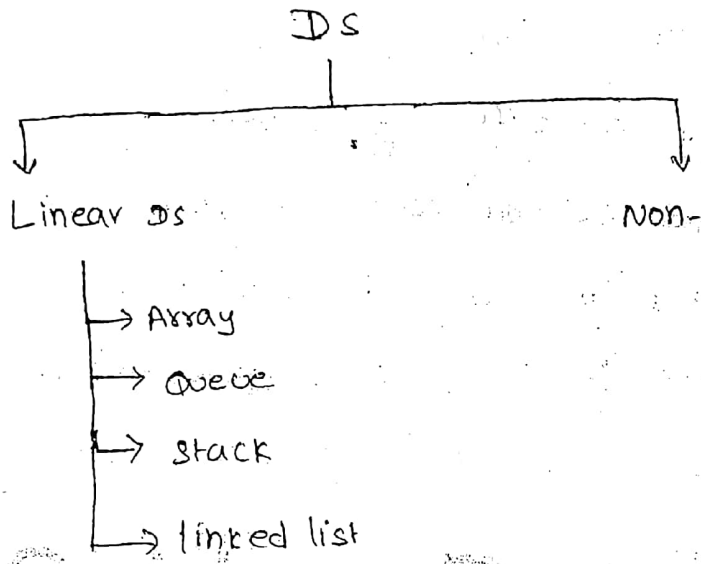
9686114422
9686995511

DATA STRUCTURES

→ It is the particular way of organising the data in the program to use them efficienty in the applications



Linear Data Structure:-

→ It is a type of data structure where the data is stored one next to another in linear order

Non- Linear Data Structure:-

→ It is a type of data structure where the data is stored according to some relations

9686114422
9686995511

→ The size of the array should be always specified using Byte (or) short (or) int data types only

→ Array size can never be specified as -ve number If we specify array size as -ve number then JVM throws NegativeArraySizeException.

→ If we specify the array size as zero, there will be no compile time errors (or) Run time Exceptions

→ * In java arrays are considered as objects.

→ In java arrays are created using new operator

→ The variable created for array is actually a reference variable which points to array object.

→ If you try to print the reference variable of the array then it always prints the String representation.

* Object array :-

→ It is a type of array where every bucket of the array is a reference variable.

→ Using the object array we can manage the objects easily.

→ Object array will be very useful whenever we want perform the same operations on the every object present in object array

9686114422
9686995511

```
class program
{
    psv main (String[] args)
    {
        Object [] ari = new Object[5];
        ari[0] = new object();
        ari [1] = new object();
        ari [2] = new object();
        ari [3] = new object();
        ari [4] = new object();

        for (int i=0 ; i< ari.length ; i++)
        {
            hi= ari[i]. hashcode();
            Sop (hi);
        }
    }
}
```


output:-

366712642

1829164700

20186995 54

1311053135

1183524 62

```
class Student
{
   String name;
   int eid;
   double marks;
   Student (String name, int eid, double marks)
   {
      this.name = name;
      this.eid = eid;
      this.marks = marks;
   }
   @Override
   Public String toString ( )
   {
      String info = sid + " " + name + " " + marks;
      return info;
   }
}
Class Program
{
  Ps v main (String [ ] args)
  {
    Student [ ] St = new student [5];
    St[0] = new Student ("smith", 7893, 451);
    St[1] = new student ("martin", 7641, 357);
    St[2] = new student ("miller", 7414, 400);
    St[3] = new student ("allen", 7454, 550);
    St[4] = new student ("scott", 7111, 521);
    for (int i= 0; i< St.length ; i++)
    {
      sop (St[i]);
    }
  }
}
```

Output:

7893 Smith 451.0

Java collection Framework (JCF)

Collection :-

→ Collection is a group of references which is represented as one unit

Properties of Collection :-

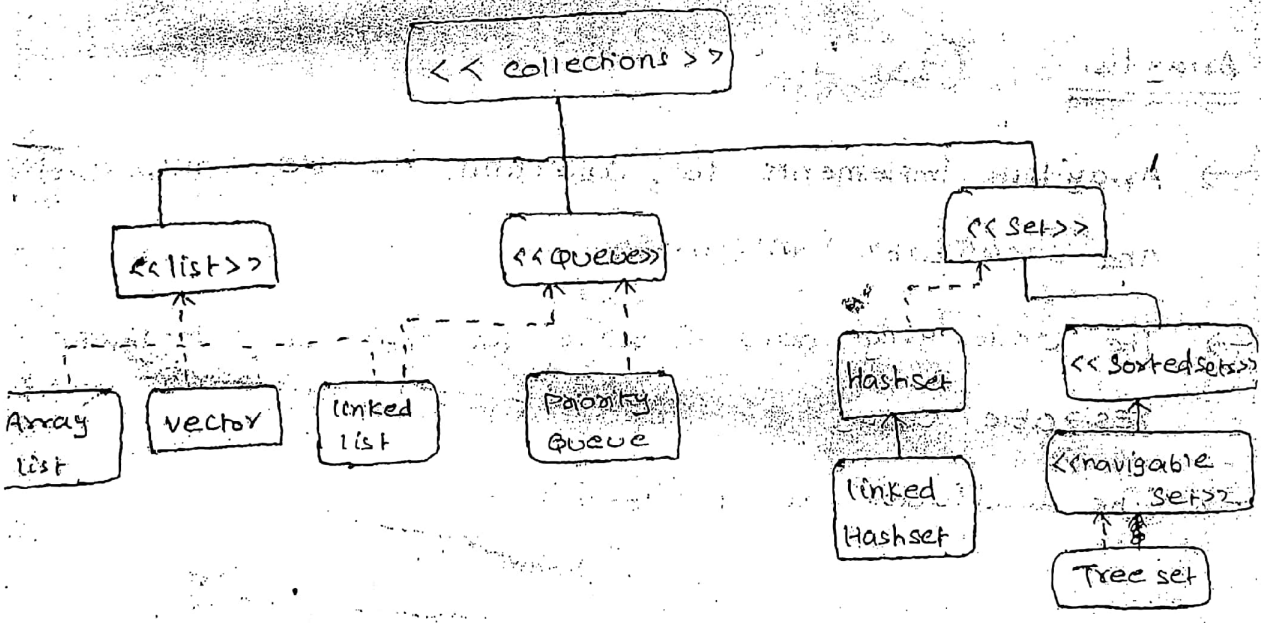
→ Collection supports heterogeneous data (references)

→ Collections are growable in nature, the size of the collection can be increased dynamically at run time

→ Every collection will have atleast one underlyning data structure

→ Every collection will have inbuilt methods for basic operations like add, remove, replace etc. on the collections

Collections Framework :-



Constructors of Arraylist :-

1. Arraylist ( ) :-

→ constructs an empty list with an initial capacity of 10

2. Arraylist (int initialcapacity) :-

→ Constructs an emptylist with specified initial capacity

3. Arraylist (collection c) :-

→ Constructs an Arraylist from the given collection.

→ The new capacity of arraylist is calculated as follows

$$\text{new capacity} = (\text{old capacity} * 3/2) + 1$$

Load factor :-

→ It is the thershold capacity which indicates when the new collection should be created with the new capacity.

→ Collection is the root interface of the java collection framework.

→ Java collection framework is a group of classes and interfaces which are used to manage the reference present in different collections

→ Every collection will have its own properties and hence every collection will contain the methods which specific to that type of collection.

<< list >>

→ It allows duplicates

→ It allows null values (multiple null values)

→ List has index

→ list preserves insertion order.

Array list :- (JDK 1.2)

→ Array list implements list, collection, iterable, clonable and serializable interfaces.

→ The underlying data structure for the arraylist is resizable array

→ The loadfactor is 1 (100%.)

#9686114422
#9686995511

```
import java.util.ArrayList;
class mainclass
{
  Psv main (String[] args)
  {
    ArrayList al = new ArrayList ();
    al.add (10);
    al.add ("hello");
    al.add (3.54);
    al.add (null);
    al.add (null);
    al.add (10);
    for(int index=0; index < al.size(); index++)
    {
      sop(al.get(index));
    }
  }
}
```


O/P:

10
hello
3.54
null
null
10

→ Within the Collection we cannot store primitive values

→ If we try to add primitive values for the given Collection then JVM

1. Converts primitive value to corresponding wrapper Class object
2. stores the address of the wrapper class Object within the given collection.

→ If we try to add any object reference to any Collection then the given reference will be upcasted to object type reference

→ While retreiving the reference if we want to access any subclass properties then, we should downcast the reference for the corresponding subclass type.


```
import java.util.ArrayList;
Class mainclass
{
  PSV main (String[] args)
  {
    ArrayList al = new ArrayList();
    al.add("hello");
    al.add(new String("java"));
    SOP(al.get(0));
    String s1 = (String) al.get(0);
    SOP(s1.length());
  }
}
```

o/p

hello

5

#9686114422
#9686995511

Vector :- (JDK 1.0)

→ Vector implements list, collection, iterable, clone and serializable interfaces.

→ The underlying data structure for the Vector is resizable array

→ The load factor is 1 (100 %)

Constructors of Vector :-

1. Vector ( ) :-

→ Constructs an empty list with an inital capacity of 10

2. Vector (int initialCapacity) :-

→ Constructs an empty list with specified inital capacity.

3. Vector (Collection c) :-

→ Constructs a list from the given collection


```
import java.util.Vector;
class mainclass
{
   Psv main (String[] args)
   {
      Vector vl = new Vector();
      vl.add(10);
      vl.add("hello");
      vl.add(3.54);
      vl.add(null);
      vl.add(null);
      vl.add(10);
      for(int index=0; index < vl.size(); index++)
      {
         vl.get(index);
      }
   }
}
```


o/p:-

10
hello
3.54
null
null
10

#9686114422
#9686995511

Set :-

→ set do not preserve insertion order

→ set do not have any index

→ set do not allow duplicates

→ only one null value is allowed

* How set preserves uniqueness?

→ Set preserves the uniqueness by comparing the hashcode values of the given object with every other object present in the set.

→ If the hashcode values are same then the corresponding object will be rejected.

Hash set :- (JDK 1.2)

→ Has set implements serializable, cloneable, Iterable, Collection and set interfaces

→ The underlined data structure of hasset is hashtable

Constructors of Hashset :-

Hashset () :- constructs a new, empty set with initial capacity (16) and load factor (0.75).

Hashset (int initialCapacity) :- Constructs a new empty set with specified initial capacity and default load factor (0.75)

Hashset (int initialCapacity, float loadfactor) :-

constructs a new empty set with the specified initial capacity and the specified load factor.

Hashset (Collection c) :-

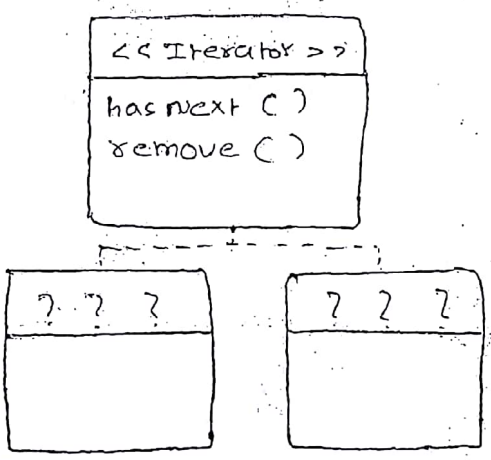
constructs a new set by converting any given collection to set

→ To retrieve the elements from the collection we can use

1. Iterator
2. Advanced for loop (For each loop)

→ Normally iterator (or) For each loop are used with collections which do not have index

Iterator implimentation :-



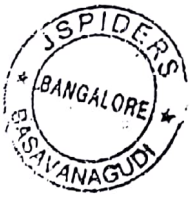

```
import java.util.HashSet;
import java.util.Iterator;
Class Mainclass
{
  psvm main (String[] args)
  {
    Hashset h1 = new Hashset ();
    h1.add (10);
    h1.add (20);
    h1.add (30);
    h1.add (10);
    h1.add (50);
```

#9686114422
#9686995511

```
Iterator il = hl.iterator();
while (il.hasNext() == true)
{
    sop (il.next());
}
}
```

Output :-

50
20
-10
30



Iterator :-

→ Iterator is used to retreive the elements from any collection with next method.

→ To retreive the elements using the iterator we have to follow 3 steps.

1. get the iterator object using any collection reference variable by calling iterator method.
2. use a while loop with hasNext, method which returns true if the next element is present else it returns false
3. use next method within the body of while loop which returns the elements from the collection.

→ The next method throws NoSuchElementException, if you try to retrieve the elements from the collection after the last element

#9686114422
#9686995511

For each loop :-

```
import java.util.Hashset;
class mainclass
{
  psv main (String[] args)
  {
    Hashset hi = new Hashset();
    hi.add("hello");
    hi.add("java");
    hi.add("Android");
    hi.add("apple");
    for (Object ref : hi)
    {
      sop(ref);
      String s1 = (String) ref;
      sop(s1.length());
    }
  }
}
```

Output:-

```
apple
5
java
4
Android
7
apple
5
```


Syntax:-

```
for (classname reference : collection_ref)
{
  Stmts - - -
  Stmts - - -
}
```

→ For each loop is used to retreive the elements from collections and also from object arrays.

→ For each loop internally creates an iterator to retrieve elements from the given collection (or) object array

Linked Hashset :- (JDK 1.4)

#9686114422
#9686995511

→ The underlined data structure for LinkedHashSet are

1. Hash table
2. Linked list

→ LinkedHasset preserves the insertion order

constructors of linked Hashset :-

LinkedHashset ( ) :-

→ constructs a new, empty set with initial capacity (16) and load factor (0.75)

~~LinkedHashset (int~~

LinkedHashset (int initialCapacity) :-

→ constructs a new empty set with specified initial capacity and default load factor (0.75)

LinkedHashset (int initialCapacity, float loadfactor) :-

→ constructs a new empty set with specified initial capacity and specified load factor

LinkedHashset (collection c) :-

→ constructs a new set by converting any given collection to set

#9686995511
#9686114422

Tree Set :-

Tree :-

→ A tree is the non-linear datastructure where the data represented according to some relation.

→ The tree is represented with nodes and links

→ The node consists of data, Address of left node and address of right node

Binary tree :-

→ It is a type of tree where a parent node can h maximum of 2 child nodes

Properties of Treeset :-

→ The underlined datastructure of Treeset is Red-black

→ Tree set implements Serializable, Clonable, iterable, colle navigableSet, set, SortedSet

Constructors of Treeset :-

Treeset() :- Constructs a new, empty tree set, sor according to the natural ordering of ele

Treeset (Comparator comp) :-

Constructs a new, empty tree set, sorted according to the specified comparator.

→ *** Every object added to the treeset should be implimenting comparable interface and override compare To method

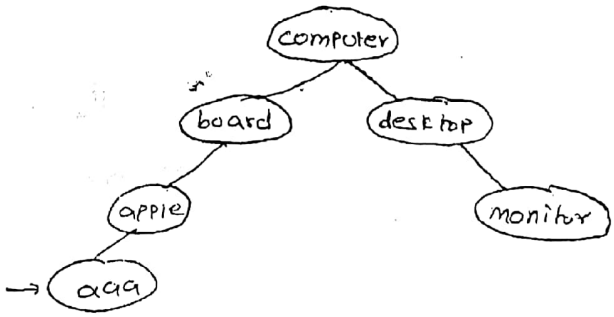
⇒ Non - comparable classes :-

String Buffer

String Builder

Object class etc.

```
import java.util.Treeset;
import java.util.Iterator;
Class Program
{
  PSVmain (String [] args)
  {
    Treeset hl = new Treeset ();
    hl.add ("computer");
    hl.add ("desktop");
    hl.add ("board");
    hl.add ("monitor");
    hl.add ("apple");
    hl.add ("aaa");
    Iterator il = hl.iterator();
    while (il.hasNext()== = true)
    {
      SOP (il.next());
    }
  }
}
```

Output :-

aaa
apple
board
computer
desktop
monitor

board.compareTo(computer) → left

desktop.compareTo(computer) → right

Left
Parent
right

#9686114422
#9686995511

Comparable interface:-

→ It helps in sorting the java objects.

→ It contains only one abstract method by the name compareTo

→ Whenever the objects needs to be compared to perform sorting (or) to decide polority compareTo method should be overridden to contain the logic for the compa

→ After comparing given two objects, compareTo() re
+ve integer if the given object value is greater than
-ve integer if the given object value is lesser than
(zero) 0 if the given object value is equal to another

Comparator interface:-

→ It is used to sort the objects in customized sorting

→ If we dont want to use natural sorting order then can customize the sorting order according to the application requirement with comparator interface

→ comparator interface contains an abstract method by name compare

→ comparator is a functional interface and supports lamda functions.

→ compare method compares given two objects and

#-9686114422
#-9686995511

```
import java.util.Comparator;
import java.util.Treeset;
import java.util.Iterator;

Class Mainclass3
{
   PSV main (String[] args)

   Comparator cmp1 = (Object obj1, Object obj2) ->
                     {
                       String s1 = (String) obj1;
                       String s2 = (String) obj2;

                       int value = s2.compareTo(s1);
                       return value;
                     };

Treeset t1 = new Treeset(cmp1);
t1.add ("computer");
t1.add ("beer");
t1.add ("duke");
t1.add ("makeupkit");
t1.add ("apple");
t1.add ("aaa");

Iterator il = t1.iterator();
While (il.hasNext() == true)
{
  Sop (il.next());
}
}
```

Output:-

```
makeupkit
duke
computer
beer
apple
aaa
```

9686114422
9686995511

Queue :-

→ Duplicates are allowed

→ Multiple Null values are allowed

→ Insertion order is preserved

→ It has Index.

Priority queue :- (JDK 1.5)

→ Duplicates are allowed

→ Not even one null value is allowed.

→ Insertion order is not preserved.

→ It does not have index

→ Objects are removed from the priority queue based Some priority.

→ Priority queue implements Serializable, Iterable, Collec Queue interfaces

→ The underlined datastructure of the priority queue is Priority queue data structure.

Constructors of priority Queue :-

PriorityQueue() :-

→ creates a priorityQueue with the default initial c that orders its elements according to their natural

PriorityQueue (int initialCapacity) :-

→ Creates a PriorityQueue with specified initial capa orders its elements according to their natural order

#9686114422
#9686995511

PriorityQueue (Collection c) :-

→ Creates a PriorityQueue by converting any Collection to PriorityQueue

```
import java.util.PriorityQueue;
import java.util.Iterator;
class customer implements comparable
{
  String name;
  String address;
  int priority;
  customer (String name, int priority)
  {
    this.name = name;
    this.priority = priority;
  }
  Public int compareTo (object obj)
  {
    customer ctl = (customer) obj;
    int value = ctl.priority - this.priority;
    return value;
  }
  Public String toString ( )
  {
    String info = name + " " + address + " " + priority;
    return info;
  }
```

```
Class mainclass
{
  PSV main (String[] args)
  {
    PriorityQueue q1 = new PriorityQueue();
    q1.add (new Customer ("Rama", 3));
    q1.add (new Customer ("Bhima", 2));
    q1.add (new Customer ("soma", 5));
    q1.add (new Customer ("Shyma", 5));
    q1.add (new Customer ("obama", 3));
    SOP (q1.size());
    while (q1.isEmpty() == false)
    {
      SOP (q1.pool);
    }
    SOP (q1.size());
  }
}
```

Output :-

Size = 5

Soma 5

Shyma 5

Obama 3

Rama 3

Bhima 2

Size = 0

#9686114022
#9686995511

Poll ():-

→ It returns the first element of the queue and also removes it from the given queue.

Peek():-

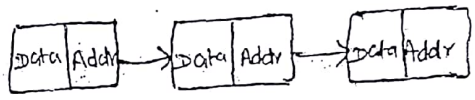
→ It returns the first element of the given queue but does not remove it from the given queue.

Linked list:-

→ It is a group of nodes which is linked to each other.

→ The node of a linked list contains

1. Data
2. Address of next node



→ It, implements list and queue interfaces.

→ Since it is implementing both list and queue it can be used as a list (or) normal queue (FIFO)

Constructors of linked list:-

LinkedList():-

→ Constructs an empty LinkedList

LinkedList(collection c):-

→ Constructs a LinkedList by converting any given collection to LinkedList

→ To retrieve the elements from the linked list we can use get(), poll(), and peak().

→ The underlined data structure for the linked list is Collection is LinkedList datastructure

```
import java.util.LinkedList;
class mainclass
{
  PSVmain (String[] args)
  {
    LinkedList lsl = new LinkedList();
    lsl.add(50);
    lsl.add(60);
    lsl.add(80);
    lsl.add(20);
    SOP(lsl.size());
    for (int index=0; index<lsl.size(); index++)
    {
      SOP(lsl.get(index));
    }
    SOP(lsl.size());
    while (lsl.isEmpty == false)
    {
      SOP(lsl.poll());
    }
    SOP(lsl.size());
  }
}
```


Output:-

```
4 (size)
50
60
80
20
4 (size)
50
60
80
20
```

#9686995511
#9686114422

#9686114422
#9686995511

→ Generics are used with the collections to acheive, to goals

1. Providing Type safety
2. To avoid unnecessary downcasting of objects

Syntax for generic Collection :-

CollectionName < Type > ref = new CollectionName < Type > ( ) ;

Ex:- ArrayList < String > al = new ArrayList < String > ( ) ;

→ Generics are specified with declaration of collection within angular brackets

Note :-

→ Java arrays are by default Type-safe


```
import java.util.ArrayList;
class Mainclass
{
  Psv main (String[] args)
  {
    ArrayList<String> al = new ArrayList<String> ();
    al.add (new String ("hello"));
    al.add (new String ("java"));
    al.add (new String ("android"));
    SOP ();  // al.add (new StringBuffer ("HTML")  C.T.E //.

    for (int index =0 ; index < al.size() ; index ++)
    {
      String s1 = al.get(index);
      SOP (s1.length());
    }
}
```

Output :-

5
4
7

27 JDK 1.8 Enhancements for Collections :- #9686[illegible]42;
#9686[illegible]50

Streams API :-

→ It was introduced to perform bulk operations on given collection

→ Stream is not a type of collection

→ The data present in the stream cannot be changed (or) manipulated

→ Streams are used to perform iterations over the collections to aggregate the data

→ The operators (or) iterations done through the Stream will be must faster.

Methods of Streams :-

1. stream() :-

→ This method returns the object of stream constructed from the given collection.

2. filter() :-

→ It is used to filter out the data of the Stream based on some conditions.

3. map() :-

→ It is used to modify the data once the data comes out of stream

4. foreach() :-

→ It is used to iterate over the given stream.

forEach ( ) :-

```
Ctrl  import java.util.ArrayList;
Class Mainclass   import java.util.stream.Stream;
{
  Psv main (String[] args)
  {
    ArrayList<String> al = new ArrayList<String>();
    for (int count = 1 ; count <= 10 ; count++)
    {
      al.add("hello");
    }
    Stream<String> s1 = al.Stream();
    s1.forEach (str -> Sop(str));
  }
}
```


→ All the operations of the streams will be executed parellely with a different thread and hence the execution time for the iterations and agregations will have lesser time of execution.

filter ( ) :-

```
Stream<String> s1 = al.stream();
Stream<String> f1 = s1.filter(str1 -> str1.length() > 4);
f1.forEach (str2 -> Sop(str2));
```

Map ( ) :-

```
Stream<String> s1 = al.stream();
Stream<String> f1 = s1.map(str1 -> str1.toUpperCase());
f1.forEach (str2 -> Sop(str2));
```

collect ( ) :-

→ It is used to collect the result of filter ( ) (o

Collectors Class :-

→ This class helps in converting the given result of map (or) filter to any type of collection.

methods of collectors :-

1. toList ( ) :-

converts the result of the stream to List colle

2. toSet ( ) :-

converts the result of the stream to set collec

3. toQueue ( ) :- ✗

converts the result of the stream to Queue colle


```
List<String> ls1 = fl.collect(collector.toList());
Queue<String> q1 = fl.collect(collector.toQueue());
Set<String> s1 = fl.collect(collector.toQueue());
```

```
Stream<Employee> s1 = a1.stream();
Stream<Employee> fl = s1.filter(result -> result.s
fl.forEach(result -> sop(result));
```

Reducing no. of lines of code with Stream :-

```
import java.util.ArrayList;
Class mainclass
{
  Psv main (String [] args)
  {
    ArrayList<String> al = new ArrayList<String>();
    al.add("hello");
    al.add("android");
    al.add("angularjs");

    al.stream()
      .filter(str -> str.length() > 5)
      .foreach(str2 -> SOP(str2));
  }
}
```

Output :-

android
angularjs

ParellelStream() :-

→ It returns a Stream object where all the operations on the Stream (filter, map) will be executed by creating a new thread

```
import java.util.ArrayList;
Class mainclass
{
  Psv main (String [] args)
  {
    ArrayList<Integer> al = new ArrayList<Integer>();
    for (int i=1; i<=100000; i++)
    {
      al.add(i);
    }
    al.stream().map(str -> str.mapToDoubleValue())
```

Output :-

1.0
2.0
3.0
4.0
5.0
6.0

#9686114422
#9686995511

```
ArrayList < String > al = new ArrayList < String > ();
for (int i=1 ; i <= 10,000 ; i++)
{
    String Vl = String.valueOf (i);
    al.add (i);
}.
```



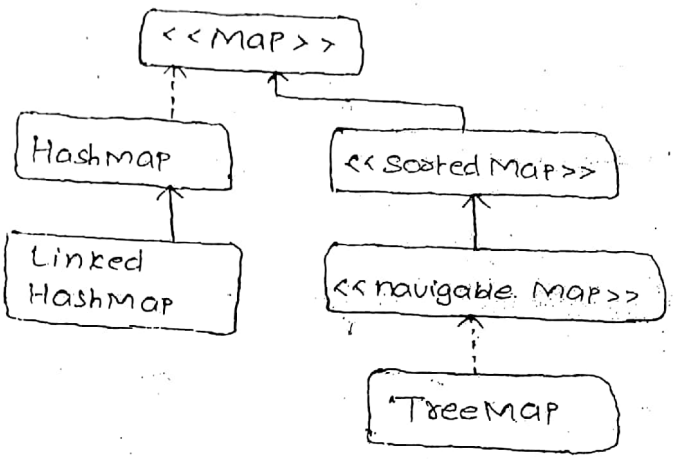
```
                                    stream()          Integer
Stream < Integer > st = al.map (Strl -> Strl.parseInt
                        (or)
List < Integer > lsl = al.stream ()
                        .map (il -> Integer.parseInt (
                        .collect (collectors.toList());

lsl.stream()
    .map((i -> (i*2)
    .foreach (i -> sop (i*2));

lsl.stream().foreach (i -> sop (i/5));
```

9686114422
9686995511

Map:-



→ The data is stored in key-value format

→ The keys of map cannot be duplicate

→ only one null can be allowed as key

→ values can be duplicate

→ multiple null can be used as values

→ Map interface donot extends collection interface

→ we can add the elements to the map using put()

Syntax:- put (key, value);

→ The key and value can be of any data type

→ To retreive the elements from the hashMap we can use get( ) Syntax:- get(key);

→ If we pass an invalid key then get() returns null value.

→ If we try to add duplicates then the old value will be replaced with the new value.

#9686114422
#9686995511

```
import java.util.HashMap;
Class mainclass
{
  PSV main (String[] args)
  {
    HashMap hl= new HashMap();
    hl.put("key1", 58);
    hl.put("key2", 78);
    hl.put("null", 87);
    SOP (hl.get("key1")); → 58
    SOP (hl.get(null)); → 87
    SOP (hl.get("key123")); → null
    SOP (hl.get
    hl.put("key1", 12345);
    SOP (hl.get("key1")); → 12345
  }
}
```


Output:-

58
~~78~~ 87
null
12345