Assignment 8.

① Difference b/w Arraylist & Array.

→ ① Array: Simple fixed sized arrays that we create in Java, like. (Ex) int arr[] = new int[10];

② Arraylist: Dynamic sized arrays in Java that implement "list interface".

(Ex) Arraylist < Type > arr = new Arraylist < Type > ();

Here Type is the type of elements in Arraylist to be created.

Differences: ① An array is basic functionality provided by Java & they are accessed using [].

(*) Arraylist is part of collection framework in Java. & has a set of methods to access elements & modify them.

(Ex) import java.util.Arraylist;
import java.util.Arrays;

class Test {
PSVM(....) {
int [] arr = new int[2];
arr[0] = 1; // normal arrays.
arr[1] = 2;
S.O.P (arr[0]);

Arraylist < Integer > arrL = new Arraylist < Integer > (2);
arrL.add(1); // add elements
arrL.add(2);

```
S.O.P (arrL.get(0)); // access elements
}
```

O/p: 1 1

(2) Array is a fixed data structure while Arraylist is not. One need not to mention the size of Arraylist while creating its object. Even if we specify some initial capacity. we can add more elements.

(Ex)
```
class Test{
    PSVM(...){
    int arr[] = new int[2];
      arr[0]=1;
      arr[1]=2;
      S.O.P(arr[0]);
    Arraylist<Integer> arrL = new Arraylist<Integer>();
      arrL.add(1);
      arrL.add(2);
      arrL.add(3);
    S.O.P (arrL);
    S.O.P (Arrays.toString(arr)); }
}
```

O/p: [1,2,3]
[1,2]

(3) Array can contain both primitive data types as well as objects of a class depending on the definition of the array.

(4) Arraylist only supports object entries, not the Primitive data types.

Note: When we do arraylist.add(i); it converts the primitive int type (data type) into Integer object

(Ex) import java.util.Arraylist;

```
class Test {
psvm() {
int[] array = new int[3];      // Allowed.
Test[] array1 = new Test[3];   // Allowed.
```

Arraylist<char> arL = new Arraylist<char>(); // not allowed

Arraylist<Integer> arL = new Arraylist<Integer>();
Arraylist<String> arL = new Arraylist<String>();
Arraylist<Object> arL = new Arraylist<Object>();
} // allowed.

(4) Since Arraylist can't be created for primitive data type, members of Arraylist are always references to objects at diff memory locations therefore in Arraylist, the actual objects are never stored at contiguous location. "references" of the actual objects are stored at contiguous location.

(W) In array, it depends whether the arrays is of primitive type or object type. In case of primitive types, actual values are continguous loc, but in case of objects, alloc is similar to Arraylist.

(5) Java Arraylist supports many operation like indexOf(), remove() etc. these func are not supported by Arrays.

② why use generic ?

→ Generic was introduced to deal with type-safe objects It makes the code stable by detecting the bugs at compile time.

Advantages ① Type - Safety : we can hold only a single type of objects in generics. It doesn't allow to store other objects.

→ without generic we can store any type of objects.

Ex: List list = new ArrayList ();
list.add(10);
list.add ("10");

→ With generics, it is required to Specify the type of object we need to store

Ex List<Integer> list = new ArrayList<Integer> ();
list.add(10);
list.add("10"); // compile time error.

② Type casting is not required : there is no need to type-cast the object.

Ex. Before generic, we need to type cast.

List list = new ArrayList ();
list.add ("Hello");
String s = (String) list.get(0); // typecasting.

After generic :- Ex List<String> list = new ArrayList<String>();
list.add ("hello");
String s = list.get(0);

Compile-time checking

→ It is checked at compile-time so problem will not occur at runtime. The good programming strategy says It is far better to handle the problem at compile time than runtime.

(Ex) List<String> list = new ArrayList<>();
list.add("Hello");
list.add(123); // Compile time error.

Syntax to use generic

ClassOrInterface<Type> (Ex) ArrayList<Integer>.

→ before generic we can store any type of objects in the collection i.e non-generic. Now generic force the java programmer to store a specific type of objects

(3) Explore the following in Java JDK & try basic codes

(1) HashMap

(*) HashMap<K,V> is a part of java collection, (java.util) Package, which provides basic implementation of Map interface in Java.

(*) It store the data in (key, value) pair, & we can access them by an index of another type (eg: Integer). One is used as a key (index) to another object (value)

(*) If we try to insert the duplicate key, it will replace the element of the corresponding key.

Eg: import java.util.HashMap;

class Test {

psvm(---) {

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

map.put("Vishal", 10);

map.put("Archu", 05);

S.O.P("Size of map is:" + map.size());

S.O.P(map);

if (map.containsKey("Vishal")) {

Integer a = map.get("Vishal");

S.O.P("value for key" + "\"Vishal\" is:" + a);

}}

O/P: Size of map is: 2

{ Vishal = 10, Archana = 20 }

Value for key "Vishal" is: 10

Application: HashMap is mainly the implementation of hashing. It is useful when we need efficient implementation of search, insert & delete operation.

* HashMap is unsynchronized. to make it synchronized. use Collections.synchronizedMap()

Ex: Map m = Collections.synchronizedMap(new HashMap(...));

* Internally HashMap contains an array of node and a node is represented as a class that contain 4 field.

① int hash ② K key ③ V value ④ Node next

Important features

(11) To access a value one must know its key. HashMap is called so bec/ it uses technique called Hashing.

Hashing → is a technique of converting a large String to small String that represents the same String. A shorter value helps in indexing & faster searches. HashSet also uses HashMap internally

(12) HashMap doesn't allow duplicate ~~values~~ keys but allows duplicate values. That means a single key can't contain more than 1 value but more than 1 key can contain a single value.

(13) HashMap allows null key also but only once & multiple null values.

(Ex) to get values & keys: (Traversal of HashMap)

```
PSVM(...)
{
HashMap<String, Integer> map = new HashMap<>();
map.put("Vishal", 10);
map.put("Sachin", 30);
for(Map.Entry<String, Integer> e : map.entrySet())
S.O.P("key:" + e.getKey() + "Value:" + e.getValue());
```

(14) Operations on HashMap

(1) Adding Elements (put())

(2) Changing elements (put()) → Ex: hm.put(2, "Archer");

(3) removing elements (remove()) → Ex: hm.remove(2);

(4) Traversal of HashMap.

(*). ConcurrentHashMap

Syntax: class ConcurrentHashMap <K,V>

where K → type of keys maintained by this map
V → type of mapped keys.

where it extends AbstractMap & implements ConcurrentMap<K,V>, Serializable.

→ A hash table supporting full concurrency of retrievals & high expected concurrency for updates.

→ This class obeys the same functional specification as HashTable & includes versions of methods corresponding to each method of HashTable.

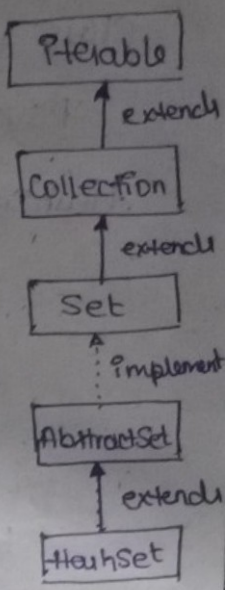
Eg: import java.util.concurrent.*;

class ConcurrentHashMapDemo {
PSVM(...) {

ConcurrentHashMap<Integer, String> m = new ConcurrentHashMap<>();
m.put(100, "Hello");
m.put(101, "Geeks"); m.put(102, "Geeks");
m.putIfAbsent(101, "Hello");
m.remove(101, "Geeks");
m.putIfAbsent(103, "Hello");
m.replace(101, "Hello");
S.O.P(m); } }

O/p: {100=Hello, 102=Geeks, 103=Hello}

HashSet



- HashSet class is used to create a collection that uses a hash table for storage.
- It inherits the AbstractSet class & implements Set interface.
- HashSet stores the elements by using a mechanism called hashing.
- It contains unique elements only, and allows null value & HashSet class is non synchronized.
- It doesn't maintain insertion order. Here elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operation. (imp)
- initial default capacity of HashSet is 16 & load factor is 0.75.

Syntax: public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable.

Methods:	Modifier & Type	Method
1)	boolean	add(E e)
2)	void	clear()
3)	Object	clone()
4)	boolean	contains(Object o)
5)	boolean	isEmpty()
6)	boolean	remove(Object o)
7)	int	size()
8)	Iterator<E>	iterator()
9)	Spliterator<E>	spliterator()

(Ex) import java.util.*;

```
class HashSet1 {
  PSVM ( --- ).
  HashSet<String> set = new HashSet<>();
        Set.add("achu");
        Set.add("anu");
        set.add("manu");
        Set.add("bindu");
    Iterator<String> i = Set.Iterator();
        while (i.hasNext())
        {
            S.O.P(i.next());
        }
    }
}
```

O/P: bindu
anu
achu
manu

(3) HashTable: → It is class implements a hash-table, which maps keys to values. Any non-null object can be used as a key or as value.

→ To successfully store & retrieve objects from a hashtable, the objects used as keys must implement the hashcode method & the equals method.

→ It is similar to HashMap, but is Synchronized.

Eg. stores key/value pair in hashTable.

Syntax: public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable.

here 'K' - the type of keys maintained by this Map.

'V' - the type of mapped values.

(Ex) import java.io.*;

class AddElementsToHashtable {

psvm(...) {

Hashtable<Integer, String> ht1 = new Hashtable<>();

Hashtable<Integer, String> ht2 = new Hashtable<Integer, String>();

ht1.put(1, "one");

ht1.put(2, "Two");

ht2.put(3, "Three");

ht2.put(4, "four");

S.O.P("Mappings of ht1 :" + ht1);

S.O.P("Mapp of ht2 :" + ht2);

}

o/p:- Mappings of ht1 : {2 = Two, 1 = one}

Mappings of ht2 : {4 = four, 3 = three}

(Ex) S.O.P("size of map is" + ht1.size());

// to check key is present & print it.

if(ht1.containKey("one")

Integer a = ht1.get("one")

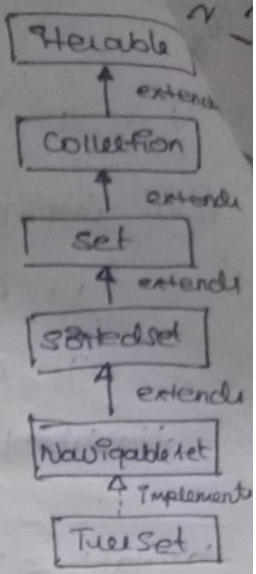
S.O.P("Value of key" + "\"one\"" is :-" + a);

}

o/p:- Size of Map - 2.

"one" is 1

(4) TreeSet



↳ this class implements the Set interface that uses a tree for storage.

↳ It inherits AbstractSet class & implements the NavigableSet interface.

→ the objects of the treeset class are stored in ascending order.

→ It contains unique elements only like HashSet & class is non synchronized.

→ Java treeset class access & retrieval times are quiet fast & doesn't allow null element.

→ ~~Syntax~~ Syntax: public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable.

(Ex) Class TreeSet {

psvm (...) {

TreeSet<Integer> set = new TreeSet<Integer>();

set.add(24);

Set.add(66);

set.add(12);

S.O.P("Highest Value" + set.pollFirst());

S.O.P("Lowest Value" + set.pollLast());

Iterator i = set.descendingIterator();

while(i.hasNext)

{ S.O.P(i.next()); } } }

O/p: "Highest Value" 66

"Lowest Value" 12

66
24
12.

(Ex) for string

```
TreeSet<String> al = new TreeSet<String>();
    al.add("Ravi");
    al.add("Vijay");
    al.add("Ajay");
Iterator<String> i = al.Iterator(); // ascending order
  while (i.hasNext()) {
  S.O.P (i.next()); } }
```

O/P: Ajay
Ravi
Vijay

(Ex)

```
TreeSet<String> al = new TreeSet<String>();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
  S.O.P("Initial set" + al); // [A,B,C,D]
  S.O.P("ReverseHeadset" + Set.descendingSet()); // [D,C,B,A]
  S.O.P("SubSet" + Set.SubSet("A", false, "D", true)); // [B,C,D]
S.O.P("HeadSet" + Set.headSet("C", true)); // [A,B,C]
S.O.P("TailSet" + Set.tailSet("C", false)); // [D]
S.O.P("HeadSet" + Set.headSet("C")); // [A,B]
S.O.P("HeadSet" + Set.tailSet("C")); // [C,D]
S.O.P("SubSet" + set.SubSet("A","D")); // [A,B,C]
```

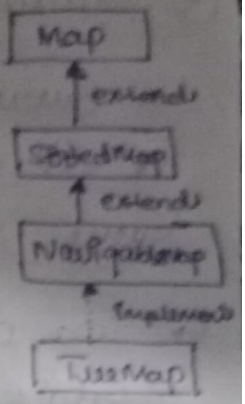
(6) Adding books to Set & printing all the books
the elements in TreeSet must be of a comparable
type

```
import java.util.*;
class Book implements Comparable <Book> {
int id;
String name, author;
int quantity;
Public Book(int id, String name, String author, int quantity)
{
    this.id = id;
    this.name = name;
    this.author = author;
    this.quantity = quantity; }
Public class TreeSetExample {
Psvm(...)
Set <Book> set = new TreeSet <Book> ();
// creating Books
Book b1 = new Book (221, "Jet mC", "Yashwant", 8);
Book b2 = new Book (123, "OS", "Galvin", 10);
// adding Books
  set.add(b1);
  set.add(b2);
// Traversing the set
for(Book b : set) {
S.O.P(b.id+" "+b.name+" "+b.author+" "+b.quantity);
} }
}
```

O/p: 123 OS Galvin 10

221 Jet mC Yashwant 8

TreeMap



- TreeMap class is red-black tree based implementation.
- It provides an efficient means of storing key-value pair in sorted order.
- TreeMap contains values based on the key. It implements the NavigableMap interface & extends AbstractMap class.
- Can't have null key but can have multiple null values

Ex: class TreeMap {

```
psvm(..) {
TreeMap<Integer, String> map = new TreeMap<Integer, String>();
map.put(100, "Rani");
map.put(102, "Ruchi");
map.put(101, "Vijay");
for (Map.Entry m : map.entrySet()) {
S.o.p(m.getKey() + " " + m.getValue());
} }
}
```

O/p: 100 Rani
101 Vijay
102 Ruchi

Ex: To remove

```
map.remove(101);
for (Map.Entry m : map.entrySet()) {
S.o.p(m.getKey() + " " + m.getValue()); } }
```

O/p: 100 Rani
102 Ruchi

(6) Queue in Java

↳ Queue is used to insert elements at the end of the queue & remove from the beginning of queue. It follows "FIFO concept"

↳ Java Queue supports all methods of collection interface including insertion, deletion etc.

→ LinkedList, ArrayBlockingQueue & PriorityQueue are most frequently used implementation.

(Ex)
```
import java.util.LinkedList;
import java.util.Queue;
Public class QueueExample {
    PSVM() {
Queue<Integer> q = new LinkedList<>();
for(int i=0; i<5; i++)          // add elements
q.add(i);                       [0,1,2,3,4]
S.O.P("Elements are": + q);     // display
int removedele = q.remove();
S.O.P("remove element" + removedele);   // remove
                                ↳ 0
S.O.P(q);                       [1,2,3,4]
int head = q.peek();
S.O.P("head of queue": + head);   // to view head of queue
                                  ↳ 1
int size = q.size();
S.O.P("size of queue": + size());   // size of queue
                                    ↳ 4
    }
}
```

```
Pq.add("Geeks");
Pq.add("For");
Pq.add("Geeks");
Iterator itr = Pq.iterator();
While (itr.hasNext()) {
S.O.P (itr.next() + " "); }
```

O/P: For Geeks Geeks.

(*) All Queue except the deque support insertion & removal at the tail & head of the queue respectively. The deque support element insertion & removal at both ends

(7) PriorityQueue: this class is implemented in the collection framework provide us a way to process the objects based on the priority. It is known that a queue follows first-in-first out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that's when the PQ comes into play.

(Ex)
```
import java.util.*;
class A {
  PSVM() {
Queue<Integer> Pq = new PriorityQueue<Integer>();
Pq.add(10);
Pq.add(20);
Pq.add(15);
S.O.P(Pq); // [10 15 20]
S.O.P (Pq.peek()); // 10
```

```
S.O.P(Pq.Poll());  // 10
S.O.P (Pq.Peek()); // 15
}
```

(8) LinkedList ; is a class which is implemented in the collection framework which inherently implements the linked list data structure.

→ It is a linear ds where the elements are not stored in contiguous locations & every element is a separate object with data part & address part

→ the elements are linked using pointer & address. Each element is known as a node.

→ due to the dynamicity & ease of insertions & deletions, they are preferred over the arrays & Queues.

(Eg) :
```
class A {
    PSVM(---)
    {
    Queue<Integer> ll = new LinkedList<Integer>();
    ll.add(10);
    ll.add(20);
    ll.add(15);
    S.O.P (ll.peek());
    S.O.P(ll.poll());
    S.O.P(ll.peek());
    }}
```

O/P: 10
10
15

Note: PriorityQueue & LinkedList are not thread safe PriorityBlockingQueue is one alternative Implementation if thread Safe Implementation is needed.

(9) Stack in Java

→ Stack is a linear ds that is used to store the collection of objects, which is based on LIFO. Collection framework provides many interfaces & classes to store the collection of objects. One of them is the Stack class that provide all the operations such as push, pop, search etc.

Top value	Meaning
-1	Stack is empty
0	have one element
N-1	stack is full.
N	stack is overflow.

(N) Stack is class that falls under the collection framework that extends the Vector class. It also implements interfaces List, Collection, Iterable, Cloneable, Serializable.

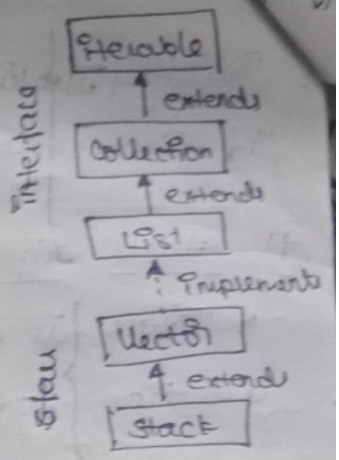
Syntax: Stack stk = new Stack();

(OR) Stack<type> stk = new Stack<>();

type → Integer, String etc.

(Ex)
```
import java.util.Stack;
Public class stackExample {
  PSVM (...)
  {
```

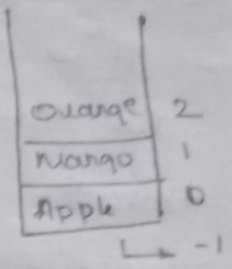
```
Stack < Integer > sk = new Stack < > ();
boolean result = sk.empty();
S.O.P("Is the stack Empty" + result);
~~SK.push  pushelement(SK,20);~~
        ~~Pushelmnt~~
~~SK~~. push (20);      // added
~~SK~~. push (13);
sk. Push (45);
sk. push (12);
sk. push (20);
sk. pop (20);   // removed

Pushelm (sk, 20);
Pushelm (sk, 10);      // add
Pushelem (sk, 20);

Popelm (sk);

try {
   Popelm (sk); }
Catch ( EmptyStackException e)
{  S.O.P(" empty stack"); }
}

Static void Pushelm ( Stack sk, Int x)
{
    sk.push (new Integer(x));   // call push method.
     S.O.P(" push -> " + x);
   S.O.P (" stack " + sk);   // prints modified stack.

Static void popelm ( Stack sk)
{
    S.O.P (" pop -> ");
  Integer x = (Integer) sk.pop();
```



```
s.o.p(x); // printing modified stack.
s.o.p (" stack" + sk);
} }
```

x)

```
Stack<String> sk = new Stack<>();
sk.push ("Apple");
sk.push ("mango");
sk.push ("Orange");
s.o.p ("stack" + sk);
String fruit = sk.peek();
s.o.p ("Element at top" + fruit); // Orange.

Int location = sk.search ("mango");
s.o.p (" location of mango" + location); // 1

Int x = sk.size();
s.o.p ("the stack size is:" + x); // 3

// Iterator over the stack.
Iterator iterator = sk.iterator ();
while (iterator.hasNext())
{   Object value = iterator.next();
    s.o.p( value); // Apple, mango, Orange
}

// for Each method.
sk.forEach(n ->
{   s.o.p(n)
});

}
}
```



(19) Vector

↳ Vector is like the dynamic array which can grow or shrink its size. Unlike array, we can store 'n' no/ of elements in it as there is no size limit.

→ It is part of collection framework & implements list interface. Vector class are recommended in thread safe implementation only. otherwise ArrayList will perform better in such case.

→ Vector is Synchronized. & contains legacy methods.

Syntax: Public class Vector <E> extends Object <E> implements List <E>, Cloneable, Serializable.

(Ex): Public class VectorExample {

PSVM() {

Vector <String> V1 = new Vector <String> ();

V1.add ("Tiger");

V1.add ('Lion');

V1.addElement ("Rat");

V1.addElement ("cat");

S.O.P ("Elements are" + V1);

S.O.P (V1.indexOf ("Tiger"));

S.O.P (V1.firstElement() + V1.lastElement()); } }

O/P: Elements are : [Tiger, Lion, Rat, cat]

0

Tiger, Cat