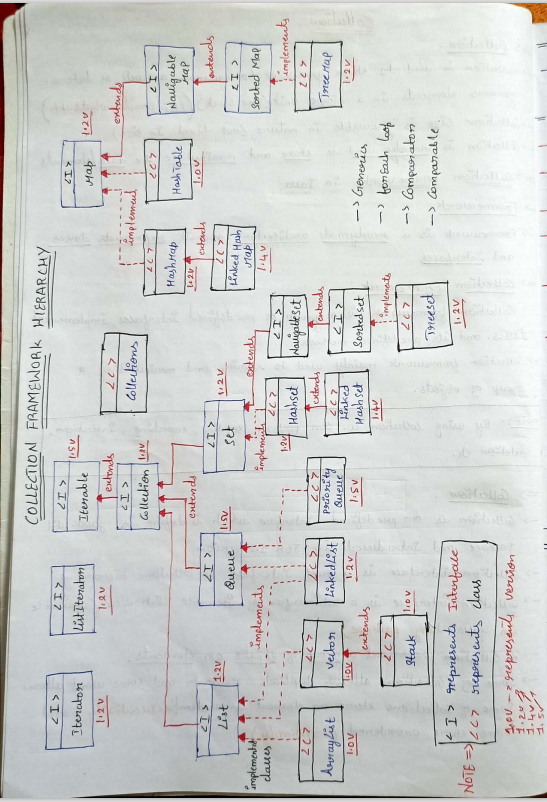
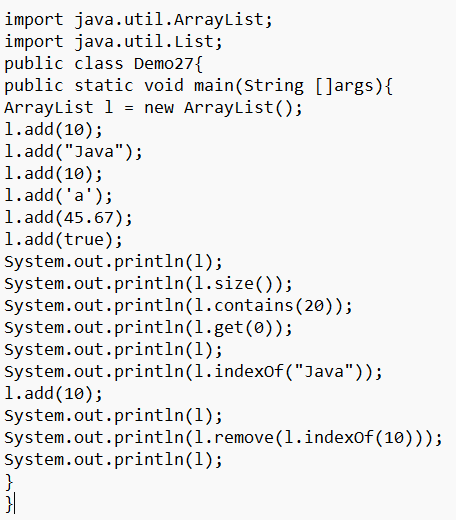
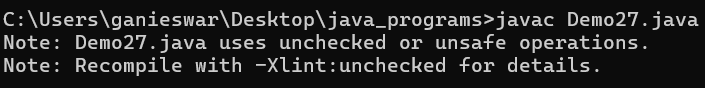
Collections

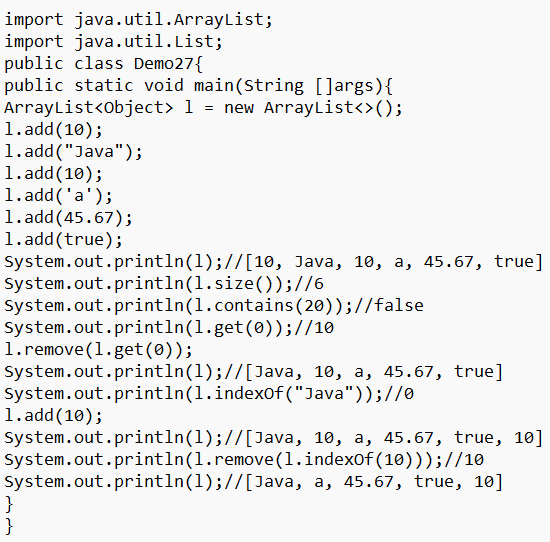
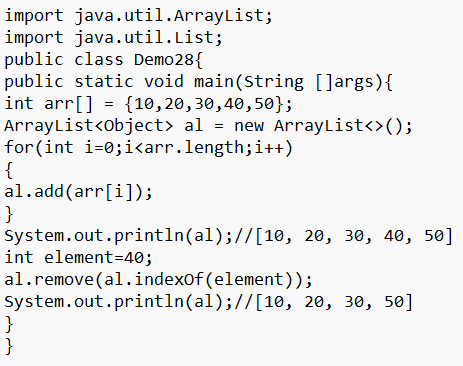
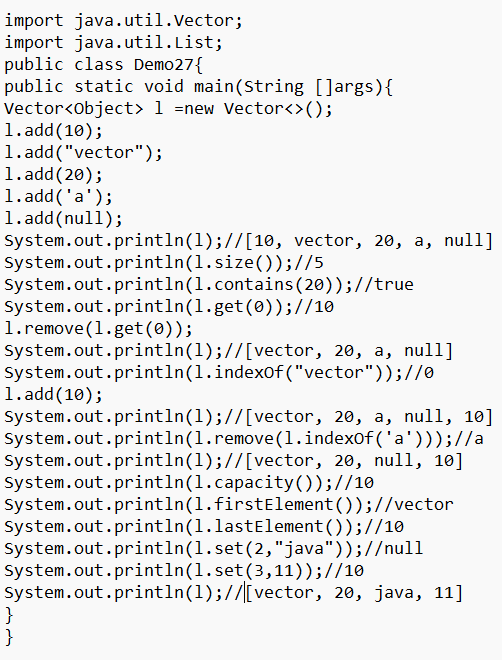
* Collection is used to store group of homogeneous as well as heterogeneous elements in a single unit (one block)
* Collection size is growable in nature. (not fixed in size)
* Collection is mainly used to store and manipulate the data/ elements.
* COLLECTION FRAMEWORK:
* Framework is a readymade architecture 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 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.**
* 
* 
* 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>:

* 
* **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.
* 
* 
* 
* 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)
* 
* 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)**

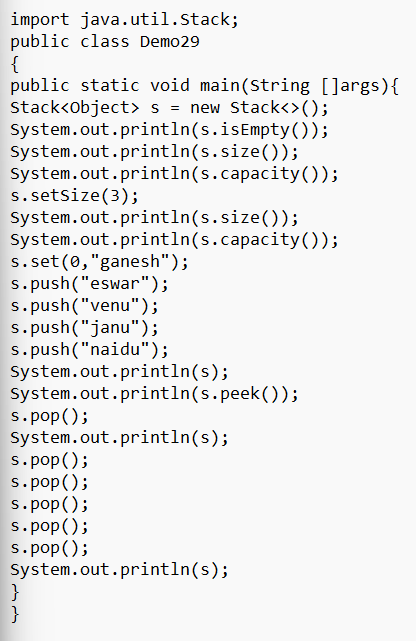
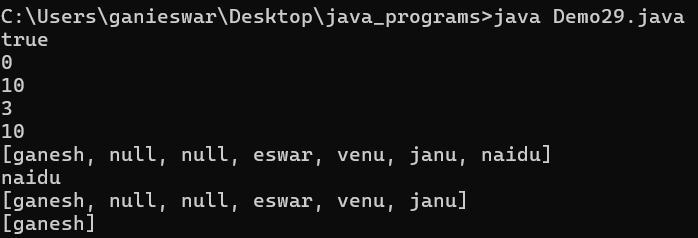
It is used to preserve the stack memory based on given index.

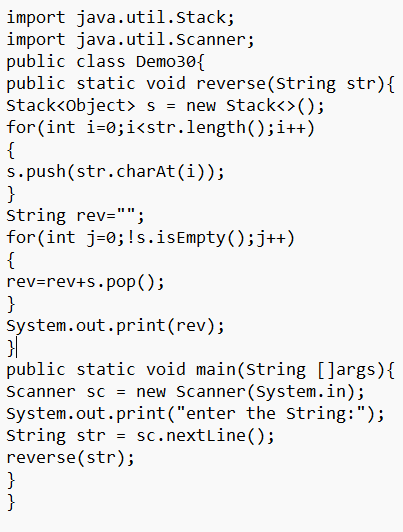
* **set (int index, Object e): (Object, vector)**

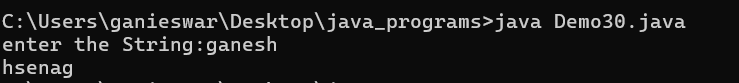
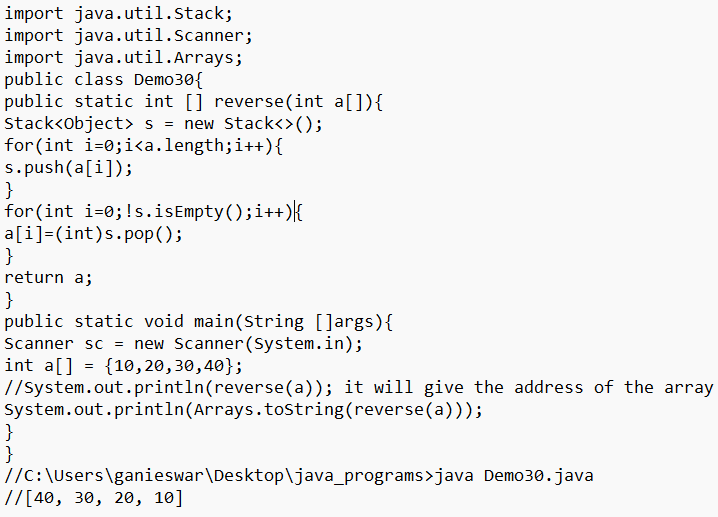
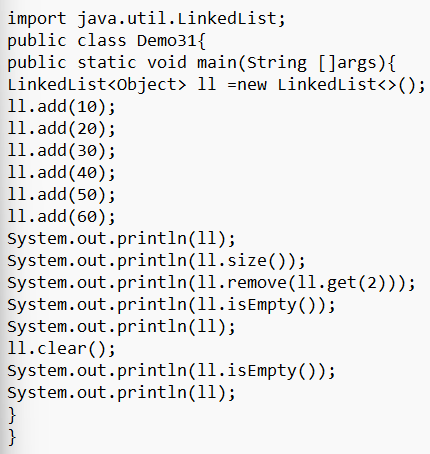
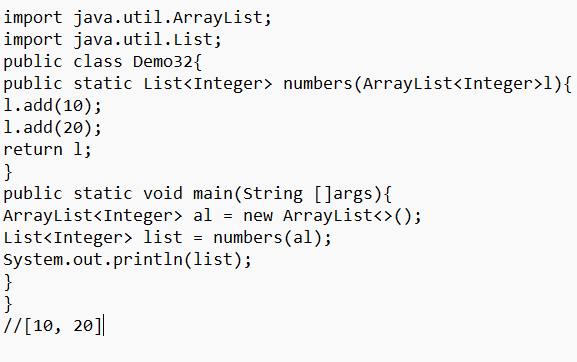
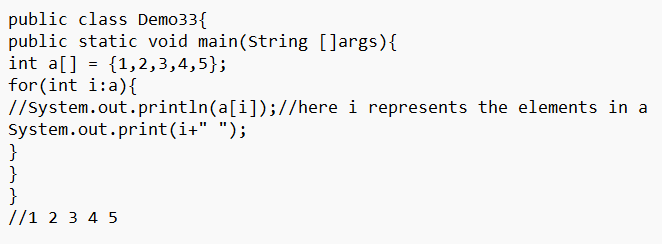
It is used to set/ update an element in stack based on given index.

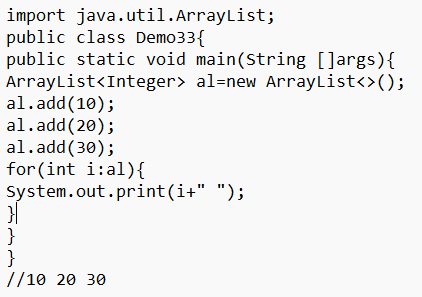
* **setElementAt (Object e, int index): (void, vector)**

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.
* 
* 
* 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:**

****

* ****
* 
* **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.
* 
* 
* 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<>()
* 
* **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.
* 
* For ArrayList:



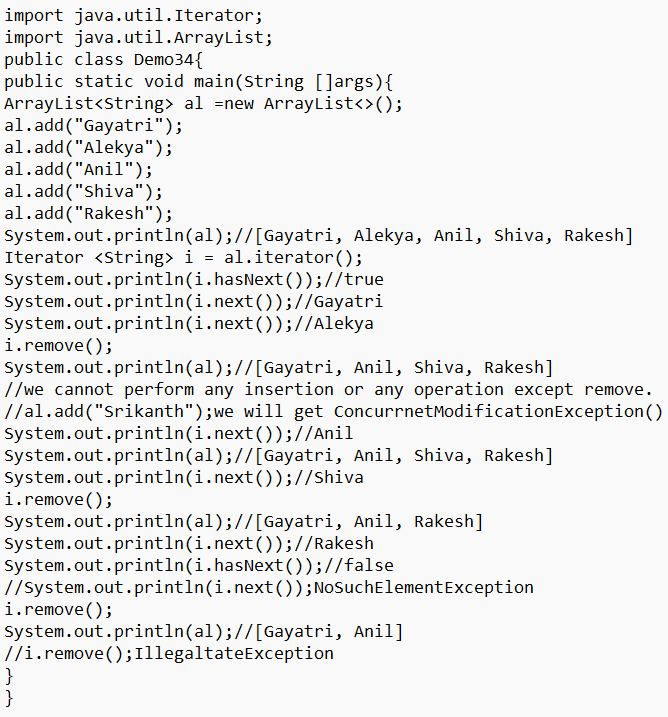
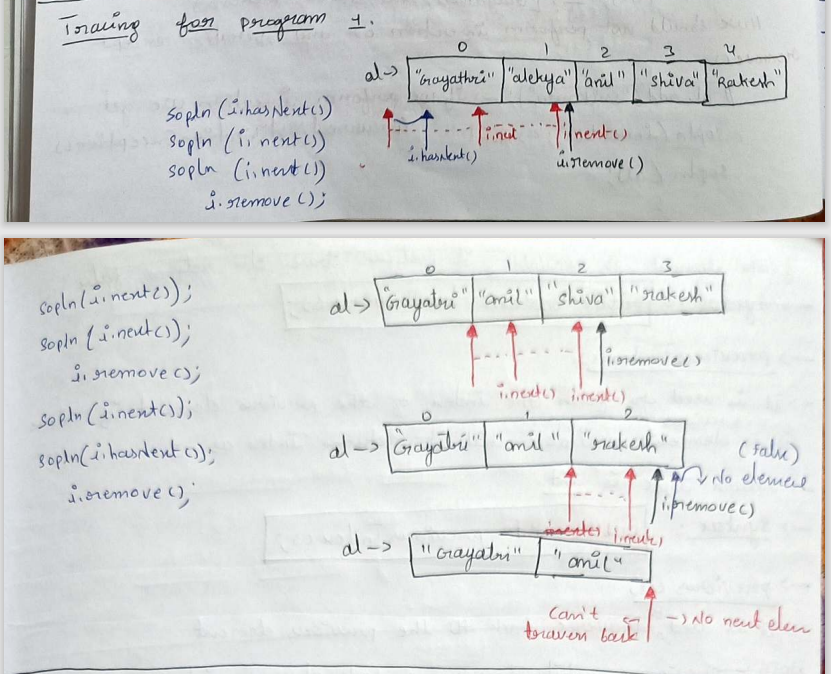
* Here we can’t use object in generics because we use int i in the for loop, because we are storing elements of ArrayList to i which need to be the same type.
* CURSORS IN JAVA: (3)
* **1.Iterator:**
* Iterator interface is a cursor which is defined in java.util.
* Iterator interface is used to traverse element by element. It can only traverse in forward direction. It has 3 abstract methods:
* Public Object next ()

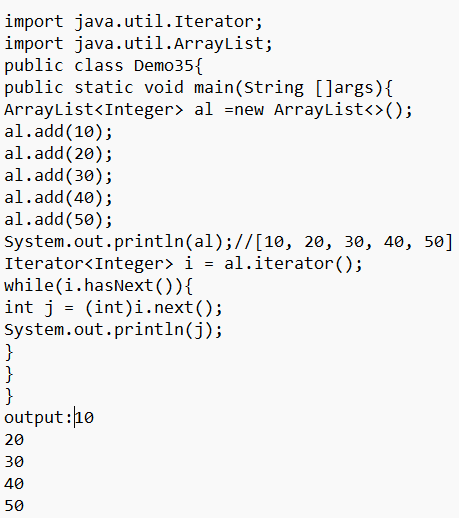
Public boolean hasNext ()

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’.
* 
* 
* Using While Loop



* 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 ()

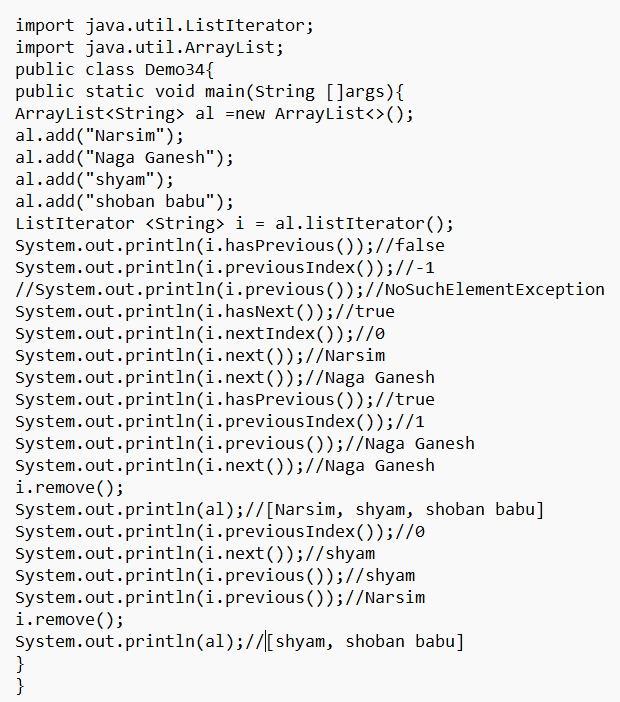
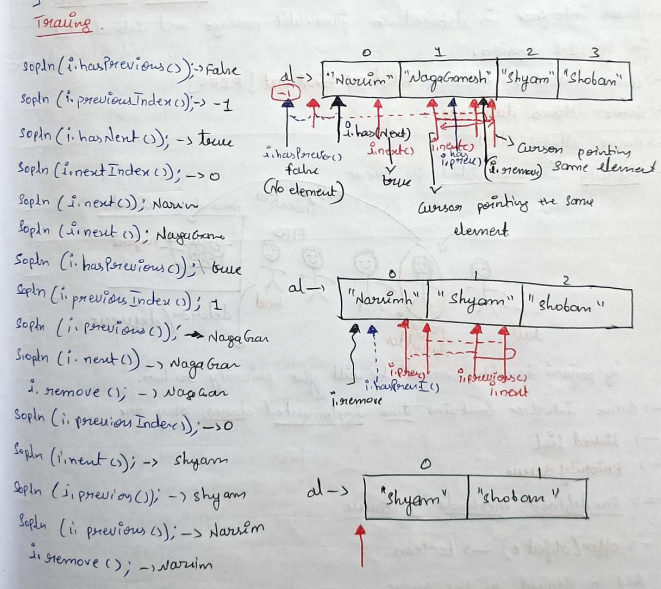
nextIndex (): fetch the next element index, return type is int.

remove ()

previous (): used to traverse back to the previous element.

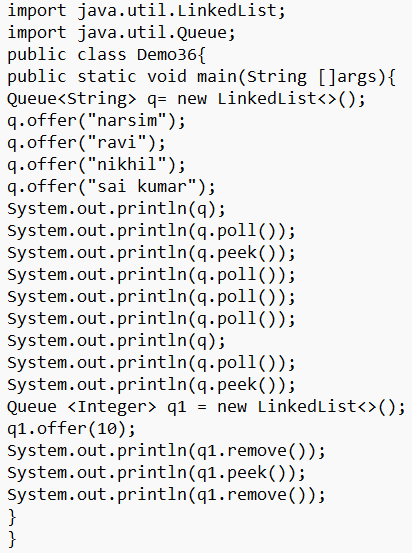
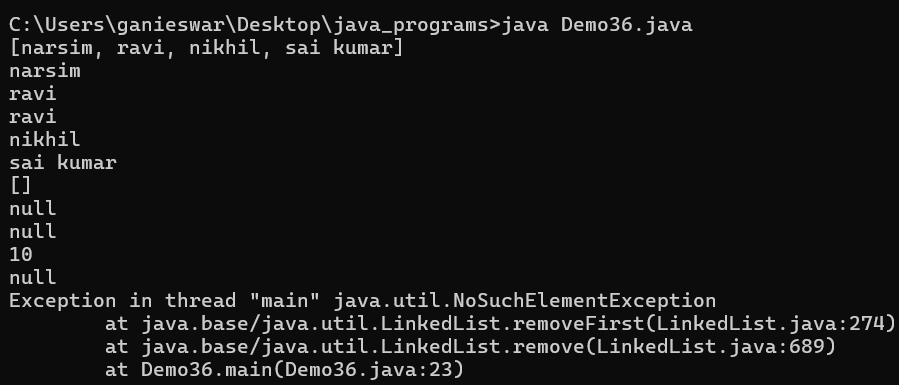
hasPrevious (): Check availability of previous element, return type is Boolean

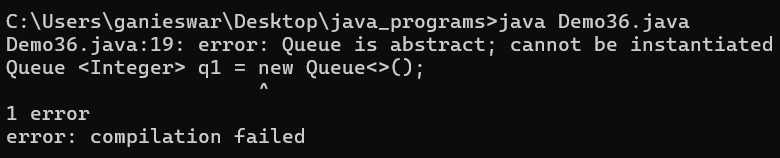
previousIndex (): used to fetch the index of the previous element if there is no previous element available still it returns index as -1.

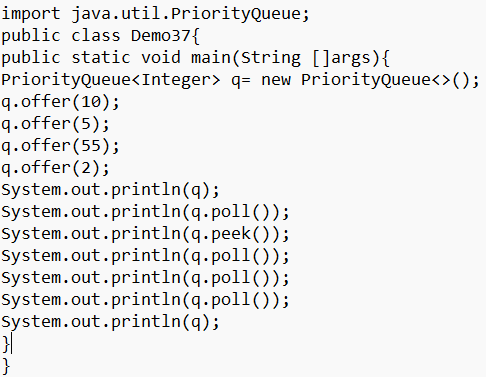
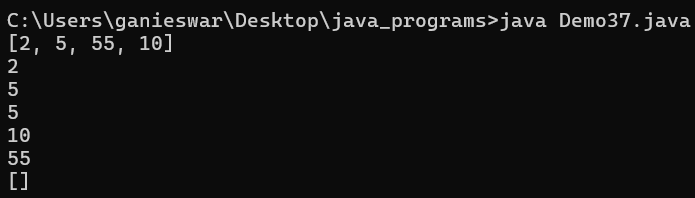
* NOTE: previous () while traversing back, always point the currently pointing element as the previous element.
* 
* List iterators used only in list Interface.
* 
* 3.Enumeration
* Enumeration is a cursor/ iteration which is defined in java.util.
* Iterator interface is a replica for Enumeration.
* 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.
* 
* 
* We cannot use like this:

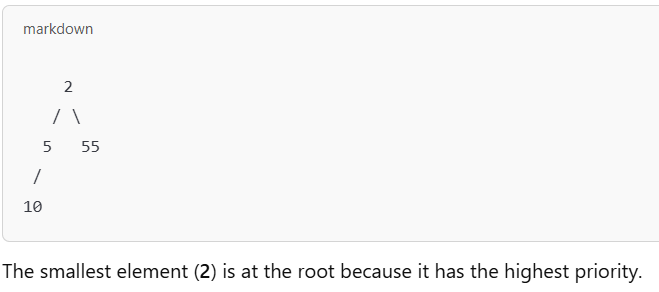
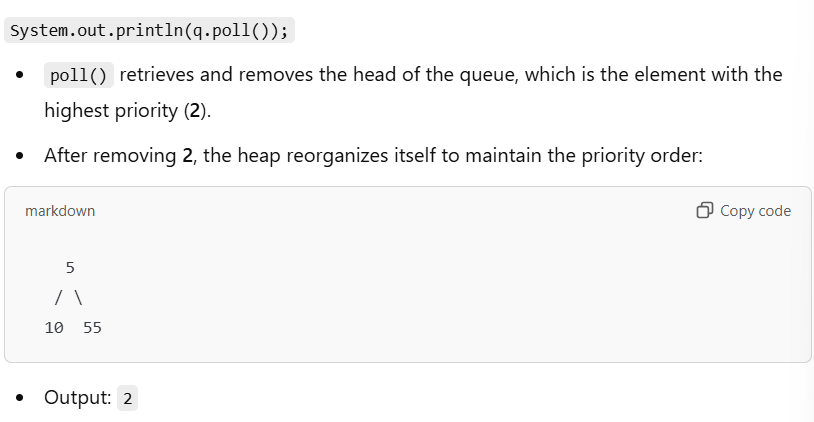
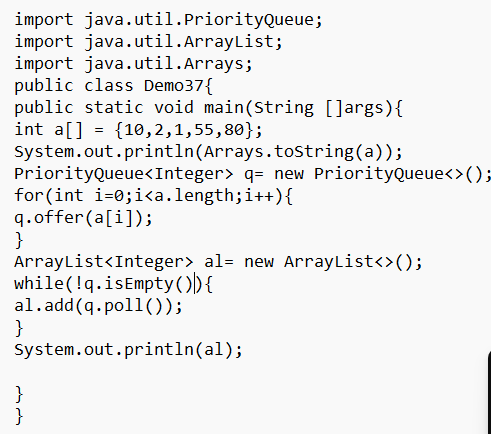
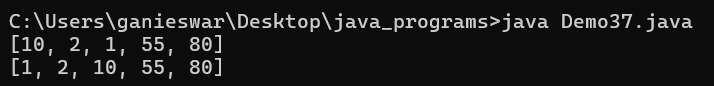
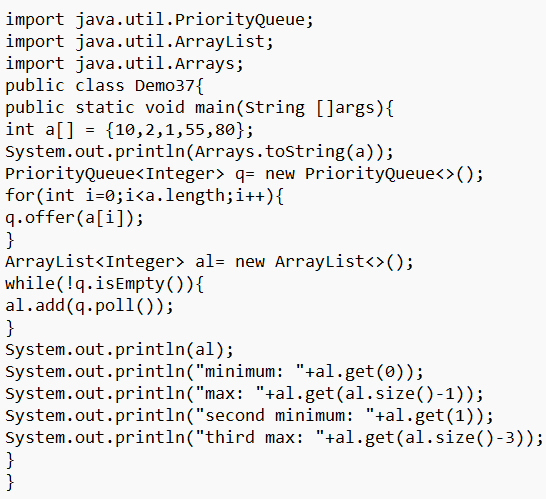
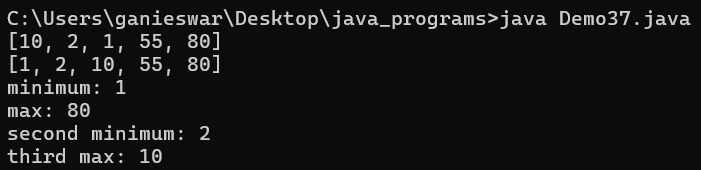
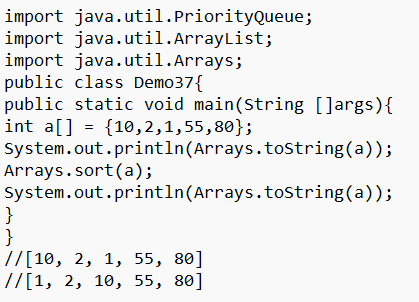


* **PriorityQueue:**
* PriorityQueue is an implemented class which implements from Queue interface
* It is defined in java.util package and it follows unordered queue (It stores elements randomly)
* It allows duplicates and null values.
* The default ordering is **natural ordering** (ascending order,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

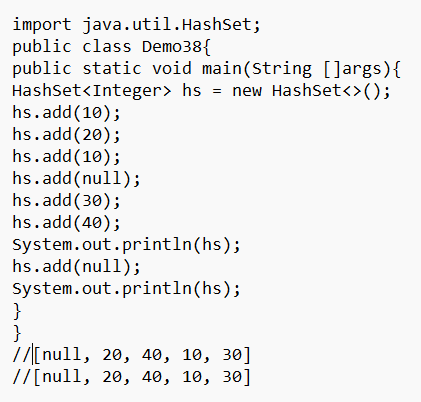
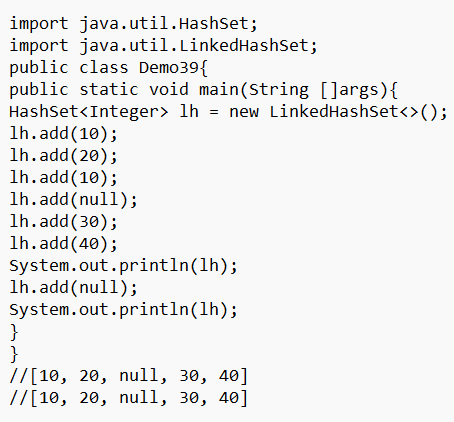
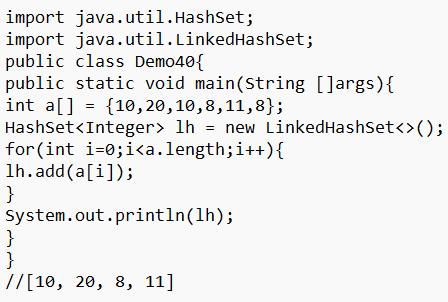
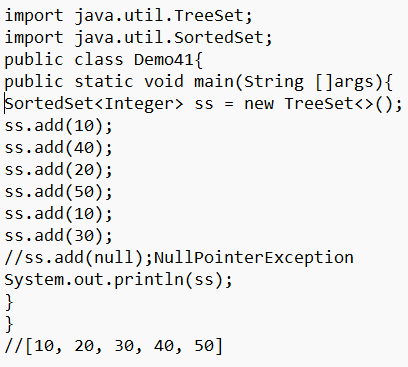
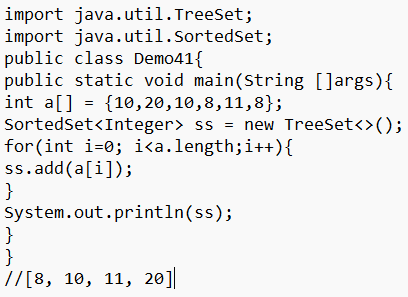
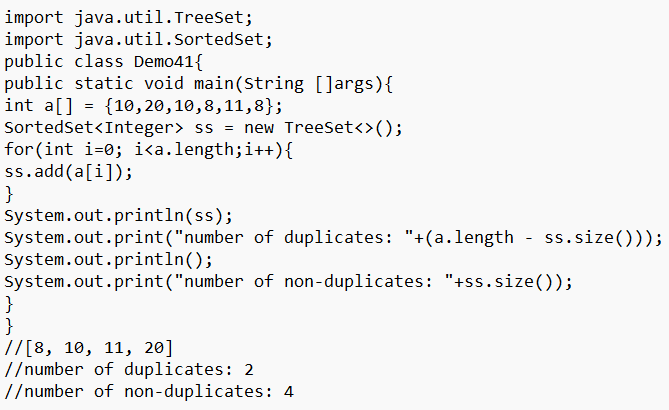
The **PriorityQueue** uses a **binary heap** internally to maintain the elements. After adding all elements, the structure will be:

* 
* 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.
* 
* 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)**
* 
* 
* **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.
* 
* 
* 
* 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

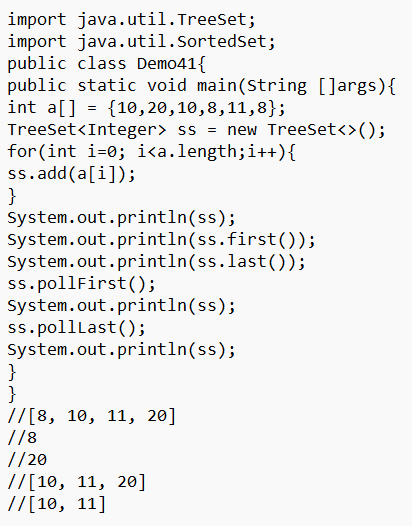
* **HashSet:** HashSet is an implemented class which implements from set interface.
* It is defined in java.util package and it won’t store elements based on index.
* It allows only one null element. It won’t preserve the element order (it stores randomly)
* 
* It stores randomly.
* **Linked HashSet:**
* 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.
* 
* **How to remove the duplicates from the given array?**
* ****
* **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.
* 
* **Write a program to remove the duplicates and sort in ascending order from the given array:**
* ****
* **Number of duplicates and non-duplicates in array.**
* ****
* 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.**

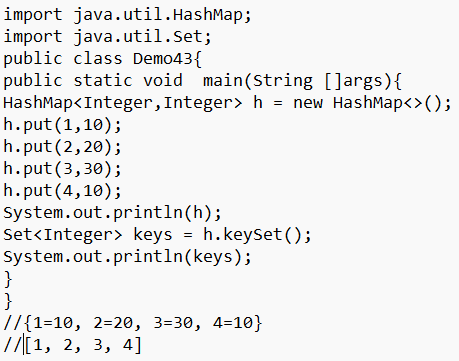
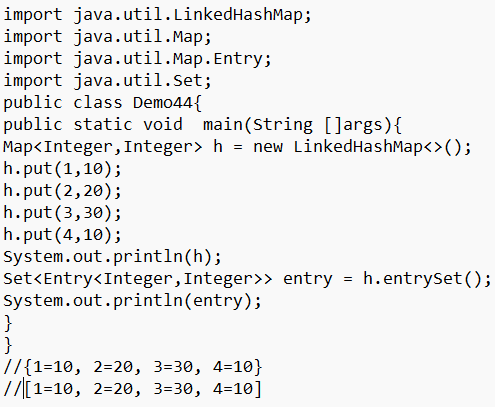
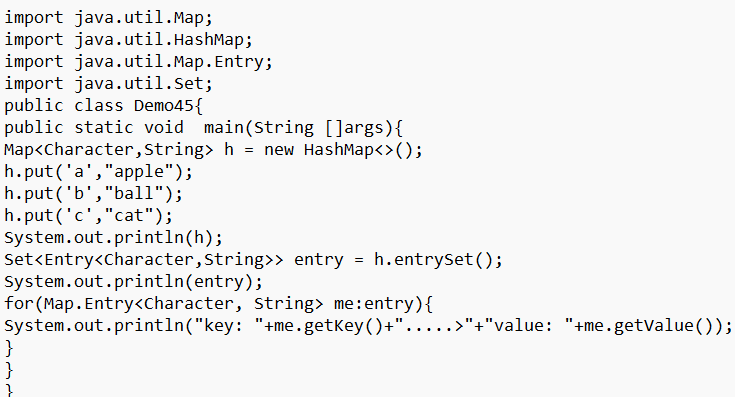
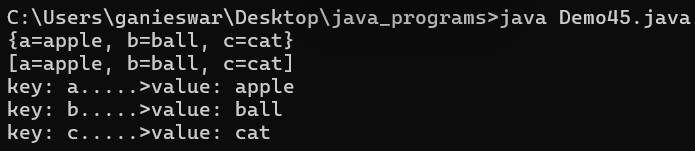
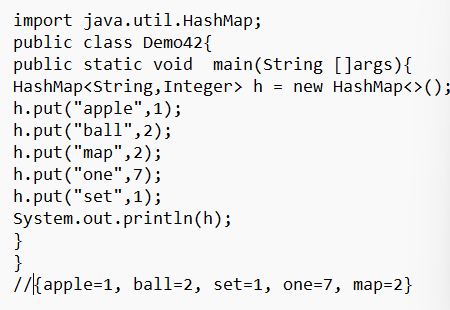
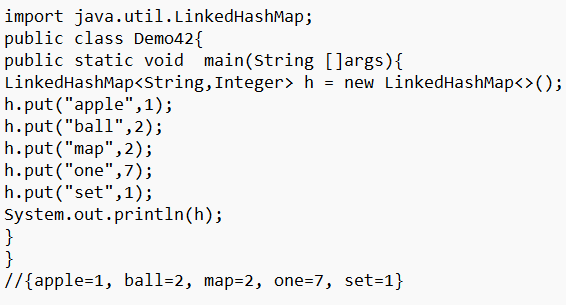
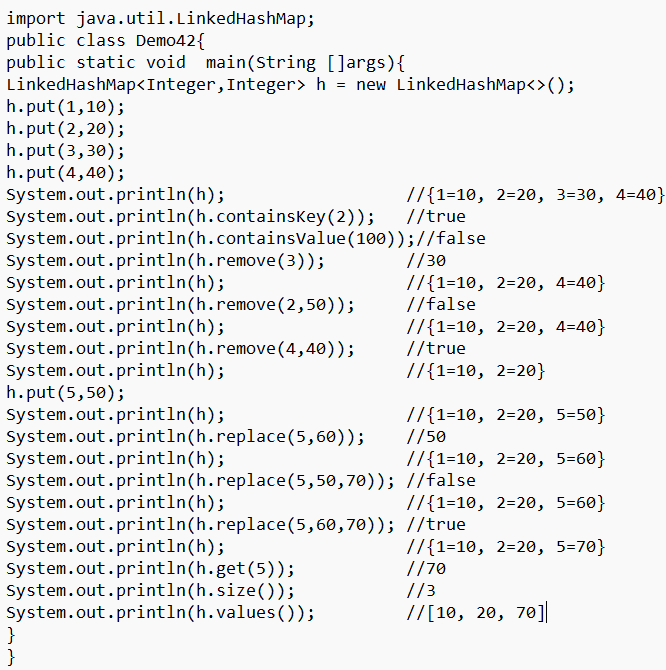
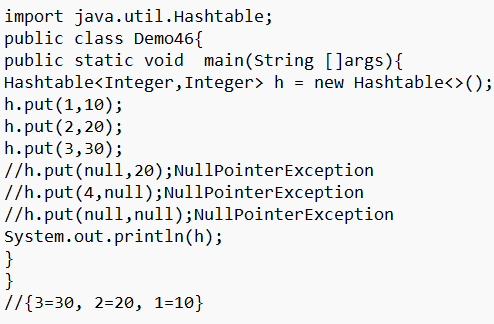
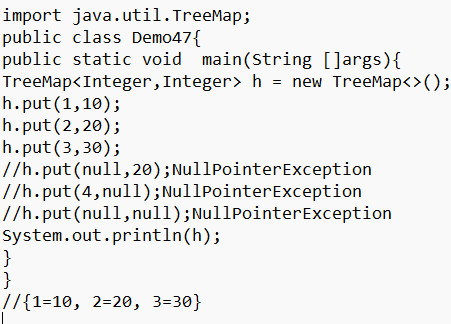
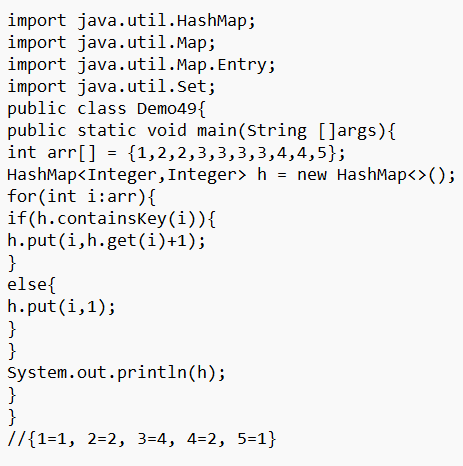
* 
* MAP
* Map is an interface which is defined in java.util package.
* Map interface have no parent interface.
* It allows to store an element based on key and value pair.
* There is no index concept.
* One key and value pair in known as entry.
* Keys won’t allow duplicates and values can be duplicate.
* Map interface which contains 4 implemented classes they are:

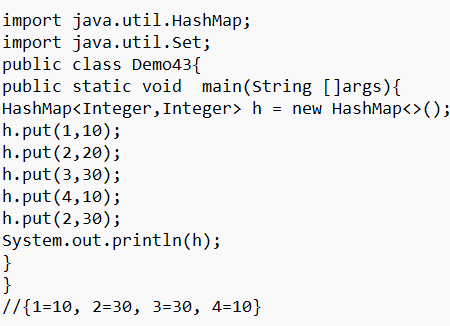
HashMap

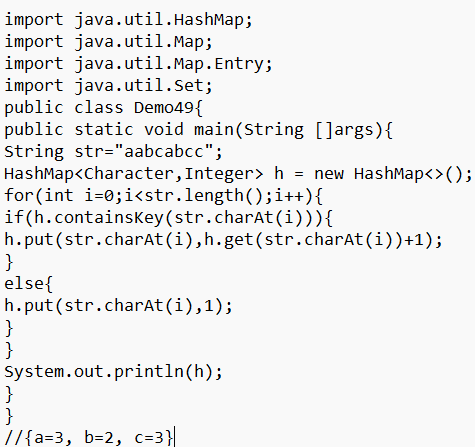
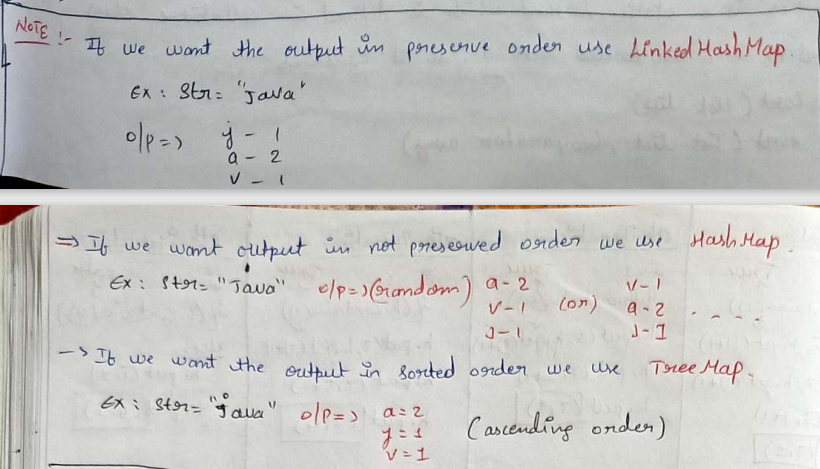
Linked HashMap

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>
* 
* **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>
* 
* **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.
* 
* 
* **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.
* 
* 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.
* ****
* ****
* **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.
* ****
* **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.
* ****
* **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.
* ****
* 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:



* Write a program to find the frequency of each character from the given string.
* 
* Or
* 
* 
* 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)

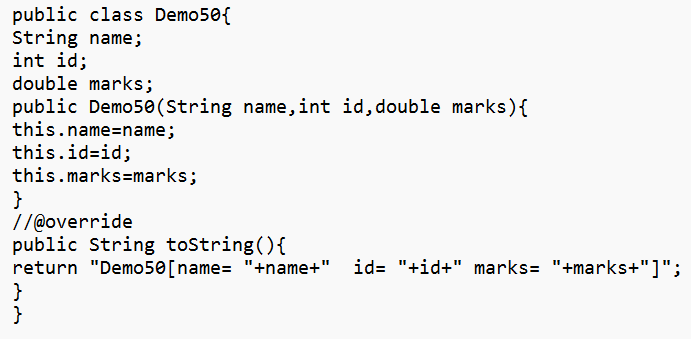
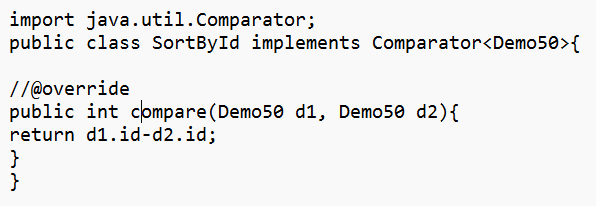
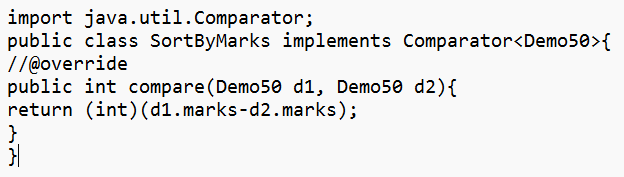
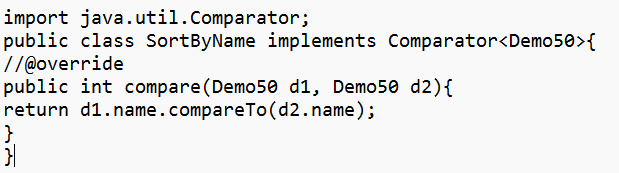
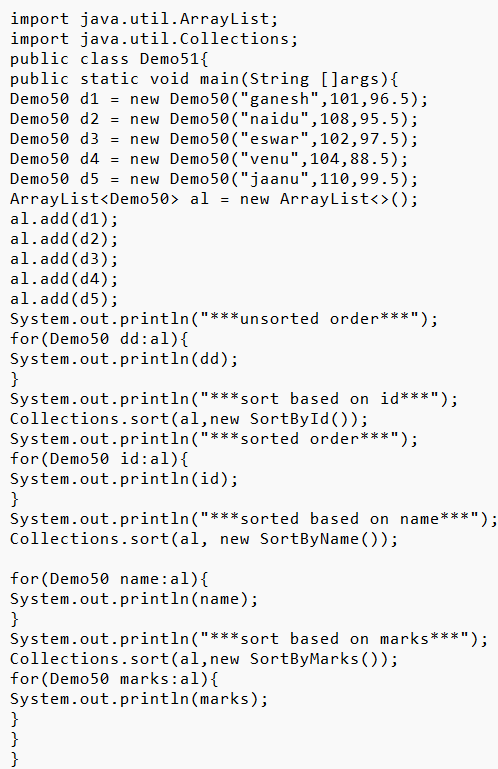
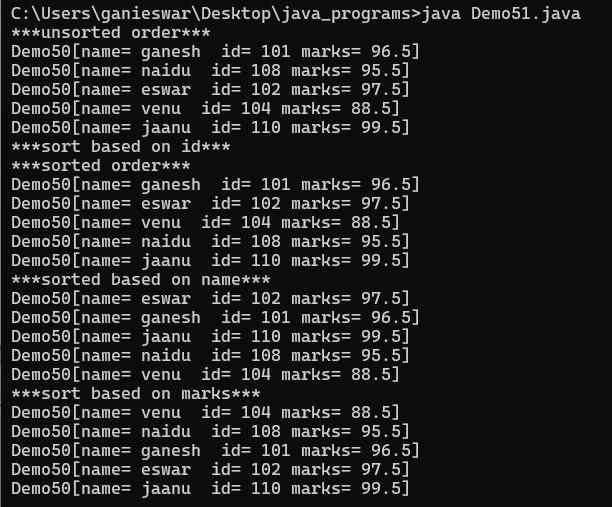
* **compare (Object obj1, Object obj2):**

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**:
* 
* 
* 
* 
* 
* 
* 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

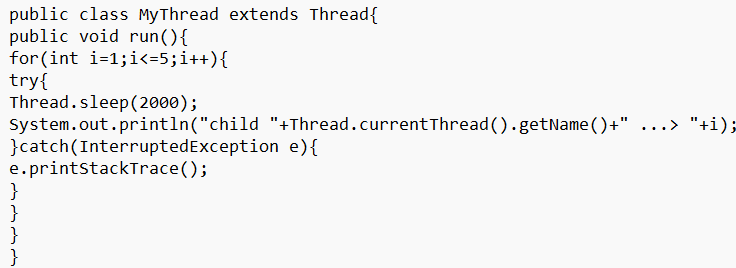
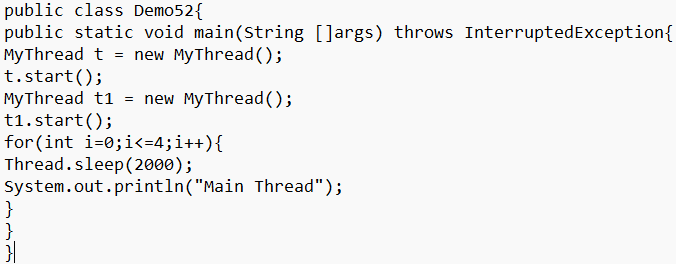
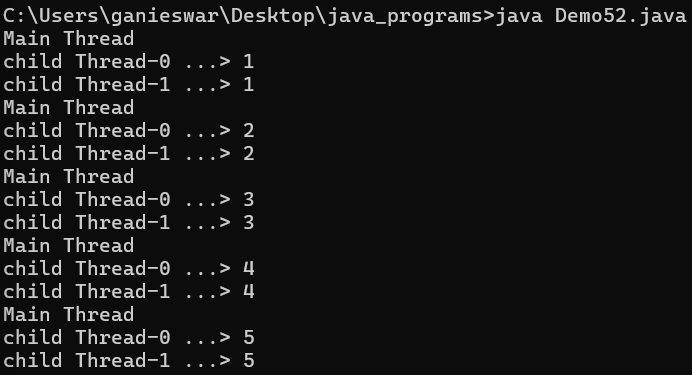
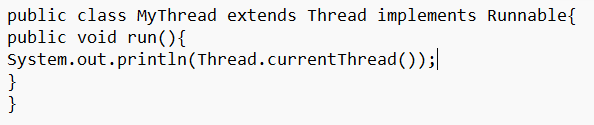
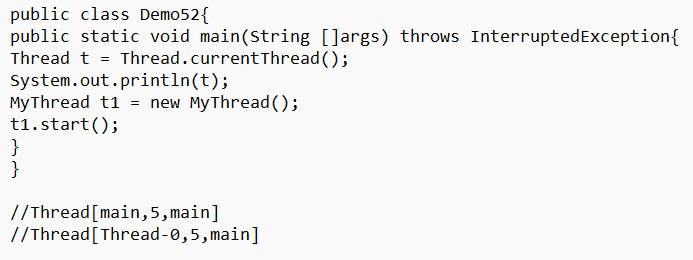
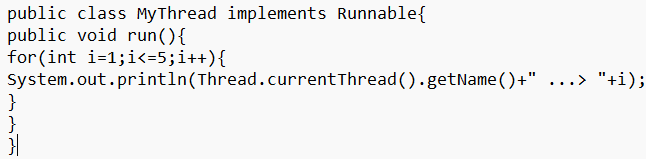
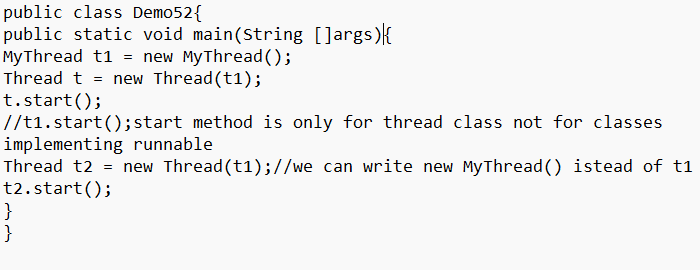
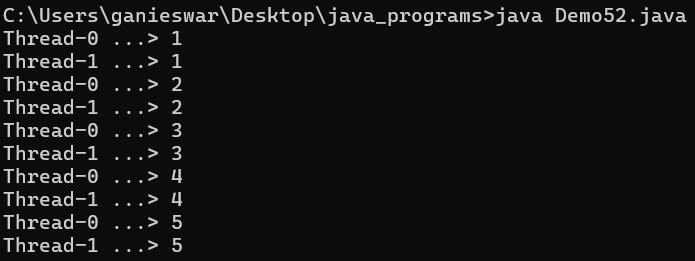
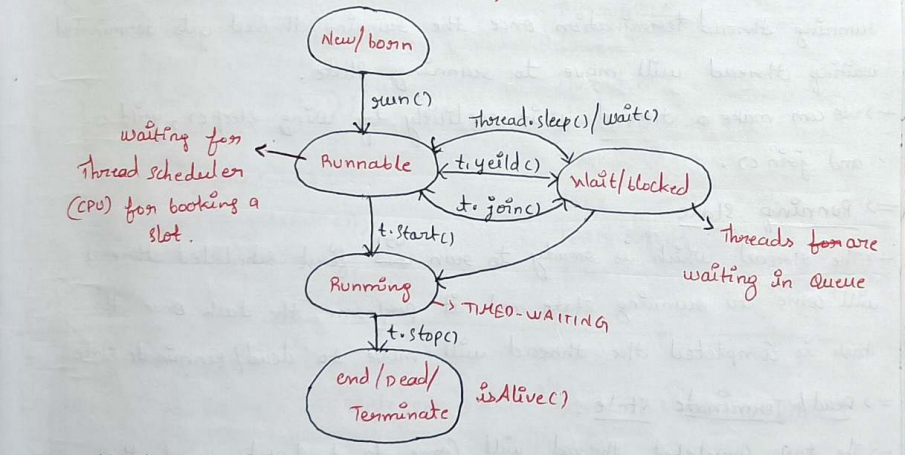
To develop video games

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:** Threadisa 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 ().**
* 
* 
* 
* 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 ().
* 
* 
* **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 ().
* 
* 
* 
* 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

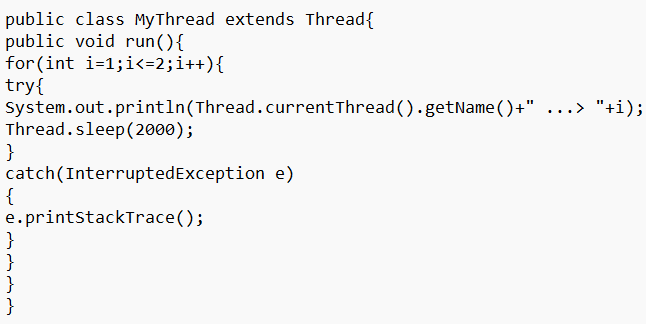
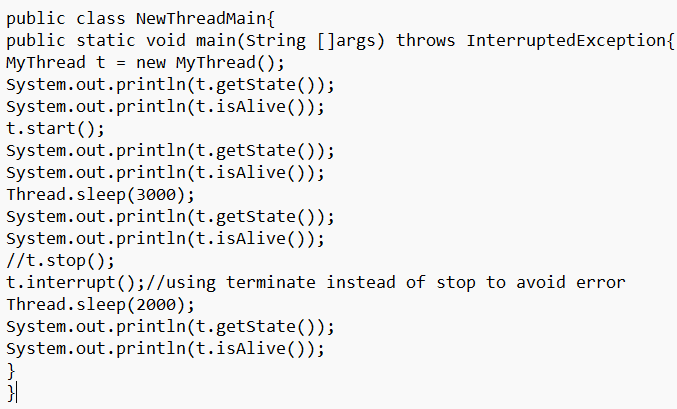
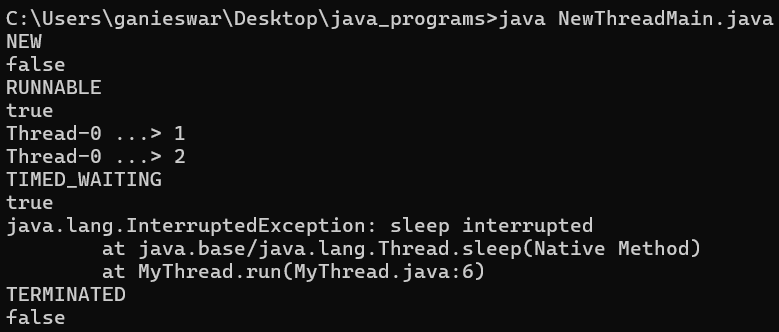
* **New/born state:** when a thread gets created (Object Creation) new thread will be born, which is ready to run a thread will move to runnable state.
* **Runnable state:** The thread is ready to run but nor running, the threads are waiting for the Thread Scheduler (CPU) to book the slot.
* If there 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:**
* 
* 
* 
* 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.

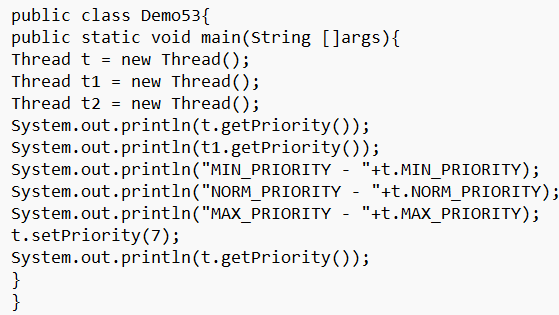
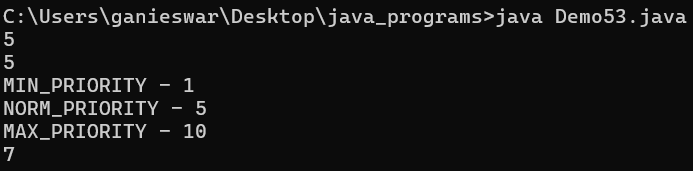
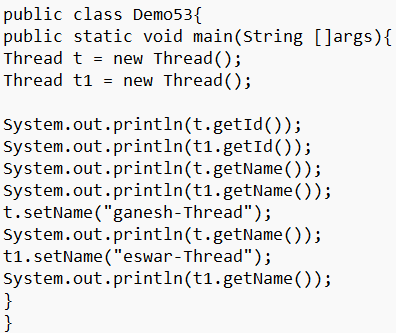
* **Thread Priority:** Thread priority is used to fetch the priority and modify the priority of the thread.

There are three types of priority. They are

MIN\_PRIORITY =>1

NORM\_PRIORITY=>5

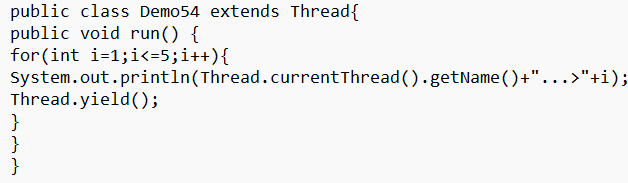
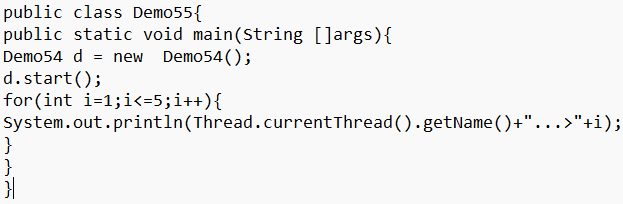
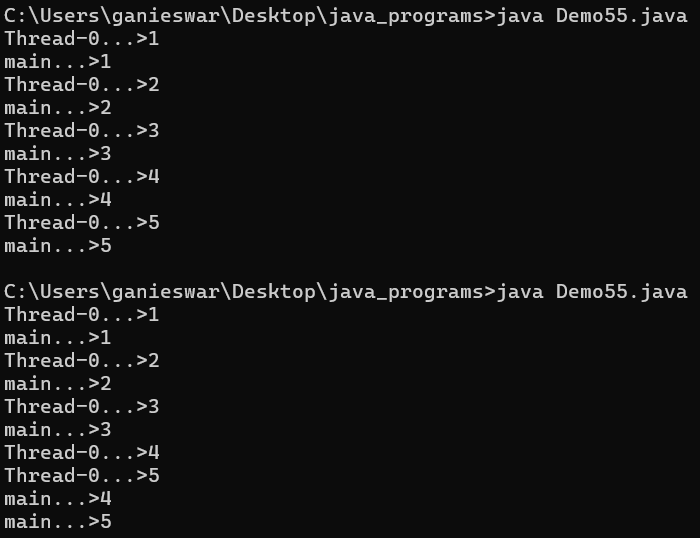
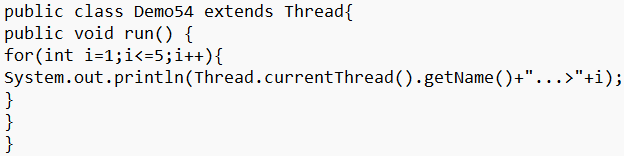
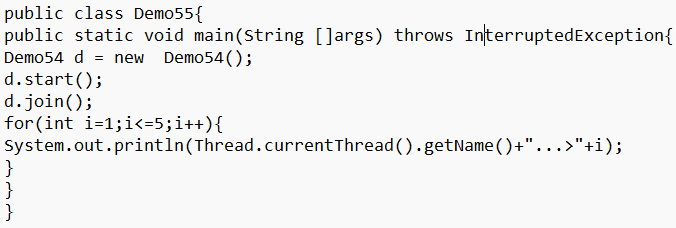
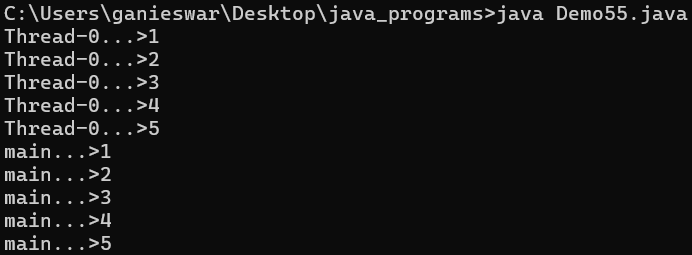
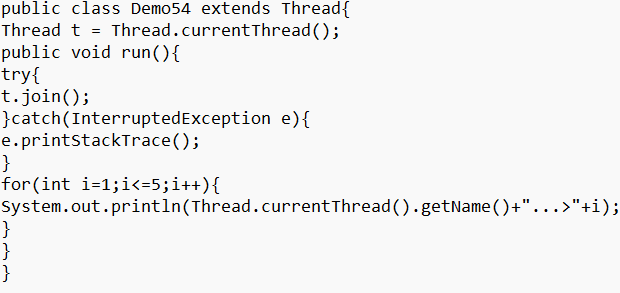
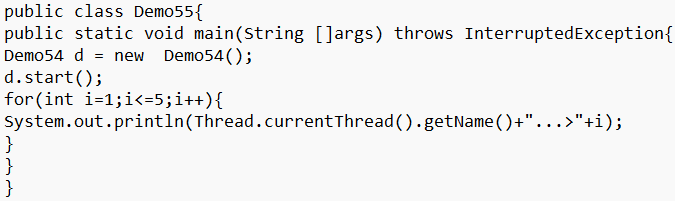
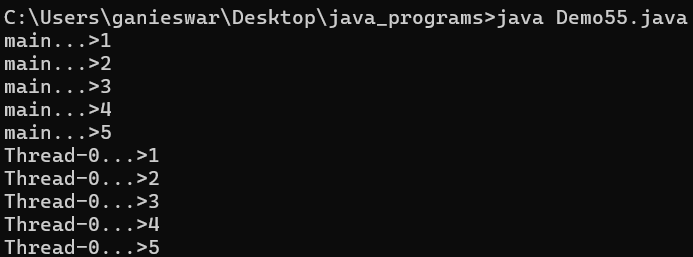
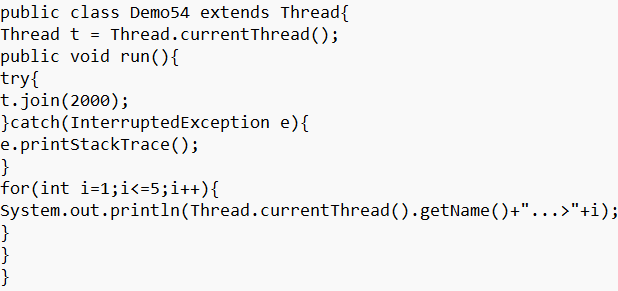
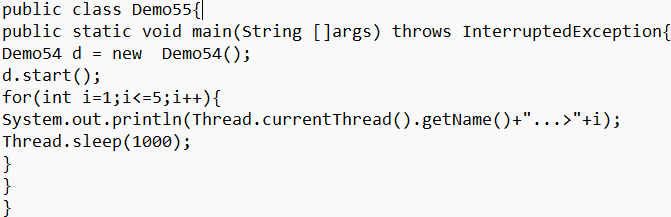
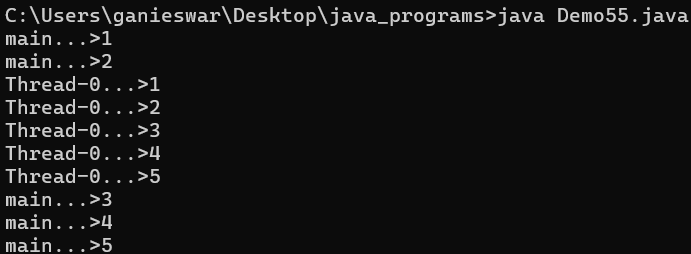
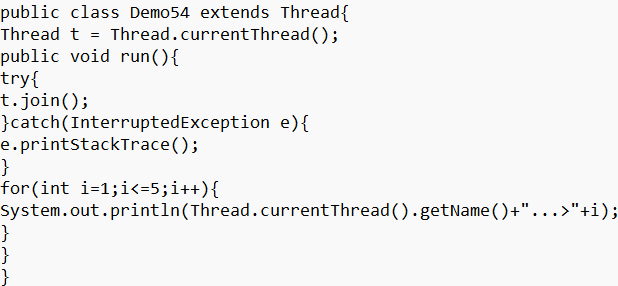
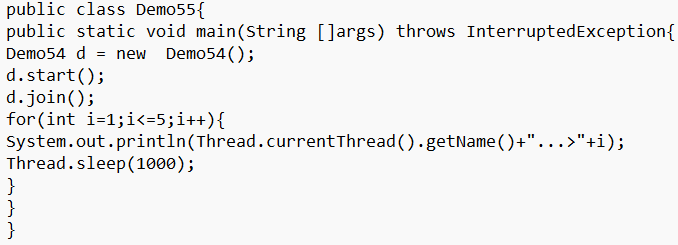
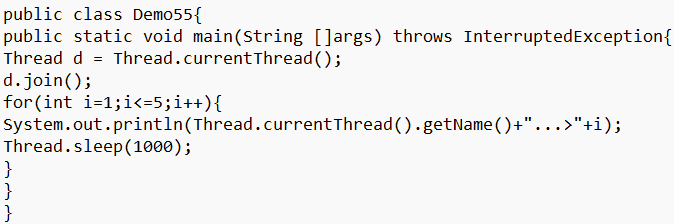
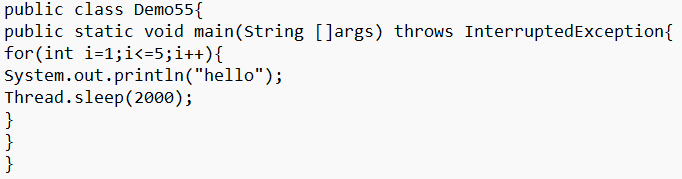
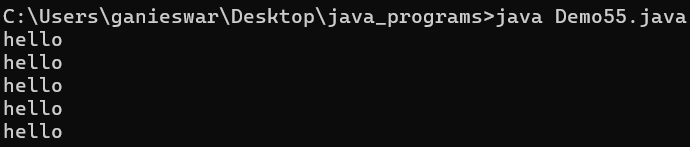
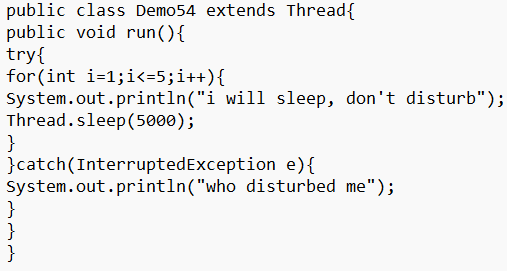
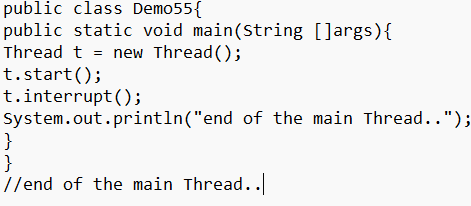
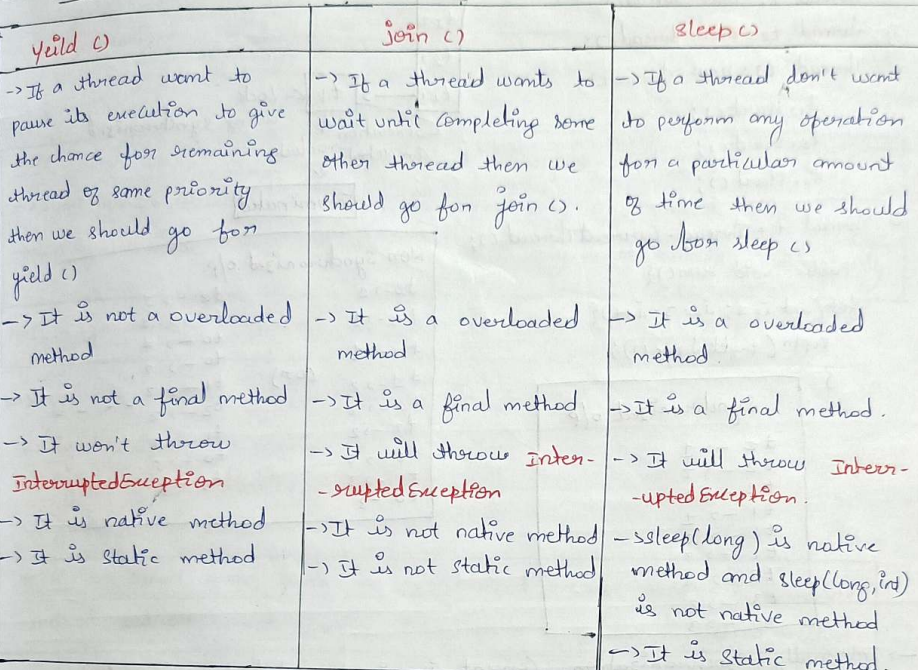
MAX\_PRIORITY=>10

* 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.
* 
* 
* **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)
* 
* 
* 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.
* 
* 
* 
* 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.**
* 
* 
* 
* **Case 2: child thread has to wait until main thread get executed.**
* 
* 
* 
* **Case 3: join thread is waiting for only specified time, it is not waiting completely until another thread get executed.**
* 
* 
* 
* 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.**
* 
* 
* **Case 2: A thread waiting itself for the execution forever.**
* ****
* **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.
* 
* 
* **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:**
* 
* 
* 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 ():
* 
* 119