LIFO FIFO

→ addFirst(Object o)
→ addLast(Object o)
→ object getFirst();
→ object getLast();
Object removeFirst();
Object removeLast();

only for linkedlist objects

→ There is no
l.add("durga");
l.add(30);
l.add(null);
l.set(0, "software");
l.set(0, "venky");
l.removeLast();
l.addFirst("ccc");
S.O.P(l);

→ Linkedlist does not support random access.

Vector:- Resizable array/Growable Array
→ Duplicates.
→ Insertion order, Heterogenous objects are allowed.
→ Serializable and Cloneable, Random access
→ Synchronized, Best choice if the frequany operation retrieval

→ addElement(Object o) for Vector;
→ removeElement(Object o)
RemoveElementAt(int index)
removeAllElements()
→ Those are older, so names also lengthy.

→ object elementAt(int index)
object firstElement() → Vector
object lastElement() → fromVector
Enumeration elements()

Size:- capacity:-

Vector v=new
→ with default initial capacity
reaches maximum capacity. Bigger one will
Create and copy all and add.
newcapacity= 2 * current capacity
Vector v=new Vector(int initialCapacity);
3) Vector v=new Vector(int initialCapacity, int incremental capacity)
(1000,5) → 1005 → 1010
4) Vector v=new Vector(Collection c)

Stack:-

Stack s=new Stack();

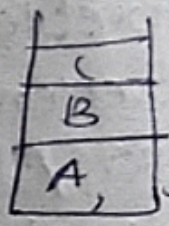
1) object push(object obj);
For inserting an object to the stack.

2) object pop():-
To removes and returns top of the stack.

3) object peak():-
To returns the top of the stack without removal of object.

4) int search(object obj)
→ If the specified object is available it returns its offset from top of the stack.
→ If the object is not available it returns -1.

5) object pop():-
For inserting an object to the stack.



S.O.P(s.Search("A")); → 3

3-cursors of java:-

If we want to retrieve objects one by one from the Collection, then we should go for cursors.
There are 3-types of cursors are available in java. → Enumeration
→ Iterator
→ List Iterator.

Vector object

2. public object nextElement();

```
public static void main(String[] args){
  Vector v=new Vector();
  for(int i=0;i<=10;i++){
    v.addElement(i);
  }
  S.o.P(v);
  Enumeration e=v.elements();
  while(e.hasMoreElements()){
    Integer I=(Integer)e.nextElement();
    if(I%2==0)
    S.o.P(I);}
  S.o.P(v);
  }
}
```

Limitations of enumerator:-

→ only used for legacy
→ only read access, no remove function

Iterator:- Implemented in 1.2 version.
We can apply iterator concept for any collection object hence it is universal cursor.
By using iterator we can perform both read and remove operations.
→ We can create iterator object by using iterator() method of Collection interface.

```
public Iterator iterator();
Iterator it=c.iterator();
```

↓ any collection object.

Methods in iterator:-

i) public boolean hasNext()
ii) public Object next()
iii) public void remove()

```
public static void main(
  ArrayList l=new ArrayList();
  for(int i=0; i<=10; i++) {
      l.add(i);
  }
  S.O.P(l); [0,1,2,--10]
  Iterator it=l.iterator();
  while(it.hasNext()) {
    Integer I=(Integer)it.next();
    if(I%2==0) {
      S.O.P(I); }
    else {
      it.remove();
  }
```

Limitations of iterator:- Implemented in 1.2 version

→ By using enumeration and iterator we can move only towards forward direction and we can't move to the backward direction, and hence these are single direction cursors.

→ By using iterator we can perform only read and remove operations and we can't perform replacement of new objects.

ListIterator:-

We can create ListIterator object by using listIterator() method of list interface.

```
public ListIterator listIterator()
    ListIterator it=l.listIterator();
                    → is any list object
```

methods:-

child interface of iterator and hence all methods of iterator by default available to listIterator

It gives following 9 methods:-

forward direction:-

1. public boolean hasNext()
2. public void next()
3. public int nextIndex()

→ public void remove()

Backward direction:-

public boolean hasPrevious()
public void previous()
public int previousIndex()

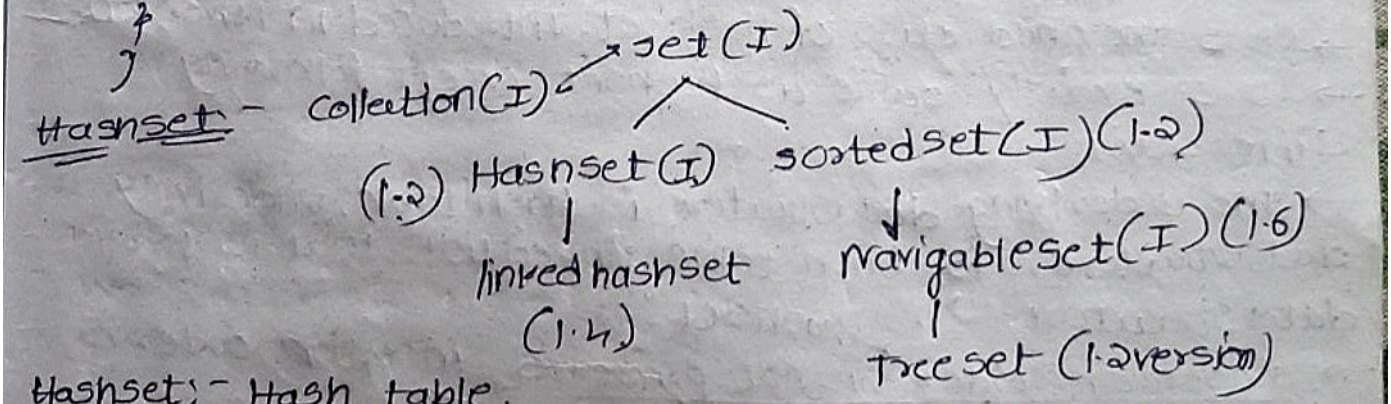
```
import ...
class ListIteratorDemo {
    public static void main(String[] args) {
        LinkedList l = new LinkedList();
        l.add("balakrishna");
        l.add("venki");
        S.O.P(l);
        ListIterator lit = l.listIterator();
        while(lit.hasNext()) {
            String s = (String)lit.next();
            if(s.equals("venki")) {
                lit.remove();
            }
        }
```

listiterator
- It can read, remove, replace addition of new objects.

Limitations:- only used for list implemented class objects and it is not a universal cursor.

How we can create an object for interface?

```
Enumeration e = v.elements();
S.O.P(e.getClass().getName());
        → o/p:- vector$1
Iterator itr = v.iterator();
S.O.P(itr.getClass().getName()); vector$Itr
ListIterator itr = v.listIterator();
S.O.P(lit.getClass().getName()); → vector$ListItr
    }
}
```



Hashset:- Hash table.

→ The underlying data structure is hash

Constructors of hashset:-

1. HashSet h = new HashSet();
   Creates an empty hashset object with default initial capacity 16 & default fill ratio 0.75.
2. HashSet h = new HashSet(int initialCapacity);
   ~~After loading the how much factor, a new hashset object will be created, that factor is called as load factor or Fill ratio.~~

capacity & default fill ratio 0.75.

3) Hashset h = new Hashset (int initialCapacity, float loadFactor);

Creates an empty hashset object with specified initial capacity & specified load factor & fill ratio.

4) Hashset h = new Hashset (Collection c);

For inter conversion between collection objects.

Load Factor / Fill Ratio:- After loading the how much factor, a new hashset object will be created, that factor is called as load factor or fill ratio.

```
public static void main(String[] args){
    Hashset h=new Hashset();
    h.add("B");
    h.add("C");
    h.add("D");
    h.add("Z");
    h.add(null);
    h.add(10);
    S.O.P(h.add("Z"));
    S.O.P(h); }}
```

Linked hashset:-

→ It is the child class of Hashset. Introduced in 1.4 version.
→ It is exactly same as hashset except following differences.
→ The underlying datastructure is hash table. underlying datastructure is hash table, Linked list that is hybrid data structure.
→ Insertion order is not preserved, insertion order is preserved. (hashset / linkedhashset)
→ Introduced 1.2 version, 1.4 version [LinkedHashset]

```
import java.util.*;
Class HashsetDemo{
    public static void main(String[] args){
        LinkedHashset h= new LinkedHashset();
        h.add("B");
        h.add("C");
        h.add("D");
        S.O.P(h.add("Z"));
```

→ It is child interface of set.
→ If we want to represent a group of individual objects according to some sorting order and duplicates are not allowed then we should go for sortedset.
→ object first() - returns first element of the sortedset.
→ object last() - returns last element of sortedset.
- sortedset headset(object obj)
  returns the sortedset whose elements are < obj.
- sortedset tailset(object obj)
  returns the sortedset whose elements are >= obj
→ sortedset subset(object obj1, object obj2)
  → returns the sortedset whose elements are
  => obj1 and < obj2.
→ comparator: returns comparator object that describes underlying sorting technique. If we are using default natural sorting order then we will get null

first() → 100
last() → 115
headset(104) → [100, 101, 103]
tailset(104) → [104, 107, 110, 115]
subset(103, 110) → [103, 104, 107]
comparator() → null

{100, 101, 103, 104, 107, 110, 115}

→ default natural sorting order for numbers Ascending order and for string alphabetical order.
→ we can apply the above methods only on sortedset implemented class objects. That is on the treeset object.

The underlying data structure is Tree.
Duplicate objects are not allowed.
Insertion order not preserved, but all objects will be inserted according to some sorting order.
Heterogenous objects are not allowed. If we are trying to insert heterogenous objects then we will runtime exception saying ClassCastException.
→ Null insertion is allowed, but only once.
→ Treeset t=new Treeset();
Creates a empty Treeset object where elements will be inserted according to default natural sorting order.

2. Treeset t=new Treeset (Comparator c);
Creates an empty Tree set object where elements will be inserted according to customized sorting order.
→ Treeset t=new Treeset (SortedSet s);
Treeset t=new Treeset (Collection c);

Create database palletraining
use database palletraining
Create table product(
pid int,
pname varchar(40)
Cost int,
Manufatured-date date
);
insert into palletraining values(1, 'UX', 34, 'HU', dec-12-20)
(2, 'locks', 1200, 'Go');
selet * from product

insert into employee values
(4, 'ravi', 36000, 'o+ve', 48, 'ravi@gmail.com'),
(6, 'suresh', 38000, 'o+ve', 56, null)

Create database expe
Use database xpa.
Create table employee(
eid int primary key,
name varchar(40)
salary int not NULL.
bg varchar(10) default o+ve,
age int, check(age between 18 and 60)
email varchar(20) unique
);

```
t.add("A");
t.add("B");
t.add("C");   ⇒ [A, B, L, Z, a]
t.add("L");
t.add("a");
s.o.p(t);
```

→ For empty treeset as the first element is null insertion is possible. But after inserting that null if we are trying to insert any another element we will get Nullpointer exception.

→ For non-empty treeset if we are trying to insert null then we will get NullpointerException.

```
import java.util.Treeset;
class TreeSetDemo {
  public static void main(String[] args) {
    Treeset t=new Treeset();
    t.add(new StringBuffer("A"));
    t.add(new StringBuffer("Z"));
    t.add(new StringBuffer("L"));
    t.add(new StringBuffer("a"));
    s.o.p(t);   } → we got ClassCastException.
  }
}
```

→ If we are depending on default natural sorting order then objects should be homogenous and comparable. otherwise we will get runtime exception saying ClassCastException.

→ An object is said to be comparable if and only if the corresponding class implements java.lang.Comparable interface.

→ String class and all wrapper classes already implements Comparable interface. But StringBuffer doesn't comparable interface.

Comparable Interface:-

This interface present in java.lang package it contains only one method. CompareTo().

```
public int CompareTo(Object obj)
```

Ex:- obj1.CompareTo(obj2)

→ returns -ve if obj1 has to come before obj2

internally JVM will call compareTo() method for inserting objects to the TreeSet. Hence the objects should be Comparable.

```
TreeSet t = new TreeSet();
t.add("B");
t.add("Z");
t.add("A");
S.O.P(t);
```

If we are not satisfied with default natural sorting order or if the default natural sorting order is not already available then we can define our own customized sorting by using Comparator.

Comparable meant for default natural sorting order where as comparator meant customized sorting order.

→ program for insert integer objects into TreeSet where the sorting order is descending order.

```
import java.util.*;
class TreeSetDemo3 {
  public static void main(String[] args) {
    TreeSet t = new TreeSet(new MyComparator());
    t.add(10);
    t.add(0);
    t.add(15);
    t.add(20);
    S.O.P(t);
  }
}
```

```
class MyComparator implements
      Comparator {
  public int compare(Object obj1,
        Object obj2) {
    Integer I1 = (Integer)obj1;
    Integer I2 = (Integer)obj2;
    if (I1<I2)
      return +1;
    else if (I1>I2)
      return -1;
    else
      return 0;
  }
}
```

At line-1 if we are not passing comparator object then internally JVM will call compareTO() method which meant for default natural sorting order (ascending order).

→ [0, 10, 15, 20].

→ If we are passing Comparator object at line 1 then

[20,15,10,0]

class

```
import java.util.*;
class TreeSetDemo2{
public static void main(String[] args){
TreeSet t=new TreeSet(new MyComparator());
t.add("Roja");
t.add("Snehapani");
t.add("Rajakumari");
t.add("Ramulamma");
S.O.P(t);
}  class MyComparator implements Comparator{
}                public int Compare(Object obj1, Object
                      obj2){
                    String s1=obj1.toString();
                    String s2=(String)obj2;
                    return -s1.compareTo(s2);
                                    }}
```

→ return 11 compareTo(12);

[0,10,15,20] ascending

→ [20,15,10,0] descending order → return -11 compareTo(12);

→ [20,15,10,0] descending order → return 12 compareTo(11);

→ [0,10,15,20] ascending → return -12 compareTo(11);

→ [10,0,15,20,20] Insertion order → return +1;

[20,20,15,0,10] reserverse → return -1;

[10] → return 0;

```
import java.util.*;
class TreesetDemo10
{
    public static void main(String[] args)
    {
    Treeset t = new Treeset(new MyComparator());
    t.add(new StringBuffer("A"));
    t.add(new StringBuffer("Z"));
    t.add(new StringBuffer("K"));
    t.add(new StringBuffer("L"));
    System.out.println(t);
    }
}
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s1.compareTo(s2);
    }
}
```

- If we are defining our own sorting by comparator the objects need not be comparable.

Difference between Comparable and Comparator

Comparable	Comparator
It is meant for default natural sorting order.	It is meant for customized sorting order.
present in java.lang. package.	present in java.util. package.
This interface defines only one method compareTo().	This interface defines two methods compare and equals()
All wrapper classes and String class implement Comparable interface.	The only implemented classes of comparator are collator and rulebased collator.

Hashset	LinkedHashset	TreeSet
Hashtable	Hashtable + LinkedList	Balanced Tree
Not preserved order	preserved	Not applicable
sorting order not applicable.	Not applicable	applicable
heterogenous objects allowed	allowed	not allowed
Duplicates objects not allowed	not allowed	not allowed.
Null acceptance Allowed only once	Allowed only once	For empty TreeSet as firstElement null is allowed and in all other cases we will get NullPointerException.

Disadvantages of Collections:-

1. Type saftely:- collections in pre-Java 5 without generics stored objects of any type. This allowed the mixing of different object types, which could lead to runtime errors if the wrong type was retrieved from a collection.

2. Casting overhead:- since collections did not enforce type saftey, objects retrieved from collections had to be cast explicitly to their original type. This increased the likelihood of ClassCastException errors at runtime and will introduce unnecessary complexity.

3. Runtime errors:- Error related to incorrect types were only caught during runtime, making de-bugging more difficult and reducing code reliablity.

How generics overcame:-

Type saftey:- Generics introduced compile type checking. This means that when you define a collection using generics (e.g List<String>) the collection

Will catch the error early.

No need for Casting:- with generics you no longer need to cast objects when retrieving them from a Collection. The type is known at compile-time, so the code is cleaner and free from class CastException errors.

→ Error will be detected at compile time.

compile time.

These errors occurs are detected by the compiler when you try to compile the code.

They occur due to syntax issues or problems in the structure of the code.

→ Eg:- missing semicolons, unmatched braces.

→ Incompatible method calls (e.g., calling a method with the wrong number or type of parameters).

→ Errors caught at compile time can be fixed before running the program, which increases reliability.

→ compile-time errors help developers identify issues early in the development process.

Run-time

These errors occur while the program is running after it has been successfully compiled.

They are generally caused by logic errors or unforeseen circumstances during execution.

Ex:- Divide by zero

Accessing an array index that is out of bounds

→ Runtime errors are harder to detect and fix because they occur while the program is running, which can lead to crashes or unpredictable behavior.

How to pass or store Collection in class?

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
```

```
        int Student Number, String
            Student Name) {
    this.StudentNumber = StudentNumber;
    this.studentName = StudentName; }
public int getStudentNumber() {
    return studentNumber;
}
public String getStudentName() {
    return studentName; }
public String toString() {
    return "student Number:" + studentNumber +
        ", Student Name:" + studentName;
}
}
public class sample {
    public static void main (String[] args) {
        List<Student> students = new ArrayList<>();
        Scanner sc=new Scanner (System.in);
        String response;
        S.o.p("Do you want to store a new object?
            yes/no");
        response = sc.next();
        while (response.equalsIgnoreCase ("yes")) {
            S.o.p("Enter the student number");
            int num=sc.nextInt();
            S.o.p("Enter the student name:");
            sc.nextLine();
            String name = sc.nextLine();
            Student student = new Student (num,name);
            students.add (student);
            S.o.p ("Do you want to add another
            Student: yes/no");
            response= sc.next(); }
    S.o.p ("List of students");
    for (Student S: students) {
```

3. List <Integer> i = new ArrayList <Integer>();
4. List <Integer> i = new ArrayList <String>();
5. ArrayList <Integer> i = new ArrayList <Integer>();

1) List <Student> students = new ArrayList <>();

→ This is correct and most commonly used.
→ <> Generics (diamond operator) tells the compiler to infer the type student in this case from the left hand side.
→ Type safety:- only accepts Students objects ensuring type saftey at compile time.
→ Interface vs Implementation:-
→ Here list is an interface whereas Arraylist is class that implements it.
→ using interface (List) allows for flexiblity in changing the underlying implementation later if needed to switch Linked List without changing code.
→ This one is the correct, modern, and prefered approach for defining a list.
→ Type inference:- The compiler can infer the type of ArrayList from the declaration on the left side so there is no need to repeat student in right hand side.

List <Student> s = new ArrayList <Student>();

→ so this not a new version, updated version is above one.

↑ there is no mistake, but compiler will automatically take, then we should use this style time

→ List <Integer> i = new ArrayList <Integer>();

↓

This one is correct but you can use diamond operator (<>) to avoid repeating of type of integer.

→ List <Integer> i = new ArrayList <String>();

↓ incorrect

→ ArrayList <Integer> i = new ArrayList <Integer>();

Current

Java ArrayList: we must create the object because arraylist is class in ...

ArrayList class is a resizeable array, which can be found in the java.util package.

The difference between a built in array and ArrayList in java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want. the syntax is slightly different.

```
import java.util.ArrayList;
ArrayList<String> cars=new ArrayList<String>();
```

ArrayList<Integer> Integer

Add items:- Arraylist al=new ArrayList();

```
import java.util.ArrayList;
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars=new ArrayList
        <String>();
    cars.add("volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("mazda");
    System.out.println(cars);
  }
}
```

→ Can have duplicate elements also

→ It implements list interface so we can use all list interface

→ It contains insertion order internally.

Access an item:-

To access an element in the ArrayList, use get() method refer to the index number.

```
cars.get(0);
```

→ It is non-synchronized

→ It will allow random access because the array works on index basis.

Modify:- set()

```
cars.set(0,"opel");
```

cars.remove(0);

To remove all the elements

cars.clear();

ArrayList size():-

cars.size();

Loop through an ArrayList:-

```
public class Main{
    public static void main(String[] args){
    ArrayList<String> cars=new ArrayList<String>
                ();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Mazda");
    cars.add("Ford");
    for(int i=0;i<cars.size();i++){
        System.out.println(cars.get(i));
        }
    }
}
```

It will add new element to that particular collection

→ In arraylist manipulation is little slower than linked list, because lot of shifting needs to occur if any element is removed from the arraylist.

For each loop:-

```
public class Main{
    public static void main(String[] args){
    ArrayList<String> cars=new ArrayList<String>
                ();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for(String i: cars){
        System.out.println(i);
        }
```

→ The size is dynamic in the arraylist, which varies according to the elements getting added or removed.

Elements in arrayList are
the example above, we created elements (objects)
of type "String". Remember that a String in Java
is an object (not a primitive type). To use other
types, such as int, you must specify an equivalent
wrapped class: Integer. For other primitive types,
use Boolean for boolean, character for char

```
import java.util.ArrayList;
    public class Main {
      public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList
          <Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
     for (int i : myNumbers) {
        System.out.println(i);
      }
    }
}
```

Sort An ArrayList

Another useful class in the java.util package
is the Collections class, which include the sort()
method for sorting lists alphabetically or numerically.

```
import java.util.ArrayList;
import java.util.Collections; Import the
  public class Main {                Collection class
    public static void main(String[] args) {
      ArrayList<String> cars = new ArrayList
            <String>();
      cars.add("Volvo");
      cars.add("BMW");
      cars.add("Ford");
```

```
        for (String i : cars) {
            System.out.println(i);
        }
    }
}
```

Sort an ArrayList of Integers

```
import java.util.ArrayList;
import java.util.Collections;
public class Main {
    public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList
                <Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);
    Collections.sort(myNumbers);
    for (int i : myNumbers) {
        System.out.println(i);
    }
}
```

Cet