Mangal
Date: / /201

Java Collection Framework JCF

* Collections are growable in nature.
* Collections supports heterogeneous data.
  (we can have any type of data)
* Collection is a framework that provides an architecture to store and manipulate the data, ie container where it allows no. of elements together in single unit.

Note: Why collection

Example: * when we try to store the data or when we write the program we know how much the data is, whats the size and how much space needed to store the data, ie in case of Array, we know that array size & the data would be stored in array in a particular size, but what if we want to store the data at runtime or dynamic, if we change array size at runtime it may cause and loco docon the performance.

* In order to overcome these disadvantage
  - ✓ * when we want to store the data at runtime
  - ✓ * where we don't want to specify the size of the object, then we go for collection.

x ----- x ----- x ----- x ----- x -----

* where we create an object, size is not defined we just store data at runtime → collection.

Why collections?

Mangal
Date: 1 / 201

Array size is fixed, size cannot be increased dynamically. The elements can be any Built in or user defined type. The insertion and deletion of elements in the array is also costly in terms of performance.

Generally data is obtained from a data source like file or database, it can be single object/ multiple objects. When one is not sure, it is safe to define return type as collection since it can hold multiple objects. This even works if one or no objects is returned.

* Basic Operation on any collection

1) Adding object in the collection dynamically/ at runtime.
2) Removing the object from a collection.
3) Iterating through collection and listing the objects
4) Retrieving Specific object from a collection
5) Searching Specific object from a collection
6) Deleting particular object from a collection.

* Collection have inbuilt methods to perform basic operations like add(), remove(), replace(), Update() etc on the collection itself.

* Every collection will have atleast one underlying data structure.

* A collection is group of references which is represented as one unit.

* within a collection it is not possible to store primitives.

↳ because

↳ container want objects, and primitive dont derive from object.

↳ Since java primitives are not coneidered objects, you would need to create a separate collection class for each of these primitives

↳ Ex: There is a class called Integer that supports int. you can not use the primitive wrappers to hold values in a collection.

Note: wrapper class —definition→ provides the mechanism to convert primitive into object and object into primitive.

Wrapper class:

* Java is an OOP language, so we need to deal with objects many times like in Collections, Serialization, Synchronization etc, let us see the different scenarios where we need to use the wrapper classes.

i) Change the value in method:

java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value (^o^)

ii) Serialization: we need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.

iii) Synchronization: java synchronization works with objects in multithreading.

iv) java.util package: provides the utility classes to deal with objects.

v) Collection Framework: java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc) deal with objects only.

The eight classes of the java.lang package are known as wrapper classes in Java. The list of eight wrapper classes are given below in the table 1.

Table-1

Primitive Type	Wrapper Class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

* Since J2SE 5.0 autoboxing and unboxing features

Autoboxing:

Conversion of primitive datatype into its corresponding wrapper class is known as autoboxing.

Example: byte to Byte
char to Character
long to Long etc

* Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitives into objects.

Program: Convert primitive into objects (Autoboxing)

```
public class WrapperExample {
public static void main(String args[])
{
//converting int to Integer
int a=20;
Integer i = Integer.valueOf(a); //converting int to Integer explicitly
Integer j = a; //autoboxing, now compiler will write Integer.valueOf(a) internally
S.o.P(a+" "+i+" "+j);
}
}
```

output:

20 20 20

Unboxing:

* The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing.
* Since java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Program: Wrapper to primitive (unboxing)

Date: / /201

```
public class WrapperExample {
public static void main(String args[])
{
// converting Integer to int
Integer a = new Integer(3);
int i = a.intValue(); //converting Integer to int explicitly
int j = a; // unboxing, now compiler will write a.intValue()
System.out.println(a + " " + i + " " + j);
}}
```

output:

3 3 3

Program: Convert all primitives onto its corresponding wrapper objects and vice-versa.

```
public class WrapperExample {
public static void main(String args[]){
byte b = 10;
short s = 20;
int i = 30;
long l = 40;
```

```
float f = 50.0f;
double d = 60.0d;
char c = 'a';
boolean b1 = true;
// Autoboxing : Converting primitives into objects.
Byte bobj = b;
Short sobj = s;
Integer iobj = i;
Long lobj = l;
Float fobj = f;
Double dobj = d;
Character cobj = c;
Boolean blobj = b1;
//printing objects.
S.O.P ("printing objects values");
S.O.P ("Byte obj : " + bobj);
    :
S.O.P ("Boolean obj : " + blobj);
//unboxing : converting objects to primitives
byte bvalue = bobj;
    :
boolean blvalue = blobj;
```

```
// printing primitives
S.O.P("printing primitives");
S.O.P("byte value : " + bvalue);
    :
    :
S.O.P("boolean value : " + blvalue);
}}
```

Output:

```
printing object values
Byte object : 10
Short         20
  :           30
  :           40
              50.0
              60.0
              a
Boolean object : true

  printing primitive values
bytevalue : 10
  :         20
  :         30
  :         40
            50.0
            60.0
            a
booleanvalue : true
```

Custom Wrapper class in java

Manpal
Date: / /201

* Java Wrapper classes wrap the primitive data type; that is why it is known as wrapper classes.

* We can also create a class which wraps a primitive data type. So we can create a custom wrapper class in java.

Program: Creating the custom wrapper class

```
class Example
{
  private int i;

  Example()
  {
    3

  Example(int i){
    this.i = i;
  3
  public int getValue(int i){
    this.i = return i;
  3
```

```
public void setValue(int i)
{
 this.i = i;
}

@Override
public String toString()
  {
   return Integer.toString(i);
  }
}

public class MainClass{
  public static void main(String[] args){
  Example e = new Example(10);
   S.O.P(e);
  }
}
```

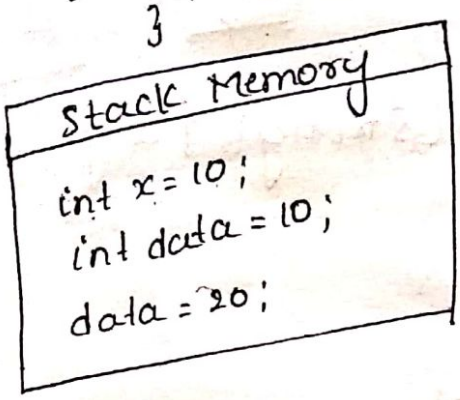
output: 10

*Difference B/w Call by value V/s Call by Reference : (◕‿◕)

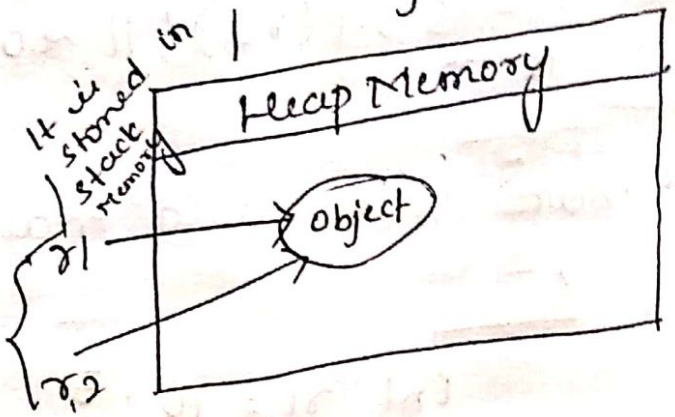
* There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value.

* The changes being done in the called method, is not affected in the calling method.

```
main()
{ int x=10;                    // Primitive Data
  modify(x);
  s.o.p(x);//o/p:10
}
public void modify(int data)
{   data.=20;
}
```




```
main(){
  Rectangle r1 = new Rectangle();//
  r1.length = 10;
  modify(r1)
  s.o.p(r1.length) // o/p:20
}
public void modify(Rectangle r2)
{   r2.length=20;
}
```



Program: call by reference

```
public class Example {
  public static void modify(Rectangle r2)
    {  r2.length = 40;
       s.o.p(r2.length);   // 40
    }
  public static void main(String[] args)
    {
  Rectangle r1 = new Rectangle();
     r1.length = 10;
     modify(r1);
     s.o.p(r1.length); // output: 40
  }
}
```

Program: call by value

```
public class Example {
    public static void modify(int r2)
    {
        r2 = 140;
        S.O.P(r2); // 140
    }
    public static void main(String[] args)
    {
        int r1 = 10;
        modify(r1); // 10;
        S.O.P(r1); // 10
    }
}
```

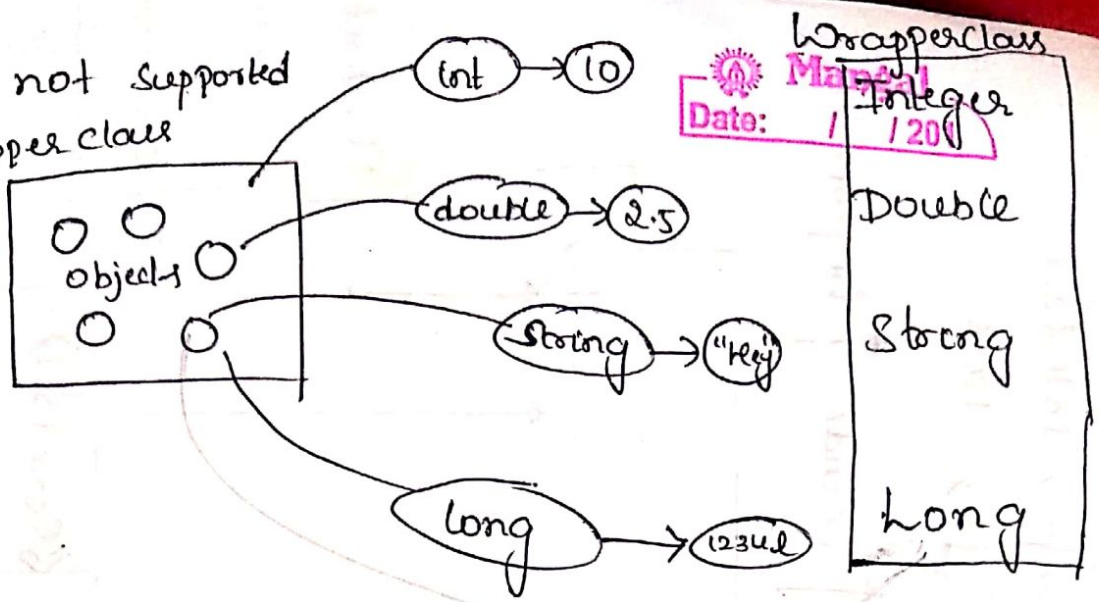
Call by reference:

* Original value is changed if we make changes in the called Method.
* If we pass object in place of an primitive value, original value will be changed.

---

* Within a collection it is not possible to store primitives.
* If you try to store primitive values, then the object of corresponding wrapper class will be created and its address or reference will be stored in the given collection.

Note:
Primitives not supported in the wrapper class



Date: / / 20

* In Java collection framework, collection is an root interface.

Note:

- What is collection in java? ***

→ A collection represents a single unit of objects, ie a group.

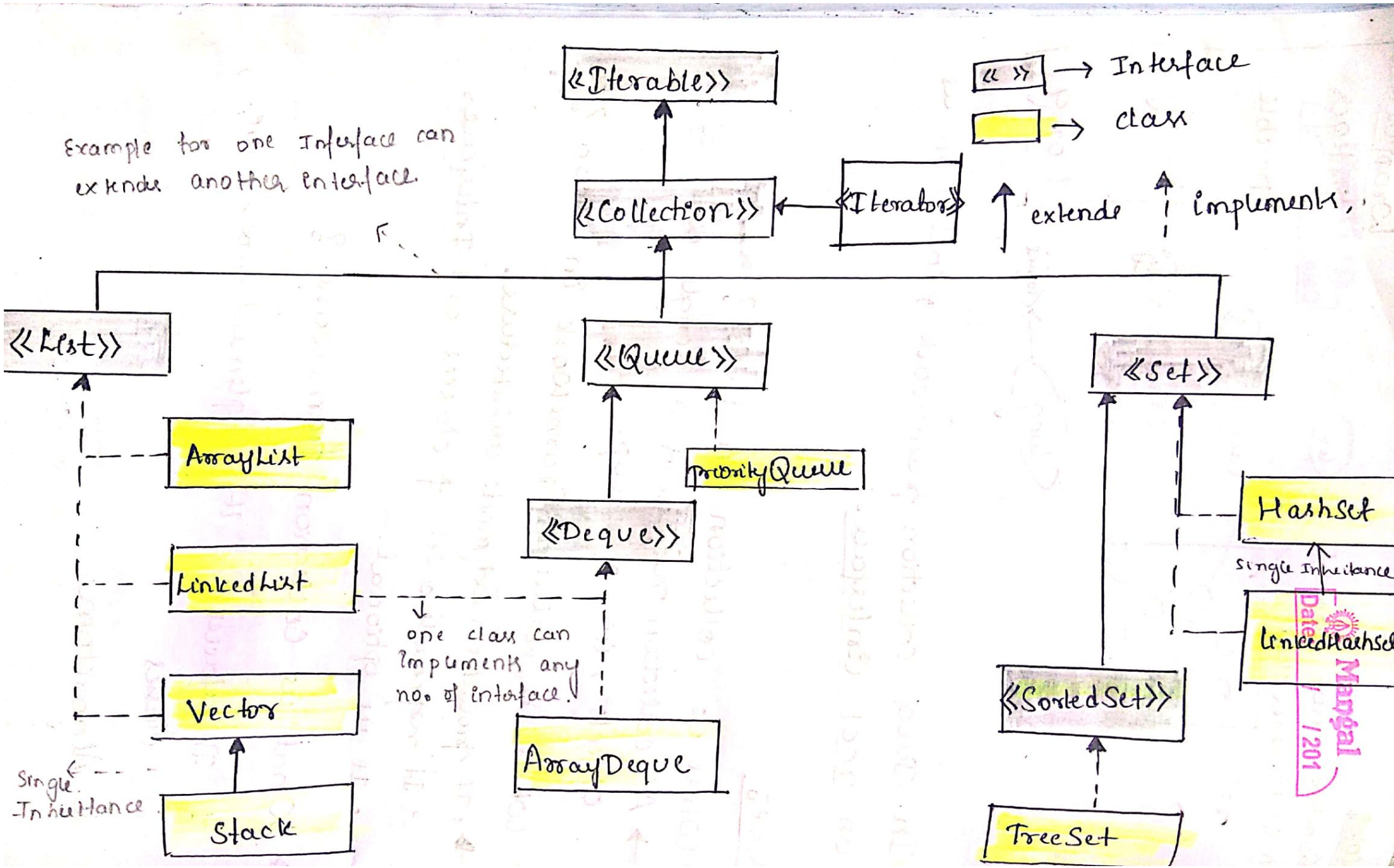
- What is ~~collection~~ Framework in Java?

→ • It provides readymade architecture.
• It represents a set of classes and Interfaces
• It is optional.

- What is collection framework?

→ Collection Framework represents a unified architecture for a group of objects. It has (i.e., Sorting and manipulating its implementations,

1. Interfaces and
classes
2. Algorithm



There are three types of collection:

Mangal
Date: / /201

1) List
2) Set
3) Queue

Methods of Collection Interface:

(1) public boolean add(E e)
↳ It is used to insert an element in this collection

(2) public boolean addAll(Collection<? extends E> c)
↳ It is used to insert the specified collection elements in the invoking collection.

(3) public boolean remove(Object element)
↳ It is used to delete an element from the collection.

(4) public boolean removeAll(Collection<?> c)
↳ It is used to delete all the elements of the specified collection from the invoking collection.

(5) default boolean removeIf(Predicate<? super E> filter
↳ It is used to delete all the elements of the collection that satisfy the specified predicate.

(6) public boolean retainAll(Collection<?> c)
↳ It is used to delete all the elements of invoking collection except the specified collection.

(7) public int size()
↳ It returns the total no. of elements in the collection.

(8) public void clear()

↳ It removes the total no. of elements from the collection.

(9) public boolean contains(Object element)

↳ It is used to search an element.

(10) public boolean containsAll(Collection<?> c)

↳ It is used to search the specified collection in the collection.

(11) public Iterator iterator()

↳ It returns an iterator

(12) public Object[] toArray()

↳ It converts collection into array

(13) public <T> T[] toArray(T[] a)

↳ It converts collection into array, here, the runtime type of the returned array is that of the specified array

(14) public boolean isEmpty()

↳ It check if collection is empty.

(15) default Stream<E> parallelStream()

↳ It returns a possibly parallel Stream with the collection as its source.

(16) default Stream<E> Stream()
↳ It returns a sequential Stream with the collection as its source.

(17) default Spliterator<E> Spliterator()
↳ It generates a Spliterator over the specified elements in the collection.

(18) public boolean equals(Object element)
↳ It matches two collections.

(19) public int hashCode()
↳ It returns the hashcode number of the collection.

Iterator interface: Iterator interface provides the facility of iterating the elements in a forward direction.

(1) public boolean hasNext()
↳ It returns true if the iterator has more elements otherwise it returns false.

(2) public Object next()
↳ It returns the elements and move the cursor pointer to the next element.

(3) public void remove()
↳ It removes last element returned by the iterator. It is less used.

~~Collection HashMap Employee details~~

Date: / /201

Iterable Interface:

* The Iterable interface is the root interface for all the collection classes.
* The collection interface extends the Iterable interface, and therefore all the subclasses of collection interface also implement the iterable interface.
* It contains only one abstract method. i.e

```
Iterator<T> iterator()
```

* It returns the iterator over the elements of type T

Collection Interface:

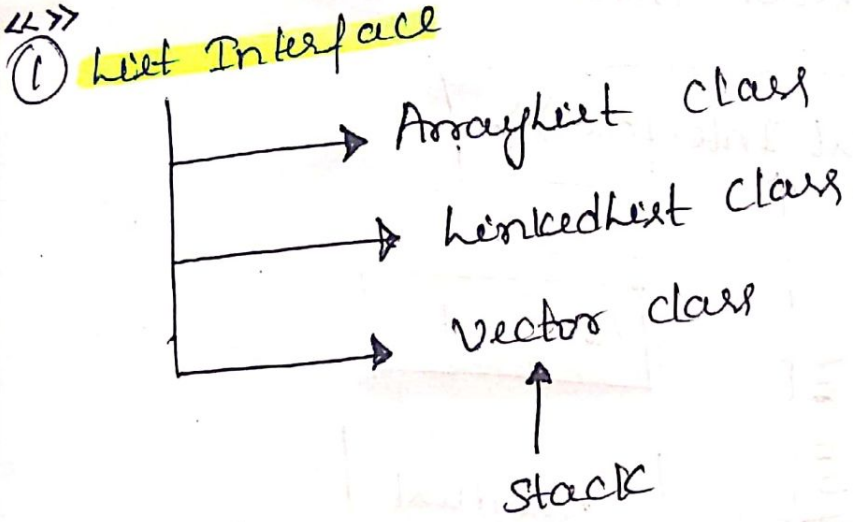
* The collection interface is the interface which is implemented by all the classes in the collection framework.
* It declares the methods that every collection will have. In other words, we can say that the collection interface builds the foundation on which the collection framework depends.
* Some of the methods of collection interface are
  * boolean add (Object obj)
  * boolean addAll (Collection c)
  * void clear()

* which are implemented by all the subclasses of collection interface.

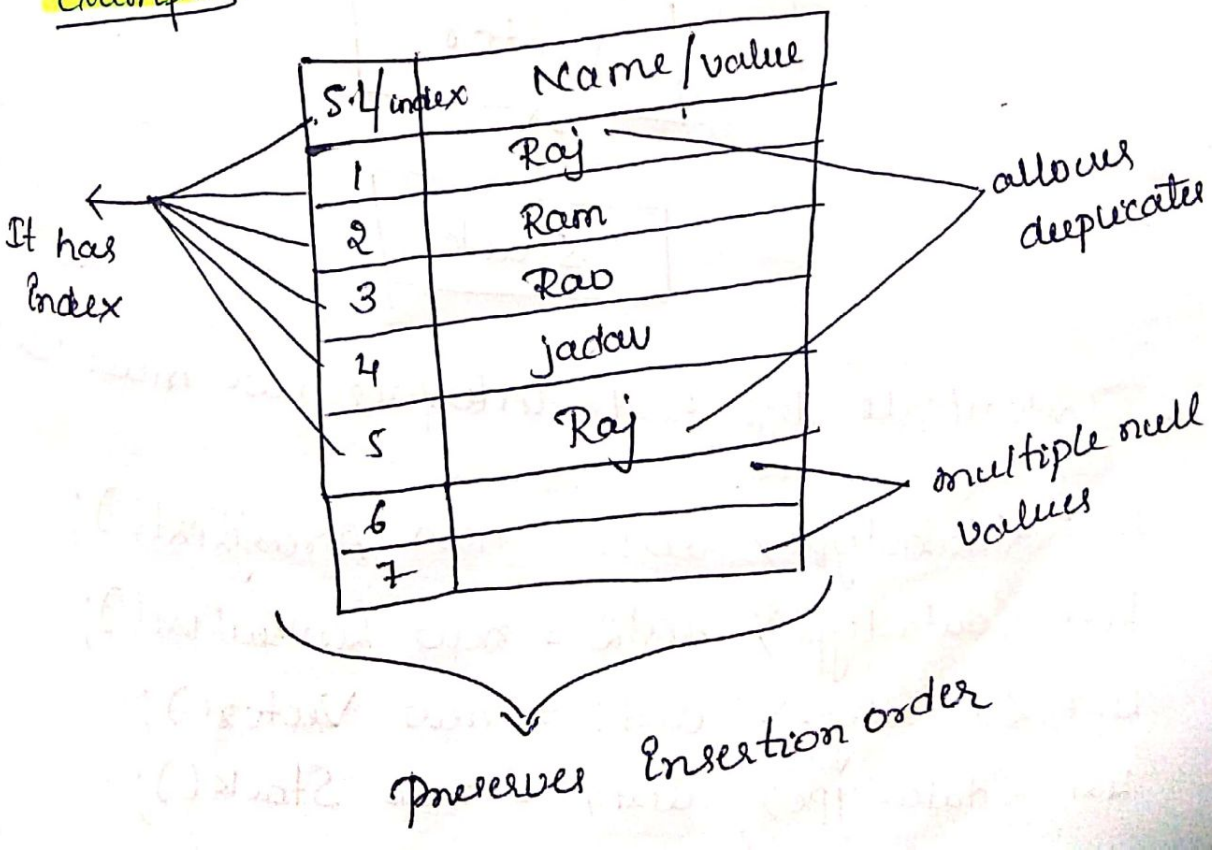
Mangal
Date: / /201

There are 3 types of collection Interface.

1. List Interface.
2. Set Interface.
3. Queue Interface.



Example:

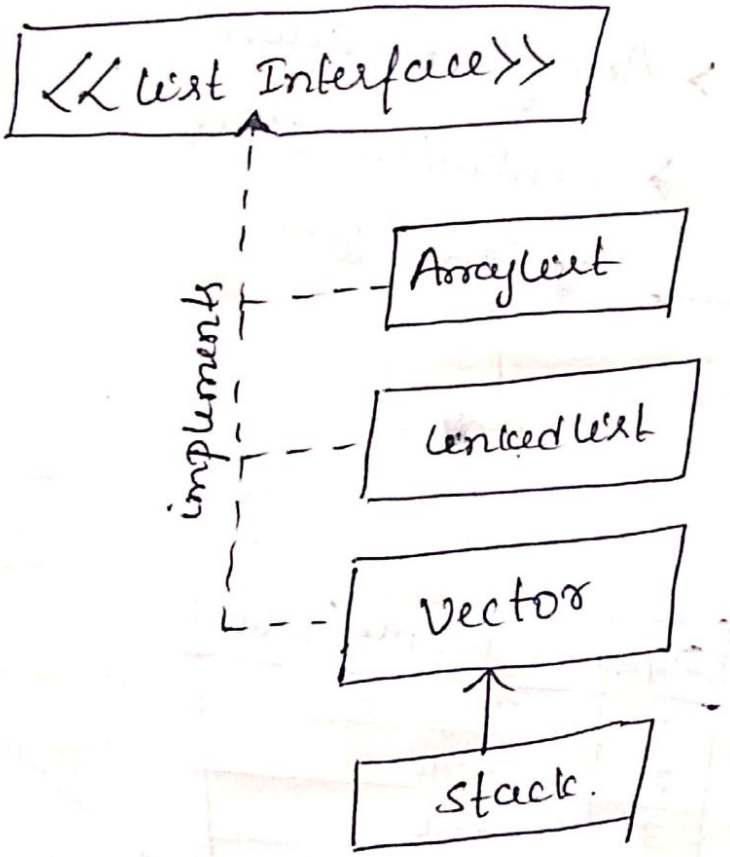


* List interface is the child interface of collection interface.

Mangal
Date: / /201

* It inhibits a list type data structure in which we can store the ordered collection of objects.

* List has Index
* List allows duplicates
* List allows multiple Null values
* List preserves Insertion order



To instantiate the List Interface, we must use:

```
List <data-type> list1 = new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();
```

* There are various methods in List Interface that can be used to insert, delete, and access the elements from the list.

a) Arraylist

* the Arraylist class implements the List interface.
* It uses a dynamic array to store the duplicate element of different data types.
* It has index.
* It allows multiple null value.
* It preserves Insertion order.
* It is growable in nature
* Arraylist implements List, RandomAccess, Clonable, Serializable Interface.
* It is non-Synchronized.
* The elements stored in Arraylist class can be randomly accessed.
* The underlying data structure of arraylist is Resizeable array

Note: It is Impossible to use subscript[] in an array list.

Note: Diff b/w array and arraylist

Mangal
Date: / /201

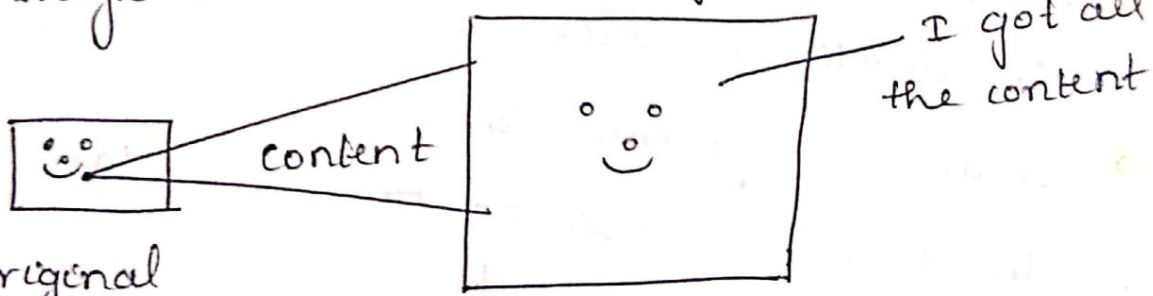
Array

* both used to store elements
* Fixed length
* It can store primitives or objects

ArrayList

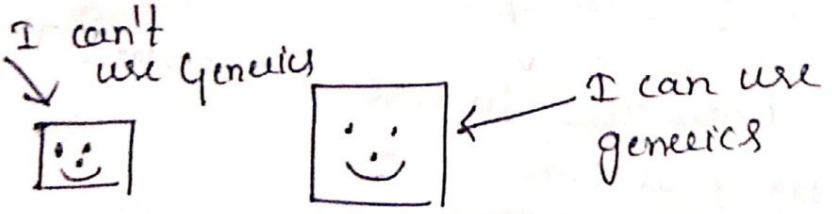
* Store elements
* Variable Length
* It can only store objects like turtles and pixels...

* We can create the wrapperclass & then we can use it in an ArrayList.
* ArrayList has the ability to re-size itself



original Array inside ArrayList

New bigger Array



ArrayList<T>

↑ Generic Type

ArrayList<T> al = new ArrayList<T>();

↑

ArrayList<Integer> al = new ArrayList<Integer>();

* Replacing a Generic Type is called as a Generic Instantiation

* Generic types must be replaced with reference types, Not with primitives!

Ex: Integer, Dog etc

Program:

```
import java.util.ArrayList

class Mainclass
{
  public static void main(String[] args)
  {
     S.O.P("Pgm start");
     ArrayList alist = new ArrayList();
     alist.add(10);
     alist.add(null);
     alist.add(20);
     alist.add(10);
     alist.add(30);
```

```
for(int i=0 ; i < alist.size() ; i++)
    {
      S.o.p (alist.get(i));
    }
  S.o.p(" Pgm ends ");
} }
```

Mangal Date: / /201

o/p :-

```
Pgm starts
10
null
20
10
null
30
Pgm ends
```

Note:

Constructor overloading

↓ Example:

Arraylist Constructor

b) Vector

```
import java.util.Vector;
Class Mainclass
{
  public static void main(String[] args){

  Vector v = new Vector();

   v.add(10);
   v. add(null);
   v. add(20);
   v. add (10);
```

```
    v.add(30);
    for(int i=0; i<v.size(); i++)
    {
        s.o.p(v.get(i));
    }
}
```

Mangal
Date: / /201

O/P:

```
10
null
20
null
30
```

Check the diffe b/w ArrayList and vector

↓ not-synchronized

↓ Synchronized

Program 2:

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

Class Test {

P.s.v.m(String args[]) {
// Creating arraylist
ArrayList<String> l = new ArrayList<String>();

// Adding object in arraylist
   l.add("Ram");
   l.add("Raj");
   l.add("Ram");
   l.add("Rao");
// Traversing list through Iterator
   Iterator<String> i = l.iterator();
   while(i.hasNext()) {
```

O/P:

```
Ram
Raj
Ram
Rao
```

```
s.o.p(i.next());
}}}
```

Mangal
Date: / /201

Constructors of ArrayList:

Note: using constructor, size can be increased

① ArrayList()
↳ Construct an empty list with an initial capacity of ten.

② ArrayList(Collection c)
↳ Construct a list containing the elements from given collection.

③ ArrayList(Int initialCapacity)
↳ Constructs an empty list with the specified initial capacity.

Load Factor:

↳ It decides when should be the new collection created based on the capacity.

↳ Load factor of arraylist is 1.0 (100%)

↳ Load factor can not changed, initial size can be varied.

Advantage of Arraylist

Mangal Date: / /201

- The time required to retrive the data from the array list is very less (and is constant)

- Disadvantages:
  - ↳ The time required to add the elements will increase with increasing number of elements.
  - ↳ The methods of arraylist are not synchronized and hence arraylist is not threadSafe.

Vector Class:

* Vector uses a dynamic array to store the data elements.
* It is similar to Arraylist.
* However, it is Synchronized and contains many methods that are not the part of Collection Framework.
* Vector implements List, Randomaccess, Clonable Serializable interface.
* The underlying data structure of vector is Resizable

Note: It is impossible to use subscript [] in an vector.

Example :

Mangal
Date: / /201

```
import java.util.*;
public class Test {
public static void main (String args[]){
Vector <String> v = new Vector<String>();
   v.add("Ram");
   v.add ("Raj");
   v.add ("Rao");
Iterator<String> i = v.iterator();
while(i.hasNext()) {
S.O.P(i.next()); }}}
```

O/P:

```
Ram
Raj
Rao
```

Constructor of Vector:

1. Vector()
   ↳ Construct an empty list with an initial capacity of Ten
2. Vector(Collection c)
   ↳ Construct a list containing the elements from given collection
3. Vector(int initialcapacity)
   ↳ Construct an empty list with the specified initial capacity.

Load Factor:

Mangal Date: / /201

* Load factor decides when should be new collection created based on capacity.
* Load factor of vector is 1.0 (100%)

Advantage:

* the time required to retrieve the data from vector is less.

Disadvantage:

* the time required to add the elements will increase with increasing number of elements.
* the methods of vector are synchronized and vector is threadSafe.

Program:

```
Import java.util.ArrayList;
Class Shoes
{
    double size;
    String color;
    String brand;
    double price;
Public Shoes (double size, String color, String brand,
                double price)
    {
        this.size = size;
        this.color = color;
        this.brand = brand;
```

```
        this.price = price;
    }
@Override
    public String toString()
    {
    return brand + " " + color + " " + size + " " +
            price ;
} } }
```

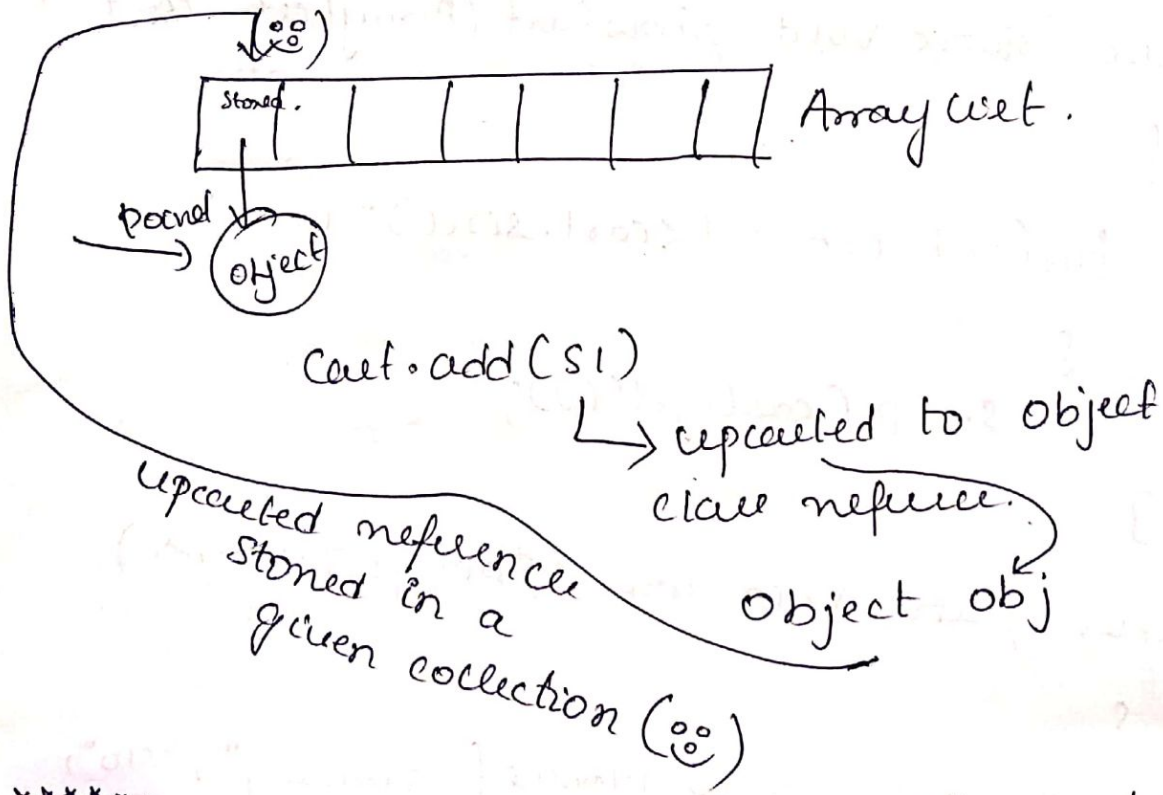
```
Class Mobile
  {
      String brand;
      String model;
      String color;
      int ram;
      double price;

Public Mobile (String brand, String model,
                String color, int ram, double price)
    {
      this.brand = brand;
      this.model = model;
      this.color = color;
      this.ram = ram;
      this.price = price;
    }
```

```
@Override
public String toString()
{
return brand + " " + model + " " + color + " " +
       ram + " " + price;
}}}

class MainClass
{
public static void showCart(ArrayList cart)
  {
   for(int i=0 ; i<cart.size(); i++)
    {
     s.o.p(cart.get(i));
}}
public static void main(String[] args)
  {
Mobile m1 = new Mobile("samsung", "s10",
                "black", 128, 40000);

Shoes s1 = new Shoes(6, "white", LM10, 2000);

Mobile m2 = new Mobile("redme", "3sprime",
                "gold", 128, 20000);

Shoes s2 = new Shoes(5, "blue", 1x20, 1000);
```

```
ArrayList cart = new ArrayList();
```

Date: / /201

```
cart.add(m1);
cart.add(m2)
cart.add(s1);
cart.add(s2);
showCart(cart);
}
}
```



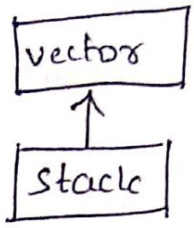
Note: Whenever we add any reference to the collection, it will be upcasted to object class reference and the upcasted reference will be stored in the given collection.

supermost class

c) Stack

Mangal
Date: / /201

* The stack is the subclass of vector.

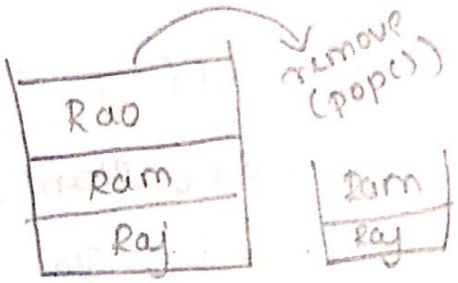


* It Implements LIFO datastructure. i.e Stack.

* The stack contains all of the methods of Vector class and also provides its methods like
  - boolean push() → push new item into the stack
  - boolean peek() → Get the top item of stack, without removing the item
  - boolean push(object o), which defines its properties.

Example:

```
import java.util.*;
public class Test{
public static void main(String arg[]){

Stack<String> s = new Stack<String>();
  s.push("Raj");
  s.push("Ram");
  s.push("Rao");
  s.pop();
Iterator<String> i = s.iterator();
while(i.hasNext()){
  S.o.pln(i.next());
}}}
```



O/P:

Raj
Ram

d) LinkedHeist

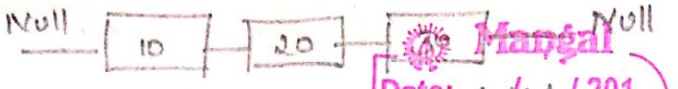


fig :- doubly linked list

* LinkedHeist implements the collection interface
* It use a doubly linked list internally to store the elements → use can add or removed elements from both sides
* It can store the duplicate elements. It maintains the insertion order.
* It is not synchronized.
* In LinkedList, the manipulation is fast because no shifting is required.

Example:

```
import java.util.*;
public class Test{
public static void main(String args[]){
LinkedList <String> l = new LinkedHeist<String>();
l.add("Raj");
l.add("Ram");
l.add("Rao");
l.add("Raj");
Iterator <String> i = l.iterator();
while(i.hasNext()){
S.o.pln(i.next());
}}
```

```
l.remove("Rao");
or
l.remove(2);   // index
LinkedList<String> l2 = new LinkedList<String>();
l2.add("Ravi");
l.addAll(l2);
l.removeAll(l2);
```

O/P

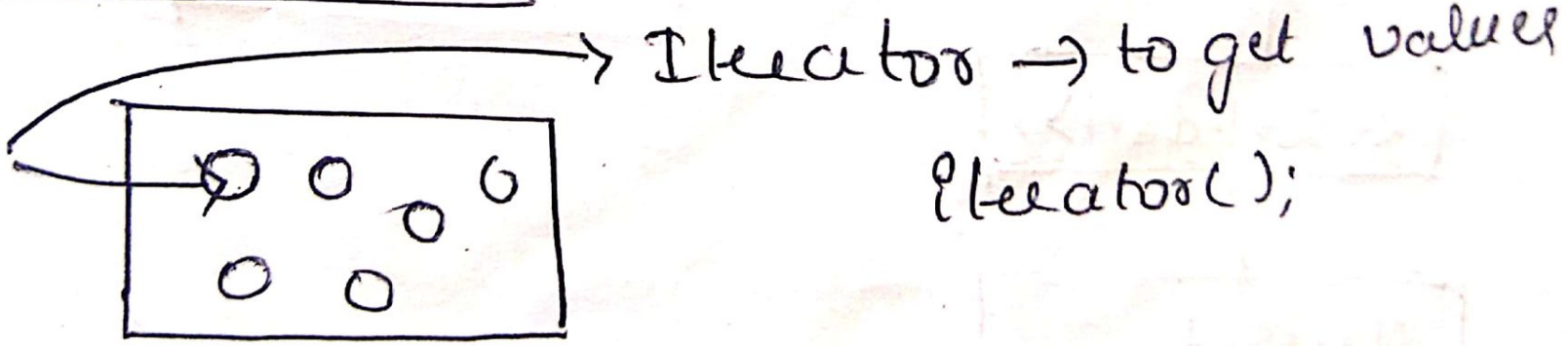
Raj
Ram
Rao
Raj

→ updating
→ removing

```
                = ar.add(d1);
boolean b2      = ar.add(d2);
S.opln(ar);

ar.trimToSize();
  S.o.p(ar.size());
}}
```

<<>>

(2) Set Interface



Mangal
Date: / /201

S1 = {A, E, I, O, U}

↳ set of vowels

S1 = {A, E, I, O, U, A}

↳ A will be removed, duplicate are not allowed

↓

so no index

↓

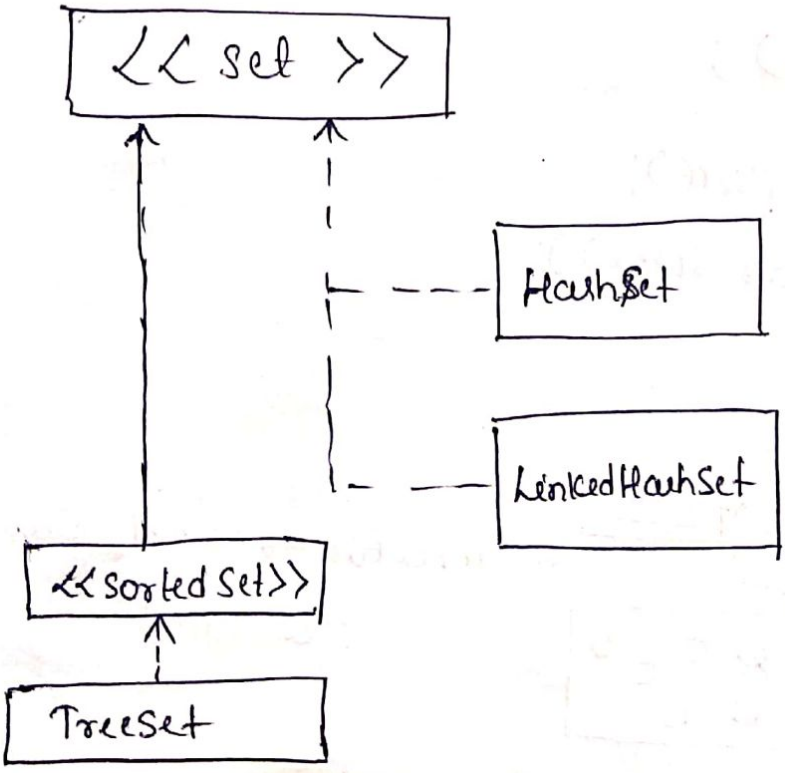
data itself unique

S1 = {A, E, U, O, I}

↳ order doesn't matter.

Properties of Set

* Set do not allow Duplicates.
* Set do not have Index.
* Set allows only one null value.
* Set do not preserve insertion order.



Set can be instantiated as:

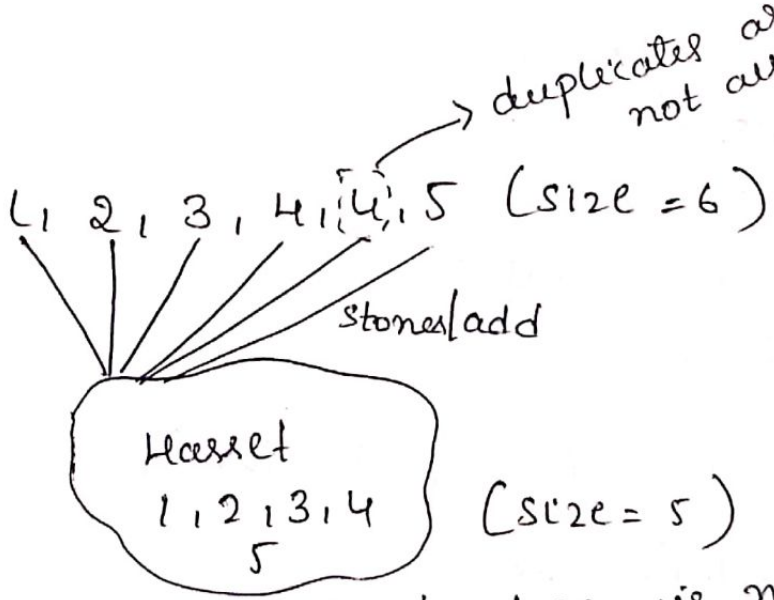
Mangal
Date: / /201

```
Set <datatype> S1 = new HashSet <datatype> ();
Set <datatype> S2 = new LinkedHashSet <datatype>();
Set <datatype> S3 = new TreeSet < datatype > ();
```

1) HashSet

* It is the concrete class of the Set Interface
* Hashset do not allow duplicates.
* Hashset allows only one null value.
* Hashset do not preserve insertion order.
* It represents the collection that uses a Hash table for storage.
* Hashing is used to store the elements in the HashSet.

Notes



* An object of Hashset class is responsible to creating Hashtable (legacy) in memory.
  → HashMap (Collection)
  ↓
  Array of LinkedList

Note:

In List, the object of LinkedList is responsible to create the Arrays in the memory (i.e dynamic Array).

Example:

```
import java.util.HashSet;
Import java.util.Iterator;

Class Mainclass
{public static void main (String[] args)
  {
    System.out.println ("program starts...");
    HashSet hs = new HashSet();

    hs.add(10);
    hs.add(null);
    hs.add(20);
    hs.add(10);
    hs.add(null);
    hs.add(30);
```

output:

```
program starts...
null
20          ← no insertion order
10
30
program ends...
```

* The underlying data structure of hashset is hashtable
* The initial capacity of HashSet is 16
* The load factor of HashSet is 0.75.
* Both loadfactor and size can be varied.

Note:

for int
100/5 = 20 => 2+0 = 2.
size

200/5 = 40 => 4+0 = 4
↳ size → Hashing technique.

for string

ABC
XY2
PQR

PQR	XY2	ABC
0	1	2

(65+66+67)/3 = 66/3 = 22/3 7/3 =

Constructor

① HashSet()
↳ constructs a new empty HashSet with default initial capacity (16) and load factor (0.75).

② HashSet (int initialCapacity) → no. of bucket

↳ Constract an empty HashSet with specified initial capacity and default load factor 0.75.

③ HashSet (int initialCapacity, float loadfactor)

↳ Constract a new empty HashSet with specified initial capacity and load-factor.

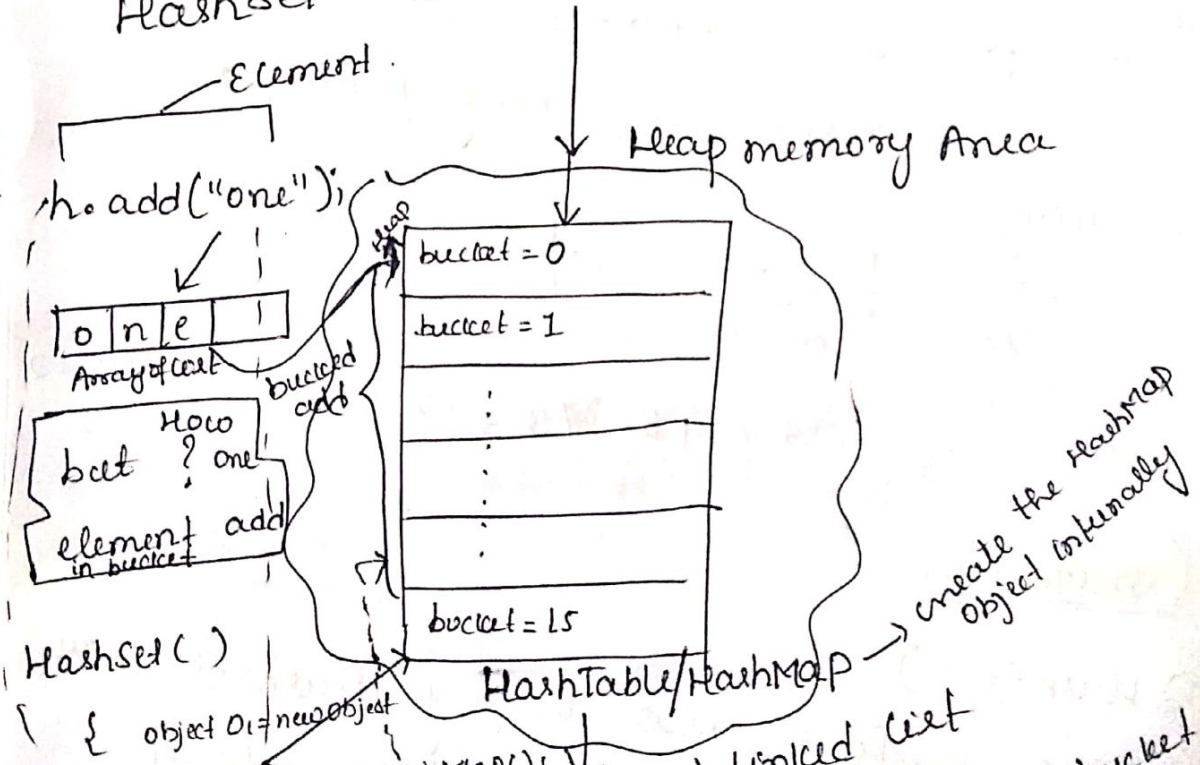
④ HashSet (Collection c)

↳ Constract a new HashSet containing the elements in the given collection.

Note:

→ constructor call

HashSet h = new HashSet();



Step 1:

HashSet h = new HashSet();

Mangal
Date: / /201

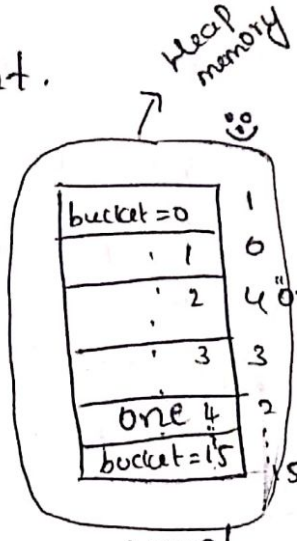
Step 2:

Create the Hashtable when we create the HashSet object.

Step 3:

In HashSet if we add any element.

h.add("one")
↳ object



* Step 4:

How elements are added in HashSet

* If we try to add any element, first it will find the bucket index. How it will find the bucket index?

↳ h.add("one")

↳ It will find the object valu

i.e add(object) ↳ "one" (add() method inter nally obj)

↳ It will find the object Hashvalue

↓

to find Hashvalue

↓

Hashcode(); called

↓

returns int value

← Hashvalue

↓

Hexa

bucketIndex

Ex: 100 % 16 = 4 → remainder is index

↓ ↳ Total no. of buckets.

Hashvalue ↓

then it

Scanned by CamScanner

Note:

Mangal Date: / /201

⊛ HashSet h = new HashSet(20);

↓

We can have more bucket

⊛ When no. of elements more than the no. of buckets → Hashtable will recreate

i.e elements → 25 . buckets → 16

i.e based on load factor.

Note:

int ↗ float ↗

* HasSet h = new HashSet (20, 0.95);

bucket load factor

* Steps: before adding the element, it check the existance of elements in HashTable

↳ It checks whether duplicate elements are present.

h.all("one")

↓ calls (internally)

h.contains( ) → boolean

↓ calls (internally)

h.equals( ) → boolean

In order to check the existance of the element.

Step 6:

Mangal Date: / /201

HashSet h = new HashSet();

Class HashSet
{
Object obj;
HashMap m;

HasSet()
{
m = new HashMap(); // by default it will create
obj = new Object();
}

key and value pairs elements are stored

Public void add(object o)
{
m.put(obj, obj)
↓ ↓
one value
↓
key
}
}

class Mainclass
{ p.s.v.M(String[] args)
h.add("one").
}

ogram

Q. How Hashset eliminate duplicates. **

* Whenever the add() method of Hashset is called it will internally call equals method and compare the Hashcode value with other Hashcode values in the Hashset.

* If the Hashcode value of the given object is same as any other object in the Hashset then it will be considered as duplicates and it won't be added to the Hashset.

Iterator :-

It is a cursor for any given collection which will be pointing to the elements present in the given collection.

Methods of Iterator:

1. hasNext() → this method returns true if there is next element available in the collection else false.
2. next() → This method will return the object pointed by the Iterator.
3. Remove() → This method delete the element pointed by the iterator in the given collection.

* we get the Iterator(cursor) on any given collection by calling Iterator method.
* Iterator() → This method returns the cursor on the given collection.

b) LinkedHashSet:

Mangal
Date: / /201

* It extends the HashSet class and implements Set interface.
* LinkedHashSet do not allow duplicates.
* It do not have index
* It allows only one null value.
* It preserves Insertion order
* the underlying datastructure LinkedHashset are Hashtable and LinkedList.

Example:

Note: Constructor for LinkedHashset
① HashSet()
② HashSet(Collection c)
③ LinkedHashSet(int capacity)
④ LinkedHashSet(int capacity, float loadfactor).

```
import java.util.*;
public class Test {
public static void main(String args[])
 {
LinkedHashSet<String> s = new LinkedHashSet
                          <String>();
 s.add("Ravi");
 s.add("Vijay");
 s.add("Ravi");
 s.add("Ajay");
Iterator<String> i = s.iterator();
while(i.hasNext()){
  s.o.pln(i.next()); }}}
```

O/P:

Ravi
Vijay
Ajay

SortedSet Interface

* SortedSet is the alternate of set interface that provides a total ordering on its elements.

Mangal Date: / /201

* The elements of the SortedSet are arranged in the increasing (ascending) order.
* It provides the additional methods that inhibit the natural ordering of the elements.

The Sorted can be instantiated as:

SortedSet<datatype> set = new TreeSet();

a) Tree Set Class :- underlying datastructure is Binary Tree

Tree Datastructure:

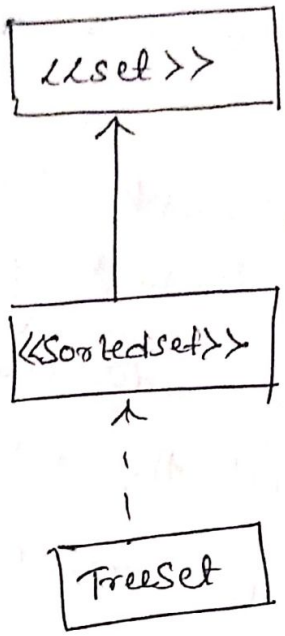
* A tree is Non-Linear datastructure where data is stored according to some relations between the data.
* A tree is group of nodes and links where the node contains the data and address of other nodes and the link represents the relation b/w given two nodes.

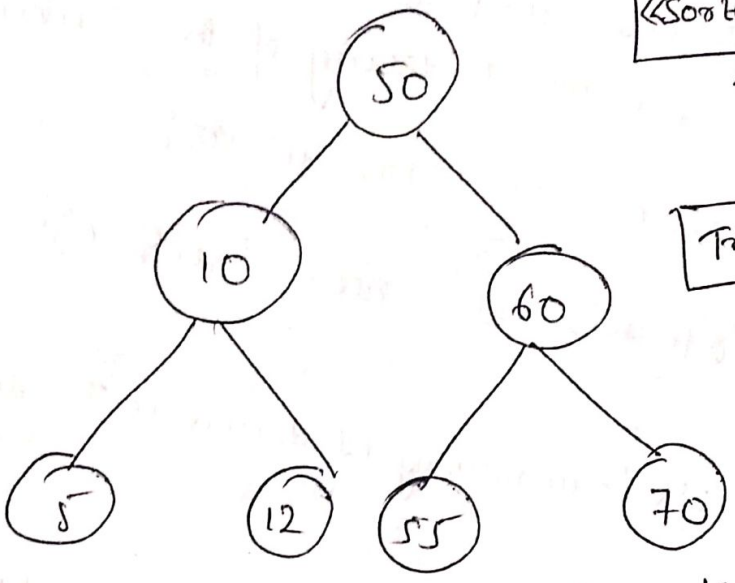
Binary tree:

It is a type of tree where the parent node can have maximum of two child nodes.

The data from the tree can be retrieved in three different order.

① pre-order.
② in-order.
③ post-order.





→ sorted order

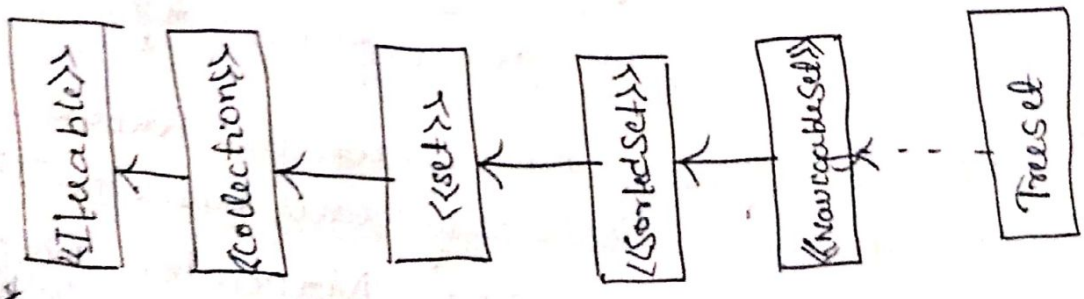
pre-order:	in-order:	post-order
parent - left - right.	left - parent - Right	left-right-parent
50	5	5
10	10	12
5	12	10
12	50	55
60	55	70
55	60	60
70	70	50

* The access and retrieval time of Treeset is quite fast.
* The elements in Treeset stored in ascending order.

* TreeSet do not allow duplicates.
* TreeSet doesn't have index.
* TreeSet do not allow null values (because null value can't used for comparision)
* TreeSet do not preserve insertion order.
* The underlying datastructure of the treeset is BinaryTree → (parent node can have maxm of 2 child nodes)
* The data of the tree set will be always sorted or ordered.
* The data of tree set can be stored in two ways
  1. natural ordering.
  2. customized ordering.

TreeSet

* It is one container object and going to store some other object (Homogenous objects)
* It holds only Homogeneous object



*

* from 1.6 it directly Implements <<NavigableSet>>
** It provide sorting technique.
** Not allowes hetrogeneous objects.
* To sort the elements internally it use Comparable Interface → which has CompareTo method and it will compare homogeneous object only.

Mangal
Date: / /201

Example:

```
import java.util.TreeSet;
class sample{
public static void main(String[] args)
 {

TreeSet ts = new TreeSet();
   ts.add(35);
   ts.add(5);
   ts.add(50);
   ts.add(25);
   ts.add(45);
   ts.add(15);
/* ts.add(40);
   ts.add("ram"); Heterogeneous data, internally
```