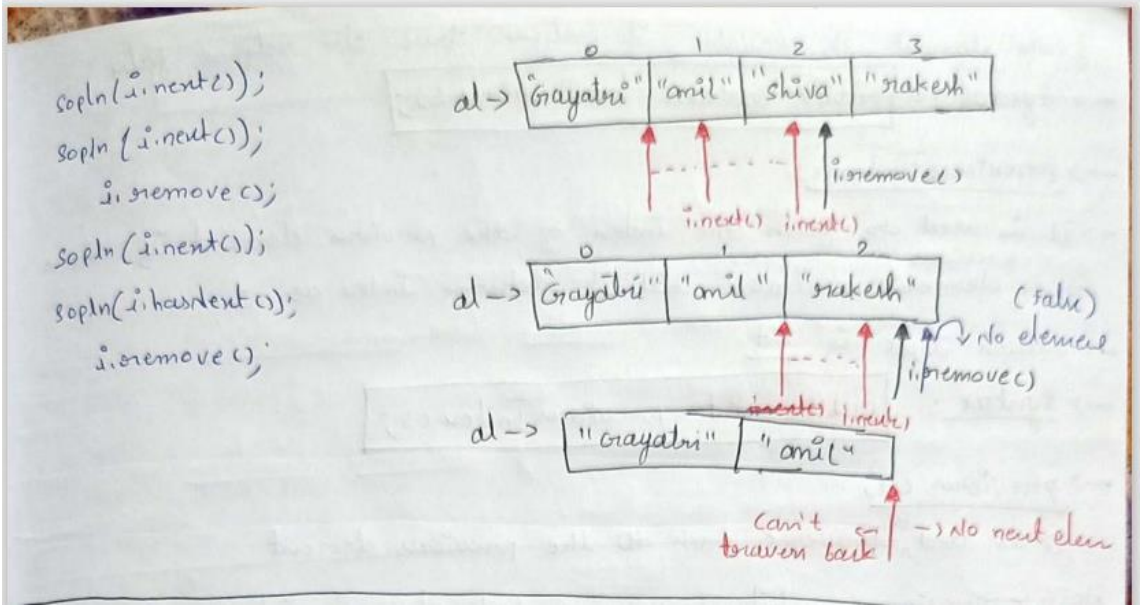
■ CURSORS IN JAVA: (3)

- 1.Iterator:

- Iterator interface is a cursor which is defined in java.util.
- Iterator interface is used to traverse element by element. It can only traverse in forward direction. It has 3 abstract methods:
- Public Object next ()
  Public boolean hasNext ()
  Public void remove ()
- next () is used to traverse to the next element, if there is no element available to traverse JVM will throw an exception called NoSuchElementException. Return type is object type.
- hasNext () is used to check the next element availability, if the element is available, it returns true else it returns false.
- remove () is used to remove the element where cursor is pointing. If there is no element available to remove operation JVM will throw IllegalStateException.
- HELPER METHODS:
- iterator () and List iterator () are used to help the iterators to traverse or loop
- Iterator Interface can be used in List, Queue.
- Iterator <> i = al.iterator (); here iterator () has lowercase 'i'.

```
import java.util.Iterator;
import java.util.ArrayList;
public class Demo34{
public static void main(String []args){
ArrayList<String> al =new ArrayList<>();
al.add("Gayatri");
al.add("Alekya");
al.add("Anil");
al.add("Shiva");
al.add("Rakesh");
System.out.println(al);//[Gayatri, Alekya, Anil, Shiva, Rakesh]
Iterator <String> i = al.iterator();
System.out.println(i.hasNext());//true
System.out.println(i.next());//Gayatri
System.out.println(i.next());//Alekya
i.remove();
System.out.println(al);//[Gayatri, Anil, Shiva, Rakesh]
//we cannot perform any insertion or any operation except remove.
//al.add("Srikanth");we will get ConcurrnetModificationException()
System.out.println(i.next());//Anil
System.out.println(al);//[Gayatri, Anil, Shiva, Rakesh]
System.out.println(i.next());//Shiva
i.remove();
System.out.println(al);//[Gayatri, Anil, Rakesh]
System.out.println(i.next());//Rakesh
System.out.println(i.hasNext());//false
//System.out.println(i.next());NoSuchElementException
i.remove();
System.out.println(al);//[Gayatri, Anil]
//i.remove();IllegaltateException
}
}
```

- 

- 

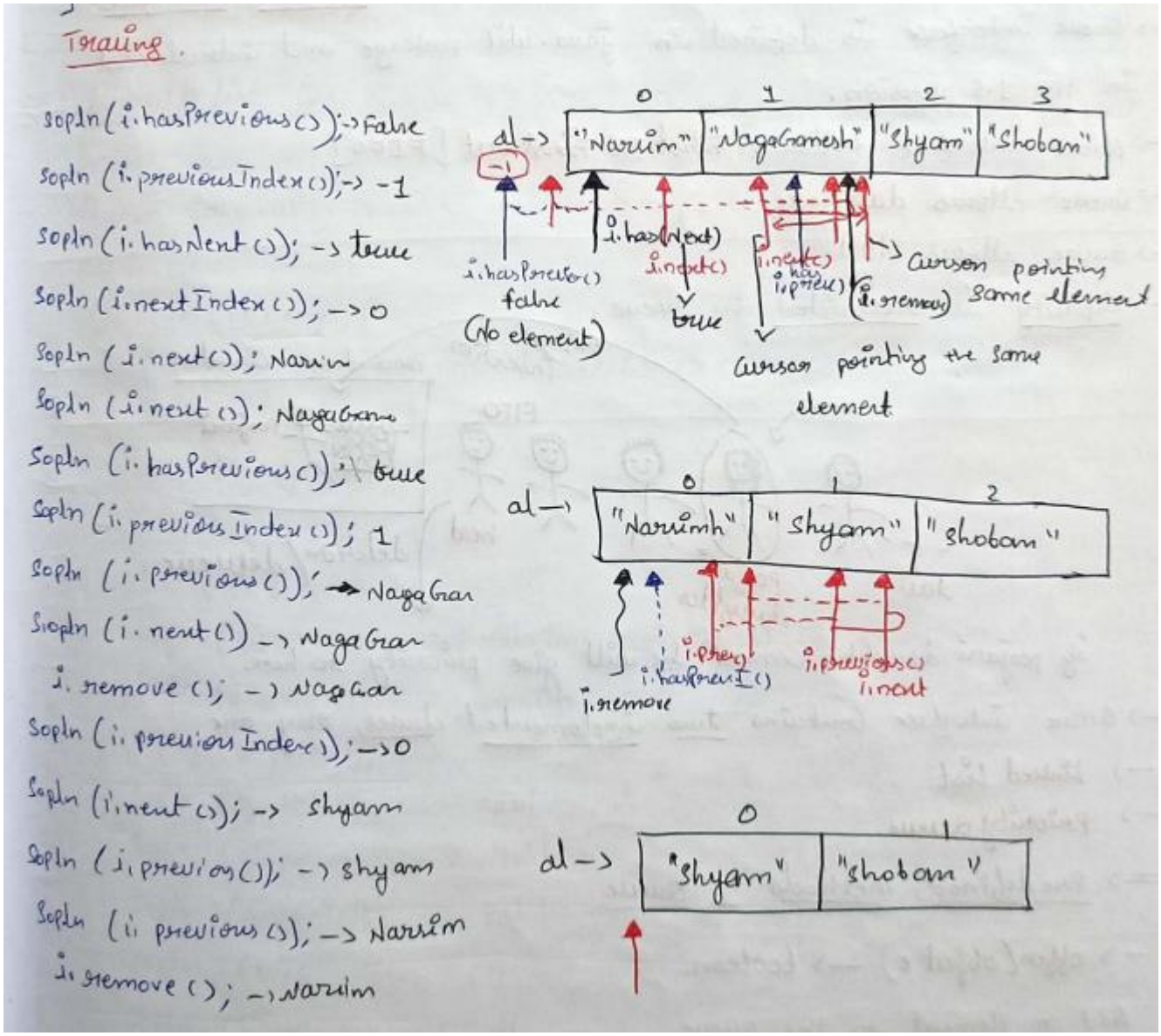
- Using While Loop

```
import java.util.Iterator;
import java.util.ArrayList;
public class Demo35{
public static void main(String []args){
ArrayList<Integer> al =new ArrayList<>();
al.add(10);
al.add(20);
al.add(30);
al.add(40);
al.add(50);
System.out.println(al);//[10, 20, 30, 40, 50]
Iterator<Integer> i = al.iterator();
while(i.hasNext()){
int j = (int)i.next();
System.out.println(j);
}
}
}
output:10
20
30
40
50
```

- **2.List Iterator**
- List Iterator interface is a cursor which is defined in java.util.
- It is used to traverse element by element in bidirectional (both forward and reverse).
- By using ListIterator method, we can fetch the index of the element.
- It contains seven Methods, they are:
  next ()
  hasNext ()
  nextIndex (): fetch the next element index, return type is int.
  remove ()
  previous (): used to traverse back to the previous element.
  hasPrevious (): Check availability of previous element, return type is Boolean
  previousIndex (): used to fetch the index of the previous element if there is no previous element available still it returns index as -1.
- NOTE: previous () while traversing back, always point the currently pointing element as the previous element.

```
import java.util.ListIterator;
import java.util.ArrayList;
public class Demo34{
public static void main(String []args){
ArrayList<String> al =new ArrayList<>();
al.add("Narsim");
al.add("Naga Ganesh");
al.add("shyam");
al.add("shoban babu");
ListIterator <String> i = al.listIterator();
System.out.println(i.hasPrevious());//false
System.out.println(i.previousIndex());//-1
//System.out.println(i.previous());//NoSuchElementException
System.out.println(i.hasNext());//true
System.out.println(i.nextIndex());//0
System.out.println(i.next());//Narsim
System.out.println(i.next());//Naga Ganesh
System.out.println(i.hasPrevious());//true
System.out.println(i.previousIndex());//1
System.out.println(i.previous());//Naga Ganesh
System.out.println(i.next());//Naga Ganesh
i.remove();
System.out.println(al);//[Narsim, shyam, shoban babu]
System.out.println(i.previousIndex());//0
System.out.println(i.next());//shyam
System.out.println(i.previous());//shyam
System.out.println(i.previous());//Narsim
i.remove();
System.out.println(al);//[shyam, shoban babu]
}
}
```

- List iterators used only in list Interface.



-

3.Enumeration

- Enumeration is a cursor/ iteration which is defined in java.util.
- Iterator interface is a replica for Enumeration.
- Enumeration is legacy which we are not using and it only traverse in forward direction.

■ QUEUE

- Queue is a sub-interface which extends from collection interface.
- Queue interface is defined in java.util.
- It follows First In First Out and allows duplicates and Null values.
- If Poojary daughter comes, he will give priority to her. Queue interface contains two implemented classes, they are:
  Linked List
  Priority Queue
- **Pre-defined methods of queue:**
  - **offer (Object e):** return type is boolean, used to add an element to the queue. Returns true if successful, otherwise false.
  - **poll ():** return type is Object and used to remove the first element from the queue.
  - **peek ():** return type is Object and used to fetch the first element from the queue, if element is available, it returns the element else it returns null.
  - **remove ():** return type is Object and used to remove the first element from the queue.
  - **add (Object e**): return type Boolean, used to add an element to the queue
- **NOTE:** If there is no element available in queue to perform deletion operation poll () returns null but remove () will throw NoSuchElementException.
-

```
import java.util.LinkedList;
import java.util.Queue;
public class Demo36{
public static void main(String []args){
Queue<String> q= new LinkedList<>();
q.offer("narsim");
q.offer("ravi");
q.offer("nikhil");
q.offer("sai kumar");
System.out.println(q);
System.out.println(q.poll());
System.out.println(q.peek());
System.out.println(q.poll());
System.out.println(q.poll());
System.out.println(q.poll());
System.out.println(q);
System.out.println(q.poll());
System.out.println(q.peek());
Queue <Integer> q1 = new LinkedList<>();
q1.offer(10);
System.out.println(q1.remove());
System.out.println(q1.peek());
System.out.println(q1.remove());
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo36.java
[narsim, ravi, nikhil, sai kumar]
narsim
ravi
ravi
nikhil
sai kumar
[]
null
null
10
null
Exception in thread "main" java.util.NoSuchElementException
        at java.base/java.util.LinkedList.removeFirst(LinkedList.java:274)
        at java.base/java.util.LinkedList.remove(LinkedList.java:689)
        at Demo36.main(Demo36.java:23)
```

- We cannot use like this:

```
C:\Users\ganieswar\Desktop\java_programs>java Demo36.java
Demo36.java:19: error: Queue is abstract; cannot be instantiated
Queue <Integer> q1 = new Queue<>();
                     ^
1 error
error: compilation failed
```

- **PriorityQueue:**
- PriorityQueue is an implemented class which implements from Queue interface
- It is defined in java.util package and it follows unordered queue (It stores elements randomly)
- It allows duplicates and null values.
- The default ordering is **natural ordering** (ascending order, smallest element has the highest priority).

```
import java.util.PriorityQueue;
public class Demo37{
public static void main(String []args){
PriorityQueue<Integer> q= new PriorityQueue<>();
q.offer(10);
q.offer(5);
q.offer(55);
q.offer(2);
System.out.println(q);
System.out.println(q.poll());
System.out.println(q.peek());
System.out.println(q.poll());
System.out.println(q.poll());
System.out.println(q.poll());
System.out.println(q);
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo37.java
[2, 5, 55, 10]
2
5
5
10
55
[]
```

- **q.offer (10); q.offer (5); q.offer (55); q.offer (2);**
  Adds elements **10**, **5**, **55**, and **2** to the priority queue using the offer () method.
  Internal heap structure (min-heap):
  The **PriorityQueue** uses a **binary heap** internally to maintain the elements.
  After adding all elements, the structure will be:

```
markdown

    2
   / \
  5   55
 /
10
```

- The smallest element (**2**) is at the root because it has the highest priority.
- Prints the **PriorityQueue**: [2, 5, 55, 10].
- Note: The elements may not appear fully sorted because **PriorityQueue** only guarantees that the smallest element (the head) will be at the front.

`System.out.println(q.poll());`

- `poll()` retrieves and removes the head of the queue, which is the element with the highest priority (**2**).
- After removing **2**, the heap reorganizes itself to maintain the priority order:

```
markdown                                                    Copy code

   5
  / \
 10  55
```

- Output: `2`
- The PriorityQueue does **not guarantee a sorted order** when printed.

- It only ensures that the smallest element (based on natural order or custom comparator) is always at the **head** (root of the heap).
- After each poll (), the heap is reorganized to maintain the priority order, which is why the next smallest element becomes the head.
- The smallest element is always at the top of the heap, ready to be accessed or removed with peek () or poll ().
- Arrange the array in ascending order: **(SORTING)**

```
import java.util.PriorityQueue;
import java.util.ArrayList;
import java.util.Arrays;
public class Demo37{
public static void main(String []args){
int a[] = {10,2,1,55,80};
System.out.println(Arrays.toString(a));
PriorityQueue<Integer> q= new PriorityQueue<>();
for(int i=0;i<a.length;i++){
q.offer(a[i]);
}
ArrayList<Integer> al= new ArrayList<>();
while(!q.isEmpty()){
al.add(q.poll());
}
System.out.println(al);

}
}
```

-

```
C:\Users\ganieswar\Desktop\java_programs>java Demo37.java
[10, 2, 1, 55, 80]
[1, 2, 10, 55, 80]
```

-

- **To perform sorting in arrays, we have a pre-defined method that is Arrays.sort (), it is method overloaded and static method.**
- To print an array Elements in array format we have a pre-defined method Arrays.toString () it is method overloaded and static method.
- Program to find maximum element and minimum element and second smallest and third largest element from an array.

```
import java.util.PriorityQueue;
import java.util.ArrayList;
import java.util.Arrays;
public class Demo37{
public static void main(String []args){
int a[] = {10,2,1,55,80};
System.out.println(Arrays.toString(a));
PriorityQueue<Integer> q= new PriorityQueue<>();
for(int i=0;i<a.length;i++){
q.offer(a[i]);
}
ArrayList<Integer> al= new ArrayList<>();
while(!q.isEmpty()){
al.add(q.poll());
}
System.out.println(al);
System.out.println("minimum: "+al.get(0));
System.out.println("max: "+al.get(al.size()-1));
System.out.println("second minimum: "+al.get(1));
System.out.println("third max: "+al.get(al.size()-3));
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo37.java
[10, 2, 1, 55, 80]
[1, 2, 10, 55, 80]
minimum: 1
max: 80
second minimum: 2
third max: 10
```

```
import java.util.PriorityQueue;
import java.util.ArrayList;
import java.util.Arrays;
public class Demo37{
public static void main(String []args){
int a[] = {10,2,1,55,80};
System.out.println(Arrays.toString(a));
Arrays.sort(a);
System.out.println(Arrays.toString(a));
}
}
//[10, 2, 1, 55, 80]
//[1, 2, 10, 55, 80]
```

■ Set

- Set interface is a sub-interface of collection interface.
- It is defined in java.util package.
- **It won't allow duplicate elements.**
- It won't store an element based on index.
- It allows null (only one null value) else we get NullPointerException.
- **How set maintains no duplicates?**
- While adding an element to the set, add () implicitly calls equals () to compare the elements.
- If equals () return true it means element is already available in set, so it won't allow to add that element to set.
- If equals () return false it means element is not available in set then it allows to add that element to set.
- Set interface contains 3 implemented classes, they are:
  HashSet
  LinkedHashSet
  TreeSet
- **HashSet:** HashSet is an implemented class which implements from set interface.
- It is defined in java.util package and it won't store elements based on index.
- It allows only one null element. It won't preserve the element order (it stores randomly)

```
import java.util.HashSet;
public class Demo38{
public static void main(String []args){
HashSet<Integer> hs = new HashSet<>();
hs.add(10);
hs.add(20);
hs.add(10);
hs.add(null);
hs.add(30);
hs.add(40);
System.out.println(hs);
hs.add(null);
System.out.println(hs);
}
}
//[null, 20, 40, 10, 30]
//[null, 20, 40, 10, 30]
```

-
- It stores randomly.

- **Linked HashSet:**
- Linked HashSet is a class which extends from HashSet.
- It is defined in java.util package, it won't allow duplicates.
- It won't store element based on index but **it will preserve the order.**
- It allows one null element.

```
import java.util.HashSet;
import java.util.LinkedHashSet;
public class Demo39{
public static void main(String []args){
HashSet<Integer> lh = new LinkedHashSet<>();
lh.add(10);
lh.add(20);
lh.add(10);
lh.add(null);
lh.add(30);
lh.add(40);
System.out.println(lh);
lh.add(null);
System.out.println(lh);
}
}
//[10, 20, null, 30, 40]
//[10, 20, null, 30, 40]
```

- **How to remove the duplicates from the given array?**

```
import java.util.HashSet;
import java.util.LinkedHashSet;
public class Demo40{
public static void main(String []args){
int a[] = {10,20,10,8,11,8};
HashSet<Integer> lh = new LinkedHashSet<>();
for(int i=0;i<a.length;i++){
lh.add(a[i]);
}
System.out.println(lh);
}
}
//[10, 20, 8, 11]
```

- **TreeSet:**
- TreeSet is an implemented class which implements from sorted set on navigable set.
- `It is defined in java.util package, it won't allow duplicates.
- It will store the elements in natural ordering (ascending order).
- It won't allow null (not even one null element), if we try to add null value JVM will throw NullPointerException.

```
import java.util.TreeSet;
import java.util.SortedSet;
public class Demo41{
public static void main(String []args){
SortedSet<Integer> ss = new TreeSet<>();
ss.add(10);
ss.add(40);
ss.add(20);
ss.add(50);
ss.add(10);
ss.add(30);
//ss.add(null);NullPointerException
System.out.println(ss);
}
}
//[10, 20, 30, 40, 50]
```

- **Write a program to remove the duplicates and sort in ascending order from the given array:**

```
import java.util.TreeSet;
import java.util.SortedSet;
public class Demo41{
public static void main(String []args){
int a[] = {10,20,10,8,11,8};
SortedSet<Integer> ss = new TreeSet<>();
for(int i=0; i<a.length;i++){
ss.add(a[i]);
}
System.out.println(ss);
}
}
//[8, 10, 11, 20]
```

- **Number of duplicates and non-duplicates in array.**

```
import java.util.TreeSet;
import java.util.SortedSet;
public class Demo41{
public static void main(String []args){
int a[] = {10,20,10,8,11,8};
SortedSet<Integer> ss = new TreeSet<>();
for(int i=0; i<a.length;i++){
ss.add(a[i]);
}
System.out.println(ss);
System.out.print("number of duplicates: "+(a.length - ss.size()));
System.out.println();
System.out.print("number of non-duplicates: "+ss.size());
}
}
//[8, 10, 11, 20]
//number of duplicates: 2
//number of non-duplicates: 4
```

-

- TreeSet pre-defined Methods:
  first (): return type is Object and used to fetch first or lowest element.
  last (): return type is Object and used fetch last or largest element.
  pollFirst (): return type is Object and used to remove first or lowest element.
  pollLast (): return type is Object and used to remove last or largest element.

```
import java.util.TreeSet;
import java.util.SortedSet;
public class Demo41{
public static void main(String []args){
int a[] = {10,20,10,8,11,8};
TreeSet<Integer> ss = new TreeSet<>();
for(int i=0; i<a.length;i++){
ss.add(a[i]);
}
System.out.println(ss);
System.out.println(ss.first());
System.out.println(ss.last());
ss.pollFirst();
System.out.println(ss);
ss.pollLast();
System.out.println(ss);
}
}
//[8, 10, 11, 20]
//8
//20
//[10, 11, 20]
//[10, 11]
```

-

■ MAP

- Map is an interface which is defined in java.util package.
- Map interface have no parent interface.
- It allows to store an element based on key and value pair.
- There is no index concept.
- One key and value pair in known as entry.
- Keys won't allow duplicates and values can be duplicate.
- Map interface which contains 4 implemented classes they are:
  HashMap
  Linked HashMap
  HashTable
  TreeMap
- **Pre-defined methods in map interface:**
  - **put (Object k, Object v):** Object – It is used to add an entry to the map interface.
  - **clear ():** Void – used to clear the map.
  - **containsKey (Object key):** boolean – used to check the given key is available or not.
  - **containsValue (Object value):** boolean – used to check the given key is available or not.
  - **get (Object key):** String - It is used to fetch the value based on the key.
  - **remove (Object key):** String – used to remove an entry based on key.
  - **remove (object key, Object value):** boolean – used to remove an entry based on key and value pair.
  - **replace (integer key, String value):** String – is used to replace based on key.
  - **replace (Object key, Object oldvalue, Object newvalue):** Boolean – is used to replace value based on key and value.
  - **size ():**
  - **values ():** collection of objects – used to fetch values from map
- **KeySet ():**
- It is used to fetch the keys from the map interface and store in set interface.
- It's return type is set<Object>

```
import java.util.HashMap;
import java.util.Set;
public class Demo43{
public static void  main(String []args){
HashMap<Integer,Integer> h = new HashMap<>();
h.put(1,10);
h.put(2,20);
h.put(3,30);
h.put(4,10);
System.out.println(h);
Set<Integer> keys = h.keySet();
System.out.println(keys);
}
}
//{1=10, 2=20, 3=30, 4=10}
//[1, 2, 3, 4]
```

- **entrySet ():**
- It is used to convert all map entries into set type of entries.
- It's return type is Set<Entry <Object_k, Object_v>>
- Entry interface can fetch by using map interface
- Map.entry<Object, Object>

```
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
public class Demo44{
public static void  main(String []args){
Map<Integer,Integer> h = new LinkedHashMap<>();
h.put(1,10);
h.put(2,20);
h.put(3,30);
h.put(4,10);
System.out.println(h);
Set<Entry<Integer,Integer>> entry = h.entrySet();
System.out.println(entry);
}
}
//{1=10, 2=20, 3=30, 4=10}
//[1=10, 2=20, 3=30, 4=10]
```

- **Entry Interface:**
- Entry is an interface which is defined in java.util.Map.
- It is used to fetch the map interface keys and values.
- Entry Interface contains pre-defined methods, they are:

- getKey (): It is used to fetch the keys from the entry interface. It's return type is Object.
- getValue (): used to fetch the values from the entry interface. It's return type is Object.

```
import java.util.Map;
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;
public class Demo45{
public static void  main(String []args){
Map<Character,String> h = new HashMap<>();
h.put('a',"apple");
h.put('b',"ball");
h.put('c',"cat");
System.out.println(h);
Set<Entry<Character,String>> entry = h.entrySet();
System.out.println(entry);
for(Map.Entry<Character, String> me:entry){
System.out.println("key: "+me.getKey()+".....>"+"value: "+me.getValue());
}
}
}
```

-
```
C:\Users\ganieswar\Desktop\java_programs>java Demo45.java
{a=apple, b=ball, c=cat}
[a=apple, b=ball, c=cat]
key: a.....>value: apple
key: b.....>value: ball
key: c.....>value: cat
```

- **HashMap:**
- It is an implemented class which implements from map.
- It is defined in java.util package.
- It won't preserve the order and keys won't allow null but values allow null.
- Keys cannot be duplicate; values can be duplicate.

```
import java.util.HashMap;
public class Demo42{
public static void  main(String []args){
HashMap<String,Integer> h = new HashMap<>();
h.put("apple",1);
h.put("ball",2);
h.put("map",2);
h.put("one",7);
h.put("set",1);
System.out.println(h);
}
}
//{apple=1, ball=2, set=1, one=7, map=2}
```

- No certain order,
- Here we can have values duplicate but not keys.

- **Linked HashMap:**
- Linked hashmap extended class which extends from hashmap.
- It is defined in java.util and it will preserve the order.

```
import java.util.LinkedHashMap;
public class Demo42{
public static void  main(String []args){
LinkedHashMap<String,Integer> h = new LinkedHashMap<>();
h.put("apple",1);
h.put("ball",2);
h.put("map",2);
h.put("one",7);
h.put("set",1);
System.out.println(h);
}
}
//{apple=1, ball=2, map=2, one=7, set=1}
```

```
import java.util.LinkedHashMap;
public class Demo42{
public static void  main(String []args){
LinkedHashMap<Integer,Integer> h = new LinkedHashMap<>();
h.put(1,10);
h.put(2,20);
h.put(3,30);
h.put(4,40);
System.out.println(h);                  //{1=10, 2=20, 3=30, 4=40}
System.out.println(h.containsKey(2));   //true
System.out.println(h.containsValue(100));//false
System.out.println(h.remove(3));        //30
System.out.println(h);                  //{1=10, 2=20, 4=40}
System.out.println(h.remove(2,50));     //false
System.out.println(h);                  //{1=10, 2=20, 4=40}
System.out.println(h.remove(4,40));     //true
System.out.println(h);                  //{1=10, 2=20}
h.put(5,50);
System.out.println(h);                  //{1=10, 2=20, 5=50}
System.out.println(h.replace(5,60));    //50
System.out.println(h);                  //{1=10, 2=20, 5=60}
System.out.println(h.replace(5,50,70)); //false
System.out.println(h);                  //{1=10, 2=20, 5=60}
System.out.println(h.replace(5,60,70)); //true
System.out.println(h);                  //{1=10, 2=20, 5=70}
System.out.println(h.get(5));           //70
System.out.println(h.size());           //3
System.out.println(h.values());         //[10, 20, 70]
}
}
```

- **HashTable:**
- HashTable is an implemented class which implements from Map interface.
- It is defined in java.util package.
- HashMap and HashSet implemented classes are replicas of HashTable.
- HashTable is a legacy, synchronized and it is threadsafe.
- It won't allow null, if we try to declare key or value as null: it will throw NullPointerException.
- It won't preserve the order.

```
import java.util.Hashtable;
public class Demo46{
public static void  main(String []args){
Hashtable<Integer,Integer> h = new Hashtable<>();
h.put(1,10);
h.put(2,20);
h.put(3,30);
//h.put(null,20);NullPointerException
//h.put(4,null);NullPointerException
//h.put(null,null);NullPointerException
System.out.println(h);
}
}
//{3=30, 2=20, 1=10}
```

- **TreeSet:**
- TreeSet is an implemented class which implements from navigable or sorted interface.
- It is defined java.util package.
- It arranges an entry in natural Ordering (ascending order) based on key.
- It won't allow null, if we try to add we get NullPointerException.

```
import java.util.TreeMap;
public class Demo47{
public static void  main(String []args){
TreeMap<Integer,Integer> h = new TreeMap<>();
h.put(1,10);
h.put(2,20);
h.put(3,30);
//h.put(null,20);NullPointerException
//h.put(4,null);NullPointerException
//h.put(null,null);NullPointerException
System.out.println(h);
}
}
//{1=10, 2=20, 3=30}
```

- **Write a program to find the occurrence or frequency of each element from the given array elements.**
- Here we need to use key-value pair, one for value and other for frequency.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
public class Demo49{
public static void main(String []args){
int arr[] = {1,2,2,3,3,3,3,4,4,5};
HashMap<Integer,Integer> h = new HashMap<>();
for(int i:arr){
if(h.containsKey(i)){
h.put(i,h.get(i)+1);
}
else{
h.put(i,1);
}
}
System.out.println(h);
}
}
//{1=1, 2=2, 3=4, 4=2, 5=1}
```

- The element of the array must be a key for the HashMap because no duplicating the elements here because we need to find the frequency of different elements.
- In the first case there is nothing in the hash map so we go for else block which is first entry for the HashMap key is the element in array and its frequency is 1.
- Too easy concept if we understand. Here get (i) gives the value to the element of array which is a key because value is nothing but the frequency, so increment it by 1. The element is already present in the HashMap (if condition) so we can get the frequency to increment its value.
- And important thing is we can use put method also to override or replace the previous one:

```
import java.util.HashMap;
import java.util.Set;
public class Demo43{
public static void  main(String []args){
HashMap<Integer,Integer> h = new HashMap<>();
h.put(1,10);
h.put(2,20);
h.put(3,30);
h.put(4,10);
h.put(2,30);
System.out.println(h);
}
}
//{1=10, 2=30, 3=30, 4=10}
```

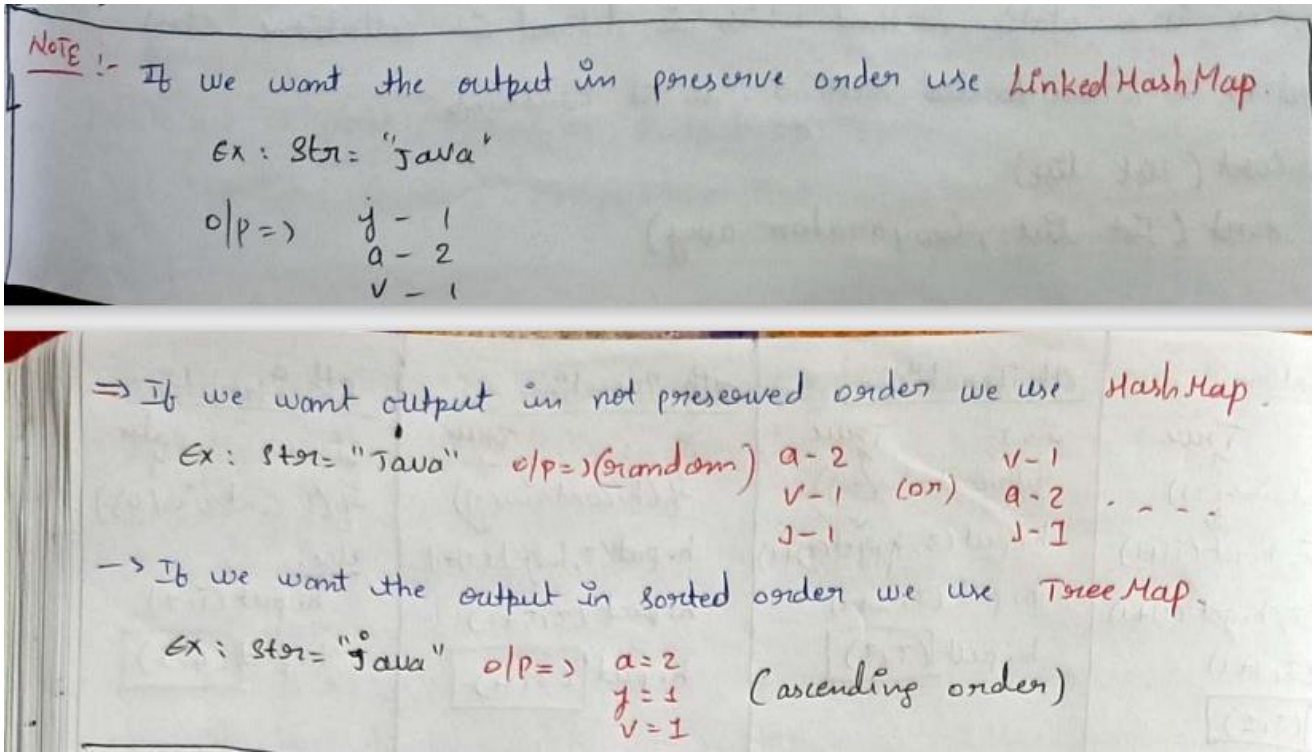
- Write a program to find the frequency of each character from the given string.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
public class Demo49{
public static void main(String []args){
String str="aabcabcc";
HashMap<Character,Integer> h = new HashMap<>();
for(int i=0;i<str.length();i++){
if(h.containsKey(str.charAt(i))){
h.put(str.charAt(i),h.get(str.charAt(i))+1);
}
else{
h.put(str.charAt(i),1);
}
}
System.out.println(h);
}
}
```

- //{a=3, b=2, c=3}
- Or

```
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
public class Demo49{
public static void main(String []args){
String str="aabcabcc";
char ch[] = str.toCharArray();
HashMap<Character,Integer> h = new HashMap<>();
for(char c: ch){
if(h.containsKey(c)){
h.put(c,h.get(c)+1);
}
else{
h.put(c,1);
}
}
System.out.println(h);
}
}
```

- //{a=3, b=2, c=3}

- 

- **Comparator Interface**:
- Java Comparator interface is used to sort the objects of a user-defined class.
- This interface is found in java.util and it contains 2 methods. They are:
  compare (Object obj1, Object obj2)
  equals (Object element)
- **compare (Object obj1, Object obj2):**
  It compares the first object with the second object. Return type is int type.
- **Collection class:** It is defined in java.util package and used to storing data, searching, sorting and insertion, deletion and updating of data on the group of elements.
- Collections of class contains so many predefined methods like sort (), reverse ()
- **sort ():** it is a static method which is defined in collections class.
- It is an overloaded method so it contains
  sort (List list)
  sort (List list, Comparator)
- compareTo (String str) is used to compare String type data and returns int type. It is present in string class.
- **PROGRAM FOR COMPARATOR () TO COMPARE STUUDENT DETAILS BASED ON ID, NAME, MARKS IN ASCENDING ORDER**:

```
public class Demo50{
String name;
int id;
double marks;
public Demo50(String name,int id,double marks){
this.name=name;
this.id=id;
this.marks=marks;
}
//@override
public String toString(){
return "Demo50[name= "+name+"  id= "+id+" marks= "+marks+"]";
}
}
```

```
import java.util.Comparator;
public class SortById implements Comparator<Demo50>{

//@override
public int compare(Demo50 d1, Demo50 d2){
return d1.id-d2.id;
}
}
```

```
import java.util.Comparator;
public class SortByMarks implements Comparator<Demo50>{
//@override
public int compare(Demo50 d1, Demo50 d2){
return (int)(d1.marks-d2.marks);
}
}
```

```
import java.util.Comparator;
public class SortByName implements Comparator<Demo50>{
//@override
public int compare(Demo50 d1, Demo50 d2){
return d1.name.compareTo(d2.name);
}
}
```

```
import java.util.ArrayList;
import java.util.Collections;
public class Demo51{
public static void main(String []args){
Demo50 d1 = new Demo50("ganesh",101,96.5);
Demo50 d2 = new Demo50("naidu",108,95.5);
Demo50 d3 = new Demo50("eswar",102,97.5);
Demo50 d4 = new Demo50("venu",104,88.5);
Demo50 d5 = new Demo50("jaanu",110,99.5);
ArrayList<Demo50> al = new ArrayList<>();
al.add(d1);
al.add(d2);
al.add(d3);
al.add(d4);
al.add(d5);
System.out.println("***unsorted order***");
for(Demo50 dd:al){
System.out.println(dd);
}
System.out.println("***sort based on id***");
Collections.sort(al,new SortById());
System.out.println("***sorted order***");
for(Demo50 id:al){
System.out.println(id);
}
System.out.println("***sorted based on name***");
Collections.sort(al, new SortByName());

for(Demo50 name:al){
System.out.println(name);
}
System.out.println("***sort based on marks***");
Collections.sort(al,new SortByMarks());
for(Demo50 marks:al){
System.out.println(marks);
}
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo51.java
***unsorted order***
Demo50[name= ganesh  id= 101 marks= 96.5]
Demo50[name= naidu  id= 108 marks= 95.5]
Demo50[name= eswar  id= 102 marks= 97.5]
Demo50[name= venu  id= 104 marks= 88.5]
Demo50[name= jaanu  id= 110 marks= 99.5]
***sort based on id***
***sorted order***
Demo50[name= ganesh  id= 101 marks= 96.5]
Demo50[name= eswar  id= 102 marks= 97.5]
Demo50[name= venu  id= 104 marks= 88.5]
Demo50[name= naidu  id= 108 marks= 95.5]
Demo50[name= jaanu  id= 110 marks= 99.5]
***sorted based on name***
Demo50[name= eswar  id= 102 marks= 97.5]
Demo50[name= ganesh  id= 101 marks= 96.5]
Demo50[name= jaanu  id= 110 marks= 99.5]
Demo50[name= naidu  id= 108 marks= 95.5]
Demo50[name= venu  id= 104 marks= 88.5]
***sort based on marks***
Demo50[name= venu  id= 104 marks= 88.5]
Demo50[name= naidu  id= 108 marks= 95.5]
Demo50[name= ganesh  id= 101 marks= 96.5]
Demo50[name= eswar  id= 102 marks= 97.5]
Demo50[name= jaanu  id= 110 marks= 99.5]
```

- To perform the descending order, first perform ascending order and by using reverse method perform descending order.

■ Multi-tasking:

- Executing multiple/ several tasks simultaneously is known as multi-tasking.
- Example is classroom student- listening class, writing notes, using mobile, sleeping.
- Multi-tasking classified into two types:
  Process based multi-tasking
  Thread based multi-tasking
- **Process based multi-tasking:** Executing several tasks simultaneously, where each task is separate independent program (process) is called process based multi-tasking.
- Process based multi-tasking is best suitable for OS level.
- **Thread based multi-tasking:** Executing several tasks simultaneously, where each task is separate independent part of some program is called thread based multi-tasking.
- Thread based multi-tasking is best suitable yet programmatic level.
- Main task of Thread/Process based multi-tasking is to reduce the response time of CPU and increasing the performance of the system.
- Important applications areas of multi-threading are:
  To develop multi-media graphics
  To develop animations
  To develop video games
  To develop web servers

■ MULTI-THREADING:

- It is a process of executing multiple thread simultaneously/ parallelly is known as multi-threading.
- **THREAD:** Thread is a flow or path or direction of execution, every thread has a separate job. Thread is a light weight process.
- **How to define a thread:**
  By extending Thread class
  By implementing Runnable interface
- **By extending Thread class:** Thread is a pre-defined class, which is defined in java.lang package.
- Thread class implements from Runnable interface
- run () is overridden in Thread class from Runnable interface and we are overriding run () to provide an implementation/ job of a thread to child thread.
- Every program contains one main thread and that main thread is responsible to call main () to start the execution.
- Main thread creates child thread Object.

- **start () method internally calls child class run ().**

```
public class MyThread extends Thread{
public void run(){
for(int i=1;i<=5;i++){
try{
Thread.sleep(2000);
System.out.println("child "+Thread.currentThread().getName()+" ...> "+i);
}catch(InterruptedException e){
e.printStackTrace();
}
}
}
}
```

```
public class Demo52{
public static void main(String []args) throws InterruptedException{
MyThread t = new MyThread();
t.start();
MyThread t1 = new MyThread();
t1.start();
for(int i=0;i<=4;i++){
Thread.sleep(2000);
System.out.println("Main Thread");
}
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo52.java
Main Thread
child Thread-0 ...> 1
child Thread-1 ...> 1
Main Thread
child Thread-0 ...> 2
child Thread-1 ...> 2
Main Thread
child Thread-0 ...> 3
child Thread-1 ...> 3
Main Thread
child Thread-0 ...> 4
child Thread-1 ...> 4
Main Thread
child Thread-0 ...> 5
child Thread-1 ...> 5
```

- In the above program there are 3 threads available, main thread which is created by JVM.
- 2 child Threads which is created by developer.

- While coming to execution three thread also execute simultaneously but we cannot predict which thread will execute first and which thread will execute last.
- All the Threads executes simultaneously.
- sleep (2000) means 2 second gap.
- Thread execution time is scheduled by Thread Scheduler (CPU).
- In above program, we can directly access the method of thread class with the object of MyThread class because of extends and it is not possible when we use implements runnable interface.

- **currentThread () method:** It is a static method which is defined in Thread class, which is used to fetch the currently running thread details like name of thread, priority of the thread, old name of the main Thread.
- It's return type is Thread.
- **start () method**: It is a non-static method which is defined in thread class and used to internally invoke the run () method to start the execution of the Thread.
- It's return type is void.
- **run () method**: It is an override method from runnable interface in Thread class.
- run () is used to start a new thread. It's return type is void.
- It is internally invoked by start ().

```
public class MyThread extends Thread implements Runnable{
public void run(){
System.out.println(Thread.currentThread());
}
}
```

```
public class Demo52{
public static void main(String []args) throws InterruptedException{
Thread t = Thread.currentThread();
System.out.println(t);
MyThread t1 = new MyThread();
t1.start();
}
}

//Thread[main,5,main]
//Thread[Thread-0,5,main]
```

- **By implementing Runnable interface:** Runnable interface is defined in java.lang package and is a functional interface because it contains only one abstract method that is run ().

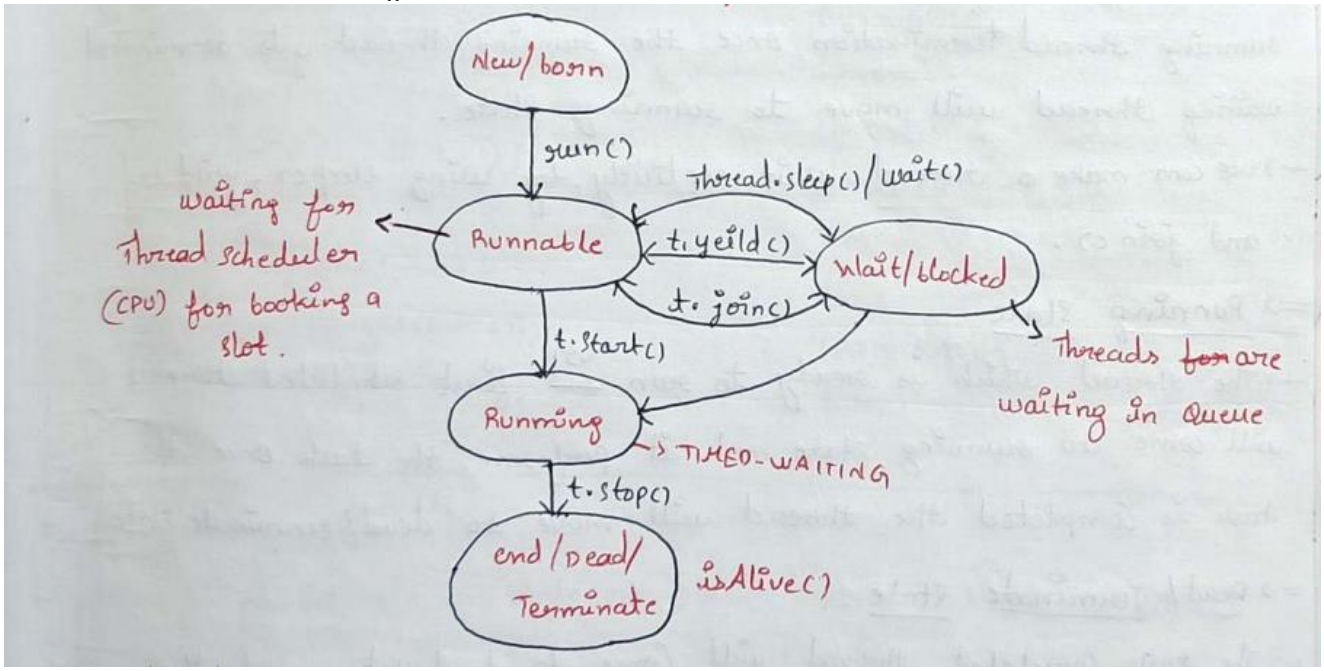
```
public class MyThread implements Runnable{
public void run(){
for(int i=1;i<=5;i++){
System.out.println(Thread.currentThread().getName()+" ...> "+i);
}
}
}
```

```
public class Demo52{
public static void main(String []args){
MyThread t1 = new MyThread();
Thread t = new Thread(t1);
t.start();
//t1.start();start method is only for thread class not for classes
implementing runnable
Thread t2 = new Thread(t1);//we can write new MyThread() istead of t1
t2.start();
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo52.java
Thread-0 ...> 1
Thread-1 ...> 1
Thread-0 ...> 2
Thread-1 ...> 2
Thread-0 ...> 3
Thread-1 ...> 3
Thread-0 ...> 4
Thread-1 ...> 4
Thread-0 ...> 5
Thread-1 ...> 5
```

■ THREAD LIFE CYCLE

- Thread t = new Thread ();



- In Thread life cycle there are 5 states, they are:
  New/born state
  Runnable state
  Waiting/ blocked state
  Running state
  Dead/end/terminate state
- **New/born state:** when a thread gets created (Object Creation) new thread will be born, which is ready to run a thread will move to runnable state.
- **Runnable state:** The thread is ready to run but nor running, the threads are waiting for the Thread Scheduler (CPU) to book the slot.
- If there is only one thread then it will move to running state directly, if there are multiple threads and time scheduled, first scheduled thread will move to running state and other threads are moved to waiting/ blocked state.
- **Waiting/blocked state:** In waiting state the scheduled threads are waiting for currently running thread termination, once the running thread gets terminates waiting thread will move to running state.
- We can make a thread wait explicitly by using sleep (), yield () and join ().
- **Running state:** The Thread which is ready to run or first scheduled thread will come to running state and it perform the task once the task is completed the thread will move to dead/terminate state.
- **Dead/terminate state:** The task completed thread will come to dead state, and the dead thread cannot be forces to perform the task again, if we try to call then JVM will throw IllegalThreadStateException.

- **getState () method:** It is defined in state (eNum) and state enum class is defined in Thread class.
    - It is used to check the current state of the thread.
    - If the Thread is created -> NEW
    - If the Thread is ready to run -> RUNNABLE
    - If the Thread is running -> TIMED-WAITING
    - If the Thread is dead ->TERMINATED
    - It's return type is state.
    - We can store the state like: NewThread n = new NewThread(); State status = t.getState ();System.out.println(status);
- **isAlive () method**: It is used to check the availability of the Thread.
  It's return type is boolean
  If the thread is alive, it returns true; if the Thread is dead then it returns false.
- **Program for Thread life cycle:**

```
public class MyThread extends Thread{
public void run(){
for(int i=1;i<=2;i++){
try{
System.out.println(Thread.currentThread().getName()+" ...> "+i);
Thread.sleep(2000);
}
catch(InterruptedException e)
{
e.printStackTrace();
}
}
}
}
```

-

```
public class NewThreadMain{
public static void main(String []args) throws InterruptedException{
MyThread t = new MyThread();
System.out.println(t.getState());
System.out.println(t.isAlive());
t.start();
System.out.println(t.getState());
System.out.println(t.isAlive());
Thread.sleep(3000);
System.out.println(t.getState());
System.out.println(t.isAlive());
//t.stop();
t.interrupt();//using terminate instead of stop to avoid error
Thread.sleep(2000);
System.out.println(t.getState());
System.out.println(t.isAlive());
}
}
```

-

```
C:\Users\ganieswar\Desktop\java_programs>java NewThreadMain.java
NEW
false
RUNNABLE
true
Thread-0 ...> 1
Thread-0 ...> 2
TIMED_WAITING
true
java.lang.InterruptedException: sleep interrupted
        at java.base/java.lang.Thread.sleep(Native Method)
        at MyThread.run(MyThread.java:6)
TERMINATED
false
```

- The thread is not considered “alive” until start () is called.
- **InterruptedException Handling**: The thread exits gracefully after being interrupted while sleeping. So, we need to throws exception if we use sleep () in the program.
- **Safe Shutdown**: The interrupt () method signals the thread to stop during sleep, triggering an exception.
- **Why RUNNABLE true Appears Before Thread Execution Output:**

  1.Thread State and start () Behaviour:

  When you call t. start (), the thread is moved to the RUNNABLE state and marked as alive.

  At this point, the thread is *ready* to run, but it may not immediately begin execution. The actual start depends on the thread scheduler.

  2.Console Output Timing:

  System.out.println(t. getState ()) and System.out.println(t. isAlive ()) are executed by the main thread right after calling t. start ().

  The main thread prints RUNNABLE and true before the new thread (Thread-0) starts printing its output.

  3.Thread Scheduling:

  The Java thread scheduler decides when Thread-0 actually starts running. There's a slight delay between the thread being marked as RUNNABLE and it actually executing its run () method.

  In the meantime, the main thread continues and prints the state and liveness checks.

  4.Buffered Output and Flushing:

  System.out.println uses buffered output. The main thread's output (RUNNABLE true) might reach the console before the output from Thread-0, even though Thread-0 may have started running.

- **Thread Priority:** Thread priority is used to fetch the priority and modify the priority of the thread.
  There are three types of priority. They are
  MIN_PRIORITY =>1
  NORM_PRIORITY=>5
  MAX_PRIORITY=>10
- These priorities are pre-defined variables and it is final, we cannot modify them.
- Every Thread default priority is 5.
- We can modify the Thread Priorities but the priority range must be between 1 to 10 only. If we try to modify the ThreadPriority while exceeding range we get IllegalArgumentException.
- getPriority () is used to fetch the priority of the thread and setPriority () is used to modify the priority of the thread.

```
public class Demo53{
public static void main(String []args){
Thread t = new Thread();
Thread t1 = new Thread();
Thread t2 = new Thread();
System.out.println(t.getPriority());
System.out.println(t1.getPriority());
System.out.println("MIN_PRIORITY - "+t.MIN_PRIORITY);
System.out.println("NORM_PRIORITY - "+t.NORM_PRIORITY);
System.out.println("MAX_PRIORITY - "+t.MAX_PRIORITY);
t.setPriority(7);
System.out.println(t.getPriority());
}
}
```

-

```
C:\Users\ganieswar\Desktop\java_programs>java Demo53.java
5
5
MIN_PRIORITY - 1
NORM_PRIORITY - 5
MAX_PRIORITY - 10
7
```

-

- **getId () method:** return type is int
  - It used to fetch the Id of the current thread
  - First thread Id is provided randomly and from the next thread it provides sequentially by adding 1.
  - We cannot modify the Id of the threads.

- **getName () method:** return type is String
  - It is used to fetch the current thread name.
  - The first thread Name always start from Thread-0
  - The second thread Name starts like Thread-1

- **setName () method:** return type is void
  - It is used to modify the current thread name.
  - setName (String arg)

```
public class Demo53{
public static void main(String []args){
Thread t = new Thread();
Thread t1 = new Thread();

System.out.println(t.getId());
System.out.println(t1.getId());
System.out.println(t.getName());
System.out.println(t1.getName());
t.setName("ganesh-Thread");
System.out.println(t.getName());
t1.setName("eswar-Thread");
System.out.println(t1.getName());
}
}
```

-

```
C:\Users\ganieswar\Desktop\java_programs>java Demo53.java
15
16
Thread-0
Thread-1
ganesh-Thread
eswar-Thread
```

-

- We can prevent a thread execution by using following methods:
  yield ()
  join ()
  sleep ()
- **yield () method:**
  - yield method is used to pause the currently running thread to give the chance for the remaining threads with same priority.
  - Yield () is a static method.
  - Yield () is a native method and we can declare native only for methods.
  - Native indicates that the method is implemented in native code using java native interface.
  - Public static native void yield ()
  - If there is no waiting thread or the waiting thread priority is low then the same thread continues its execution.
  - If there are multiple thread waiting with the same priority, we cannot expect which thread is going to execute because scheduled time slot is provided by time schedule (CPU).
  - The yielded/paused thread execution also cannot be expected because it is also scheduled by CPU only.

- ```
  public class Demo54 extends Thread{
  public void run() {
  for(int i=1;i<=5;i++){
  System.out.println(Thread.currentThread().getName()+"...>"+i);
  Thread.yield();
  }
  }
  }
  ```

- ```
  public class Demo55{
  public static void main(String []args){
  Demo54 d = new  Demo54();
  d.start();
  for(int i=1;i<=5;i++){
  System.out.println(Thread.currentThread().getName()+"...>"+i);
  }
  }
  }
  ```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo55.java
Thread-0...>1
main...>1
Thread-0...>2
main...>2
Thread-0...>3
main...>3
Thread-0...>4
main...>4
Thread-0...>5
main...>5

C:\Users\ganieswar\Desktop\java_programs>java Demo55.java
Thread-0...>1
main...>1
Thread-0...>2
main...>2
Thread-0...>3
main...>3
Thread-0...>4
Thread-0...>5
main...>4
main...>5
```

- The output varies here.
- **join () method:**
  - If a thread wants to wait until another thread gets executed completely or based on specific time to wait, we approach join () method.
  - Join method is a final method and overloaded method.
  - Join method will throw checked exception called InterruptedException which is mandatory to handle or throws Method Signature.
  - Public final void join () throws InterruptedException
  - Public final void join (long millisec) throws InterruptedException
  - Public final void join (long millisec, int nanosec) throws InterruptedException
- **Case 1: Main Thread have to wait until child thread execution completes.**

```
public class Demo54 extends Thread{
public void run() {
for(int i=1;i<=5;i++){
System.out.println(Thread.currentThread().getName()+"...>"+i);
}
}
}
```

```
public class Demo55{
public static void main(String []args) throws InterruptedException{
Demo54 d = new  Demo54();
d.start();
d.join();
for(int i=1;i<=5;i++){
System.out.println(Thread.currentThread().getName()+"...>"+i);
}
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo55.java
Thread-0...>1
Thread-0...>2
Thread-0...>3
Thread-0...>4
Thread-0...>5
main...>1
main...>2
main...>3
main...>4
main...>5
```

- **Case 2: child thread has to wait until main thread get executed.**

```
public class Demo54 extends Thread{
Thread t = Thread.currentThread();
public void run(){
try{
t.join();
}catch(InterruptedException e){
e.printStackTrace();
}
for(int i=1;i<=5;i++){
System.out.println(Thread.currentThread().getName()+"...>"+i);
}
}
}
```

```
public class Demo55{
public static void main(String []args) throws InterruptedException{
Demo54 d = new  Demo54();
d.start();
for(int i=1;i<=5;i++){
System.out.println(Thread.currentThread().getName()+"...>"+i);
}
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo55.java
main...>1
main...>2
main...>3
main...>4
main...>5
Thread-0...>1
Thread-0...>2
Thread-0...>3
Thread-0...>4
Thread-0...>5
```

- **Case 3: join thread is waiting for only specified time, it is not waiting completely until another thread get executed.**

```
public class Demo54 extends Thread{
Thread t = Thread.currentThread();
public void run(){
try{
t.join(2000);
}catch(InterruptedException e){
e.printStackTrace();
}
for(int i=1;i<=5;i++){
System.out.println(Thread.currentThread().getName()+"...>"+i);
}
}
}
```

```
public class Demo55{
public static void main(String []args) throws InterruptedException{
Demo54 d = new  Demo54();
d.start();
for(int i=1;i<=5;i++){
System.out.println(Thread.currentThread().getName()+"...>"+i);
Thread.sleep(1000);
}
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo55.java
main...>1
main...>2
Thread-0...>1
Thread-0...>2
Thread-0...>3
Thread-0...>4
Thread-0...>5
main...>3
main...>4
main...>5
```

- We can achieve deadlock concept by using join () method.

■ **Dead Lock:**

- If two threads are waiting for each other forever to execute, this type of infinite waiting is called deadlock.
- There is no resolution technique for deadlock but several prevention techniques are available.
- **Case 1: child thread is waiting for main thread and main thread is waiting for child thread forever.**

```
public class Demo54 extends Thread{
Thread t = Thread.currentThread();
public void run(){
try{
t.join();
}catch(InterruptedException e){
e.printStackTrace();
}
for(int i=1;i<=5;i++){
System.out.println(Thread.currentThread().getName()+"...>"+i);
}
}
}
```

```
public class Demo55{
public static void main(String []args) throws InterruptedException{
Demo54 d = new  Demo54();
d.start();
d.join();
for(int i=1;i<=5;i++){
System.out.println(Thread.currentThread().getName()+"...>"+i);
Thread.sleep(1000);
}
}
}
```

- **Case 2: A thread waiting itself for the execution forever.**

```
public class Demo55{
public static void main(String []args) throws InterruptedException{
Thread d = Thread.currentThread();
d.join();
for(int i=1;i<=5;i++){
System.out.println(Thread.currentThread().getName()+"...>"+i);
Thread.sleep(1000);
}
}
}
```

- **sleep () method:**
  - If a thread doesn't want to perform any operation for particular amount of time, then we should go for sleep ().
  - Public static final native void sleep (long millisec) throws InterruptedException.
  - Public static final native void sleep (long millisec, int nanosec) throws InterruptedException.
  - Every sleep () throws IE(InterruptedException), which is checked exception hence when ever using sleep () compulsory we should handle IE by using try catch or throws else will get CTE.

```
public class Demo55{
public static void main(String []args) throws InterruptedException{
for(int i=1;i<=5;i++){
System.out.println("hello");
Thread.sleep(2000);
}
}
}
```

```
C:\Users\ganieswar\Desktop\java_programs>java Demo55.java
hello
hello
hello
hello
hello
```

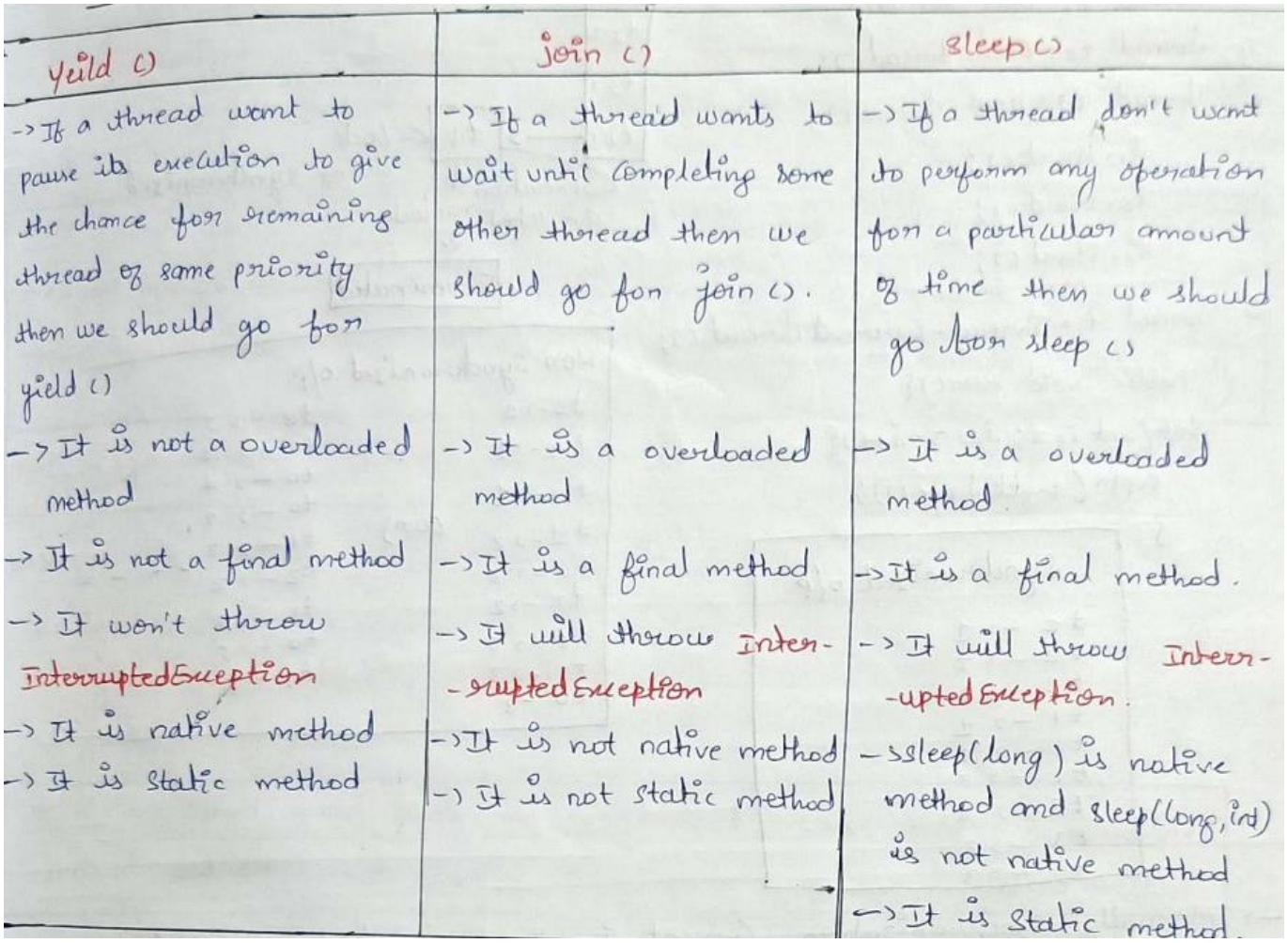
- **interrupt () method:**
- A thread can interrupt sleeping thread/waiting thread by using interrupt () of thread class.
- Syntax: public void interrupt ().
- **Interrupting the sleeping thread program:**

```
public class Demo54 extends Thread{
public void run(){
try{
for(int i=1;i<=5;i++){
System.out.println("i will sleep, don't disturb");
Thread.sleep(5000);
}
}catch(InterruptedException e){
System.out.println("who disturbed me");
}
}
}
```

-

```
public class Demo55{
public static void main(String []args){
Thread t = new Thread();
t.start();
t.interrupt();
System.out.println("end of the main Thread..");
}
}
//end of the main Thread..
```

- Whenever we are calling interrupt () and the target thread is not in sleeping state or waiting state then there is no impact of interrupt call immediately, interrupt call will be waited until target thread is entered into sleeping or waiting state.
- If the target thread entered into sleeping or waiting state then immediately interrupt call will interrupt target thread.
- If the target thread never enters into sleeping or waiting state in its life time, then there is no impact of interrupt call, this is the only case where interrupt call will be wasted.
- The difference between yield (), join (), sleep ():



yield()	join()	sleep()
-> If a thread want to pause its execution to give the chance for remaining thread of same priority then we should go for yield()	-> If a thread wants to wait until completing some other thread then we should go for join().	-> If a thread don't want to perform any operation for a particular amount of time then we should go for sleep()
-> It is not a overloaded method	-> It is a overloaded method	-> It is a overloaded method.
-> It is not a final method	-> It is a final method	-> It is a final method.
-> It won't throw InterruptedException	-> It will throw Inter-rupted Exception	-> It will throw Interr-upted Exception.
-> It is native method	-> It is not native method	-> sleep(long) is native method and sleep(long, int) is not native method
-> It is static method	-> It is not static method	-> It is static method.