# Collection

R Lawrance

# Introduction

- An array is an indexed collection of fixed no of homogeneous data elements. (or)
- An array represents a group of elements of same data type.
- The main advantage of array is we can represent huge no of elements by using single variable.
- So that readability of the code will be improved.

# Limitations of array

- Arrays are fixed in size that is once we created an array there is no chance of increasing (or) decreasing the size based on our requirement
- Hence to use arrays concept compulsory we should know the size in advance which may not possible always.
- Arrays can hold only homogeneous data elements.
- Arrays concept is not implemented based on some data structure
- Hence ready-made methods support we can't expert. For every requirement we have to write the code explicitly.

**To overcome the above limitations we should go for collection concept.**

- Collection are **growable** in nature that is based on our requirement we can increase (or) decrease the size
- Hence memory point of view collection concept is recommended to use.
- Collection can hold both homogeneous and heterogeneous objects.
- Every collection class is implemented based on some standard data structure
- Hence for every requirement ready-made method support is available
- Being a programmer we can use these methods directly without writing the functionality on our own.

# Differences between Arrays and Collections

| Arrays | Collections |
| --- | --- |
| 1) Arrays are fixed in size. | 1) Collections are growable in nature. |
| 2) Memory point of view arrays are not recommended to use. | 2) Memory point of view collections are highly recommended to use. |
| 3) Performance point of view arrays are recommended to use. | 3) Performance point of view collections are not recommended to use. |
| 4) Arrays can hold only homogeneous data type elements. | 4) Collections can hold both homogeneous and heterogeneous elements. |
| 5) There is no underlying data structure for arrays and hence there is no readymade method support. | 5) Every collection class is implemented based on some standard data structure and hence readymade method support is available. |
| 6) Arrays can hold both primitives and object types. | 6) Collections can hold only objects but not primitives. |

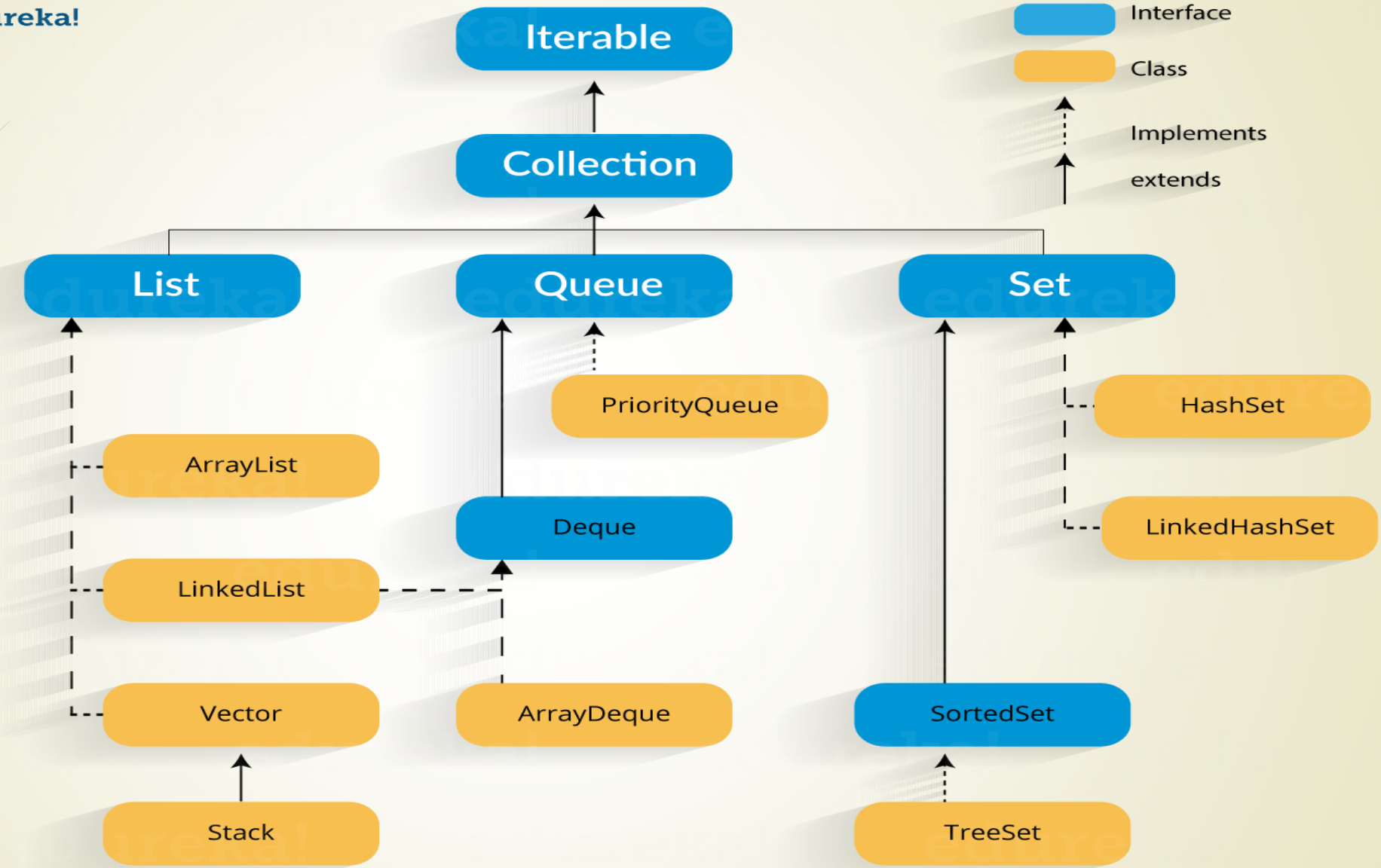
# 9(Nine) key interfaces of collection framework

- Collection
- List
- Set
- SortedSet
- NavigableSet
- Queue
- Map
- SortedMap
- NavigableMap

# Collection

- If we want to represent a group of objects as single entity then we should go for collection.
- In general we can consider collection as root interface of entire collection framework.
- Collection interface defines the most common methods which can be applicable for any collection object.
- There is no concrete class which implements Collection interface directly.

edureka!


Iterable
Collection
List
Queue
Set
Interface
Class
Implements
extends
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

# Collection interface

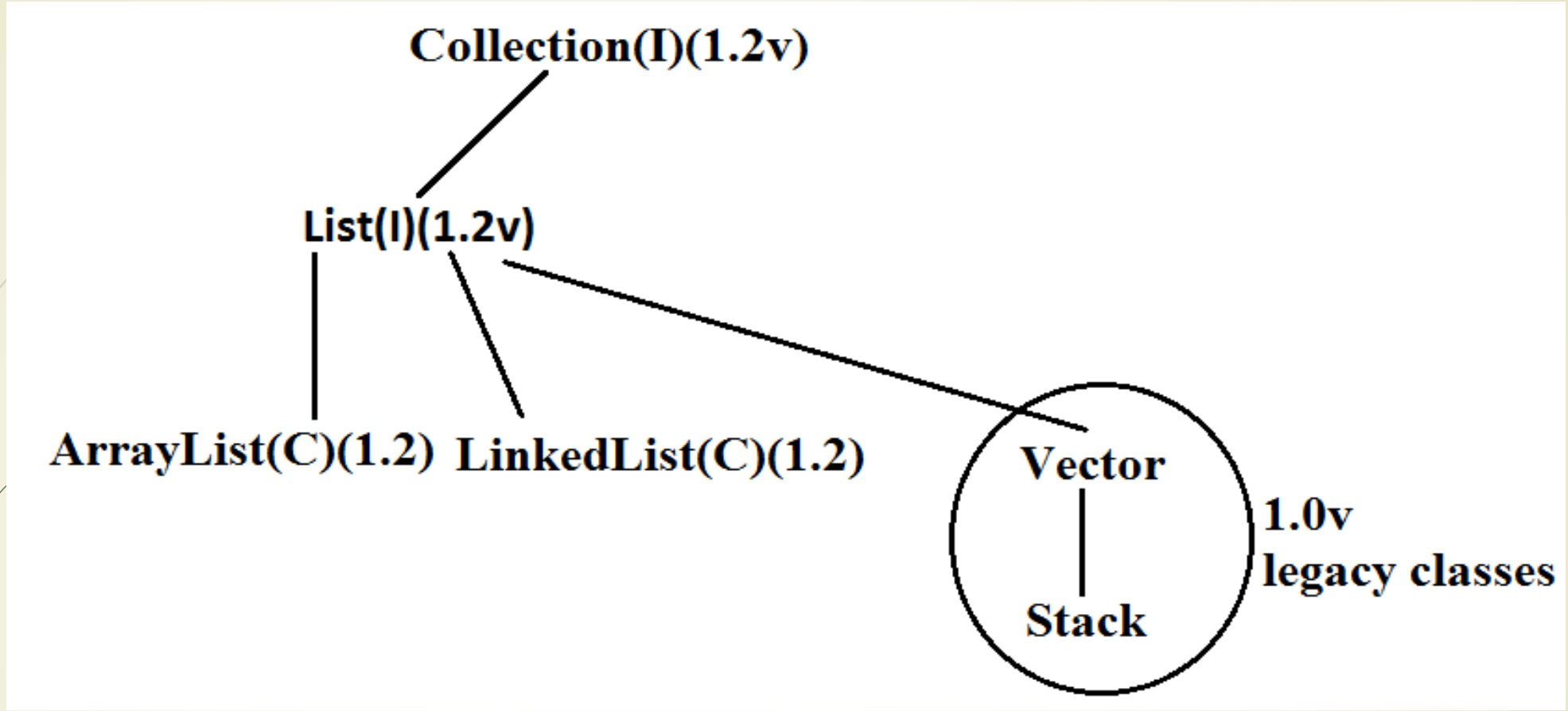
- If we want to represent a group of individual objects as a single entity then we should go for Collection interface.
- This interface defines the most common general methods which can be applicable for any Collection object.

- **The following is the list of methods present in Collection interface.**
  - boolean add(Object o);
  - boolean addAll(Collection c);
  - boolean remove(Object o);
  - boolean removeAll(Object o);
  - boolean retainAll(Collection c);
    To remove all objects except those present in c.
  - Void clear();
  - boolean contains(Object o);
  - boolean containsAll(Collection c);
  - boolean isEmpty();
  - Int size();
  - Object[] toArray();
  - Iterator iterator();

There is no concrete class which implements Collection interface directly.

# List:

- It is the child interface of Collection.
- If we want to represent a group of individual objects as a single entity where "duplicates are allowed and insertion order must be preserved" then we should go for List interface.
- We can differentiate duplicates by using index




**Vector and Stack classes are re-engineered in 1.2 versions to implement List interface**.

# List interface defines the following specific methods.

- boolean add(int index,Object o);
- boolean addAll(int index,Collectio c);
- Object get(int index);
- Object remove(int index);
- Object set(int index,Object new);//to replace
- Int indexOf(Object o);
  Returns index of first occurrence of "o".
- Int lastIndexOf(Object o);
- ListIterator listIterator();

# ArrayList:

- The underlying data structure is resizable array (or) growable array.
- Duplicate objects are allowed.
- Insertion order preserved.
- Heterogeneous objects are allowed.(except TreeSet , TreeMap every where heterogenious objects are allowed)
- Null insertion is possible.

# Constructors:

- 1) ArrayList a=new ArrayList();

  Creates an empty ArrayList object with default initial capacity "10"

  if ArrayList reaches its max capacity then a new ArrayList object will be created with

  **New capacity=(current capacity*3/2)+1**

- 2) ArrayList a=new ArrayList(int initialcapacity);

  Creates an empty ArrayList object with the specified initial capacity.

- 3) ArrayList a=new ArrayList(Collection c);

  Creates an equivalent ArrayList object for the given Collection that is this constructor meant for inter conversation between collection objects.

```
import java.util.*;
class ArrayListDemo
{    public static void main(String[] args)
     {
          ArrayList a=new ArrayList();
          a.add("A");
          a.add(10);
          a.add("A");
          a.add(null);
          System.out.println(a);//[A, 10, A, null]
          a.remove(2);
          System.out.println(a);//[A, 10, null]
          a.add(2,"m");
          a.add("n");
          System.out.println(a);//[A, 10, m, null, n]
          }
}
```

- ArrayList and Vector classes implements RandomAccess interface so that any random element we can access with the same speed.

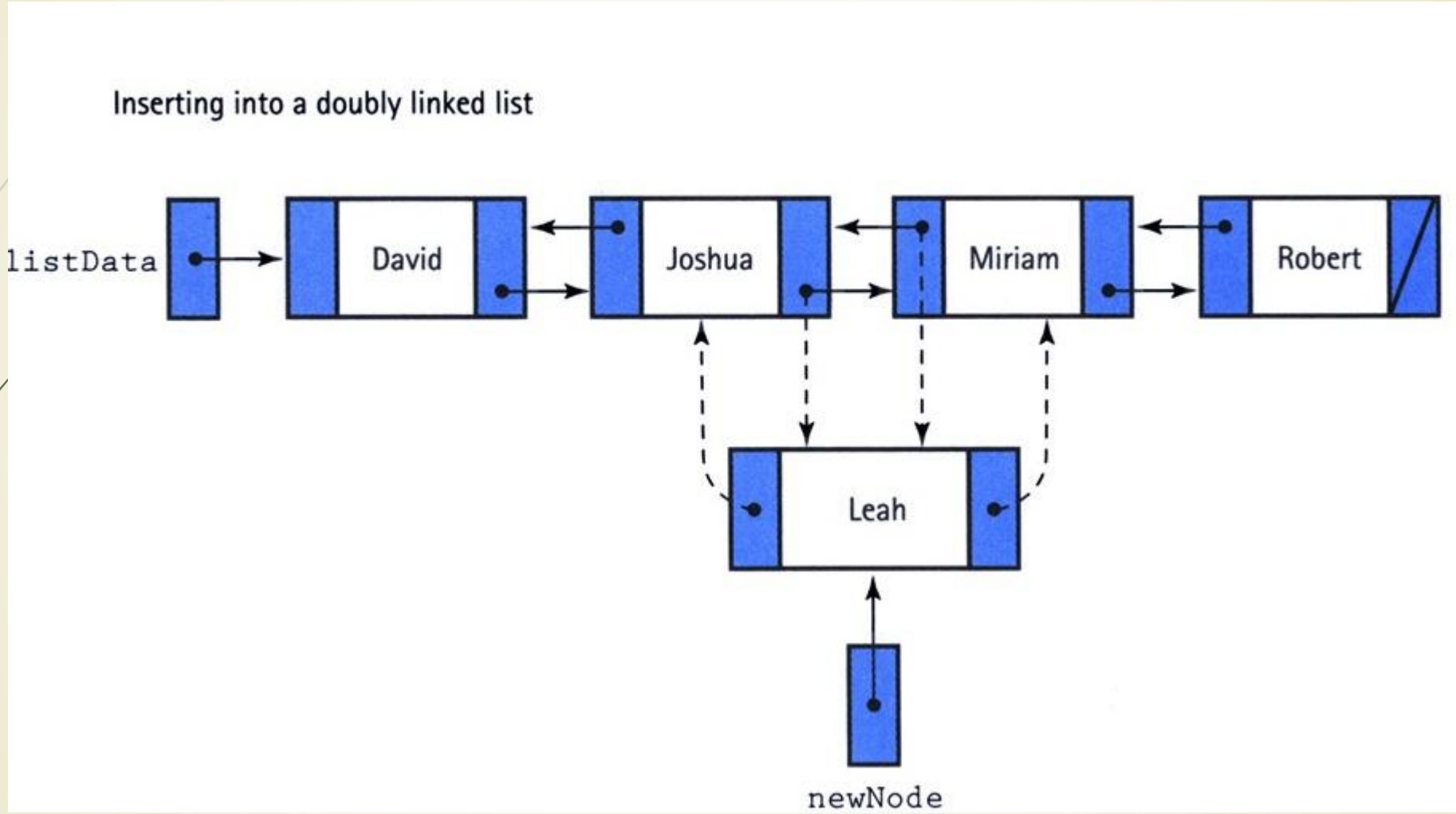
Hence ArrayList is the best choice of "retrival operation".

- RandomAccess interface present in util package and doesn't contain any methods. It is a marker interface.

- If our frequent operation is insertion or deletion in the middle ArrayList is the worst choice.

# LinkedList

- The underlying data structure is double LinkedList
- If our frequent operation is insertion (or) deletion in the middle then LinkedList is the best choice.
- If our frequent operation is retrieval operation then LinkedList is worst choice.
- Duplicate objects are allowed.
- Insertion order is preserved.
- Heterogeneous objects are allowed.
- Null insertion is possible.
- Implements Serializable and Cloneable interfaces but not RandomAccess.


Inserting into a doubly linked list
listData
David
Joshua
Miriam
Robert
Leah
newNode

- Usually we can use LinkedList to implement Stacks and Queues.

To provide support for this requirement LinkedList class defines the following 6 specific methods.

- void addFirst(Object o);
- void addLast(Object o);
- Object getFirst();
- Object getLast();
- Object removeFirst();
- Object removeLast();

# Constructors:

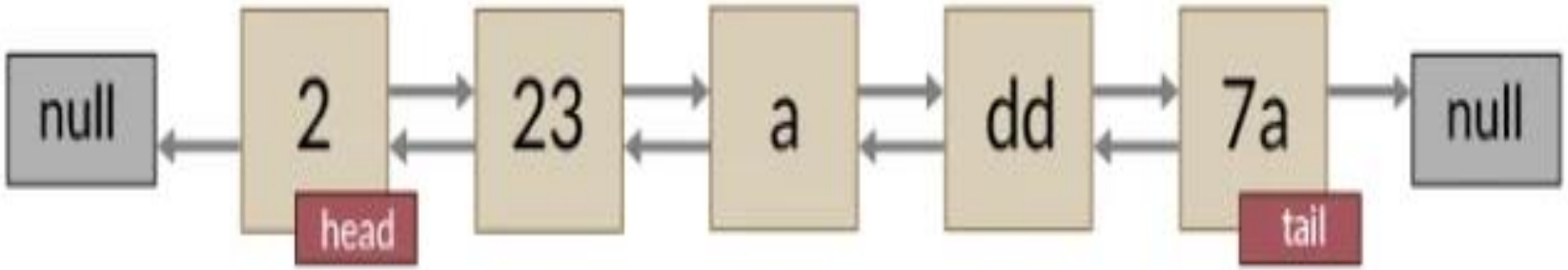
- **LinkedList l=new LinkedList();**
  Creates an empty LinkedList object.

- **LinkedList l=new LinkedList(Collection c);**
  To create an equivalent LinkedList object for the given collection.

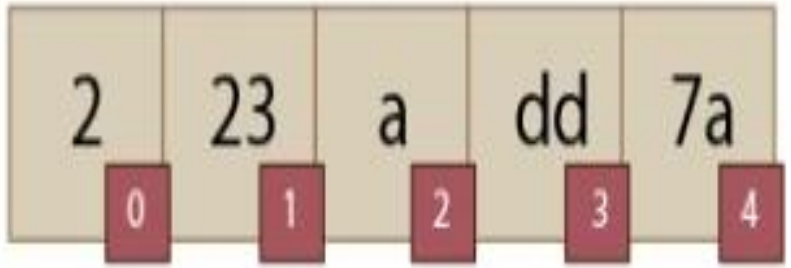
```
mport java.util.*;
class LinkedListDemo
{
	public static void main(String[] args)
	{
		LinkedList l=new LinkedList();
		l.add(“lawra");
		l.add(30);
		l.add(null);
		l.add(“lawra");
		System.out.println(l);//[lawra, 30, null, lawra]
		l.set(0,"software");
		System.out.println(l);//[software, 30, null, lawra]
		l.set(0,“Mark");
		System.out.println(l);//[Mark, 30, null, lawra]
		l.removeLast();
		System.out.println(l);//[Mark, 30, null]
		l.addFirst("vvv");
		System.out.println(l);//[vvv, Mark, 30, null]
	}
}
```

# ArrayList vs. LinkedList

## LinkedList




## Array and ArrayList



# Vector

- The underlying data structure is resizable array (or) growable array.
- Duplicate objects are allowed.
- Insertion order is preserved.
- Heterogeneous objects are allowed.
- Null insertion is possible.
- Implements Serializable, Cloneable and RandomAccess interfaces.

# Vector specific methods

**To add objects:**

- add(Object o);-----Collection
- add(int index,Object o);-----List
- addElement(Object o);-----Vector

# Vector specific methods

**To remove elements:**

- remove(Object o);-------Collection
- remove(int index);-----------List
- removeElement(Object o);----Vector
- removeElementAt(int index);-----Vector
- removeAllElements();-----Vector
- clear();-------Collection

# Vector specific methods

**To get objects:**

- Object get(int index);----------List
- Object elementAt(int index);-----Vector
- Object firstElement();----------Vector
- Object lastElement();----------Vector

# Vector specific methods

**Other methods:**

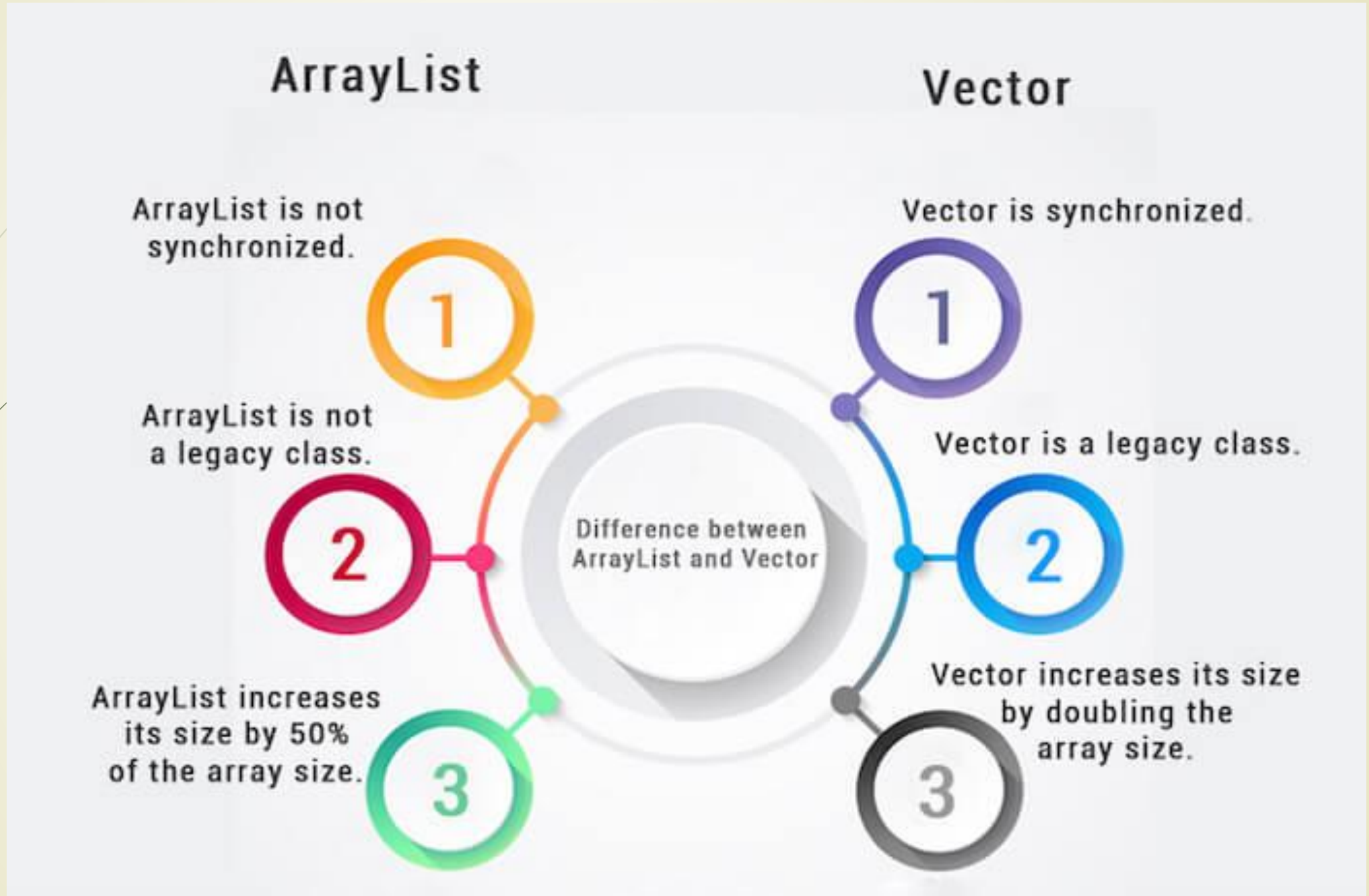
- Int size();//How many objects are added
- Int capacity();//Total capacity
- Enumeration elements();

# Constructors:

- Vector v=new Vector();
  - Creates an empty Vector object with default initial capacity 10.
  - Once Vector reaches its maximum capacity then a new Vector object will be created with double capacity. That is "**newcapacity=currentcapacity*2**".
- Vector v=new Vector(int initialcapacity);
- Vector v=new Vector(int initialcapacity, int incrementalcapacity);
- Vector v=new Vector(Collection c);

```
import java.util.*;
class VectorDemo
{
	public static void main(String[] args)
	{
		Vector v=new Vector();
		System.out.println(v.capacity());//10
		for(int i=1;i<=10;i++)
		{
			v.addElement(i);
		}
	System.out.println(v.capacity());//10
	v.addElement("A");
	System.out.println(v.capacity());//20
	System.out.println(v);//[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A]
	}
}
```

Stack:


ArrayList
Vector
ArrayList is not synchronized.
1
ArrayList is not a legacy class.
2
ArrayList increases its size by 50% of the array size.
3
Difference between ArrayList and Vector
Vector is synchronized.
1
Vector is a legacy class.
2
Vector increases its size by doubling the array size.
3

# Stack

- It is the child class of Vector.
- Whenever **last in first out(LIFO)** order required then we should go for Stack.

# Constructor

It contains only one constructor.

- **Stack s= new Stack();**

# Methods:

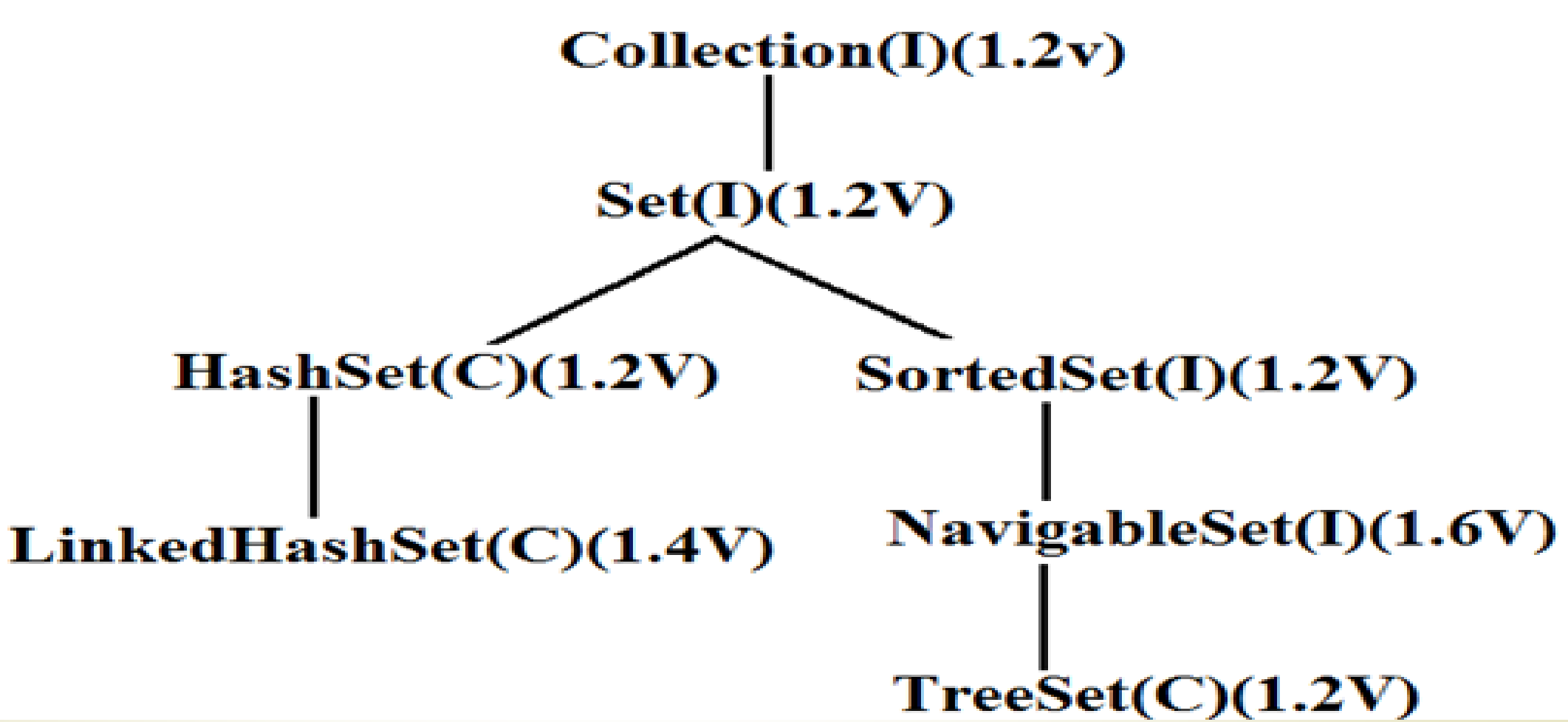
- **Object push(Object o);**
  To insert an object into the stack.
- **Object pop();**
  To remove and return top of the stack.
- **Object peek();**
  To return top of the stack without removal.
- **boolean empty();**
  Returns true if Stack is empty.
- **Int search(Object o);**
  Returns offset if the element is available otherwise returns "-1"

```
import java.util.*;
class StackDemo
{
	public static void main(String[] args)
	{
		Stack s=new Stack();
		s.push("A");
		s.push("B");
		s.push("C");
		System.out.println(s);//[A, B, C]
		System.out.println(s.pop());//C
		System.out.println(s);//[A, B]
		System.out.println(s.peek());//B
		System.out.println(s.search("A"));//2
		System.out.println(s.search("Z"));//-1
		System.out.println(s.empty());//false
	}
}
```

# Set interface

# Set Interface

- It is the child interface of Collection.
- If we want to represent a group of individual objects as a single entity where duplicates are not allowed
- insertion order is not preserved then we should go for Set interface.




Set interface does not contain any new method we have to use only Collection interface methods.

# HashSet

- The underlying data structure is Hashtable.
- Insertion order is not preserved and it is based on hash code of the objects.
- Duplicate objects are not allowed.
- If we are trying to insert duplicate objects we won't get compile time error and runtime error add() method simply returns false.
- Heterogeneous objects are allowed.
- Null insertion is possible.(only once)
- Implements Serializable and Cloneable interfaces but not RandomAccess.
- HashSet is best suitable, if our frequent operation is "Search".

# Constructors:

- HashSet h=new HashSet();

  Creates an empty HashSet object with default initial capacity 16 and default fill ratio 0.75(fill ratio is also known as load factor).

- HashSet h=new HashSet(int initialcapacity);

  Creates an empty HashSet object with the specified initial capacity and default fill ratio 0.75.

- HashSet h=new HashSet(int initialcapacity,float fillratio);

- HashSet h=new HashSet(Collection c);

- Note : After filling how much ratio new HashSet object will be created , The ratio is called "FillRatio" or "LoadFactor".

# Example

```
import java.util.*;
class HashSetDemo
{
    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);
        System.out.println(h.add("Z"));//false
        System.out.println(h);//[null, D, B, C, 10, Z]
    }
```

# LinkedHashSet:

- It is the child class of HashSet.
- LinkedHashSet is exactly same as HashSet except the following differences.

| HashSet | LinkedHashSet |
|---|---|
| 1) The underlying data structure is Hashtable. | 1) The underlying data structure is a combination of LinkedList and Hashtable. |
| 2) Insertion order is not preserved. | 2) Insertion order is preserved. |
| 3) Introduced in 1.2 v. | 3) Introduced in 1.4v. |

In the above program if we are replacing HashSet with LinkedHashSet the output is [B, C, D, Z, null, 10]. That is insertion order is preserved.

# Example

```
import java.util.*;
class LinkedHashSetDemo
{
    public static void main(String[] args)
    {
        LinkedHashSet h=new LinkedHashSet();
        h.add("B");
        h.add("C");
        h.add("D");
        h.add("Z");
        h.add(null);
        h.add(10);
        System.out.println(h.add("Z"));//false
        System.out.println(h);//[B, C, D, Z, null, 10]
        }
```

Note: LinkedHashSet and LinkedHashMap commonly used for implementing "cache applications" where insertion order must be preserved and duplicates are not allowed.

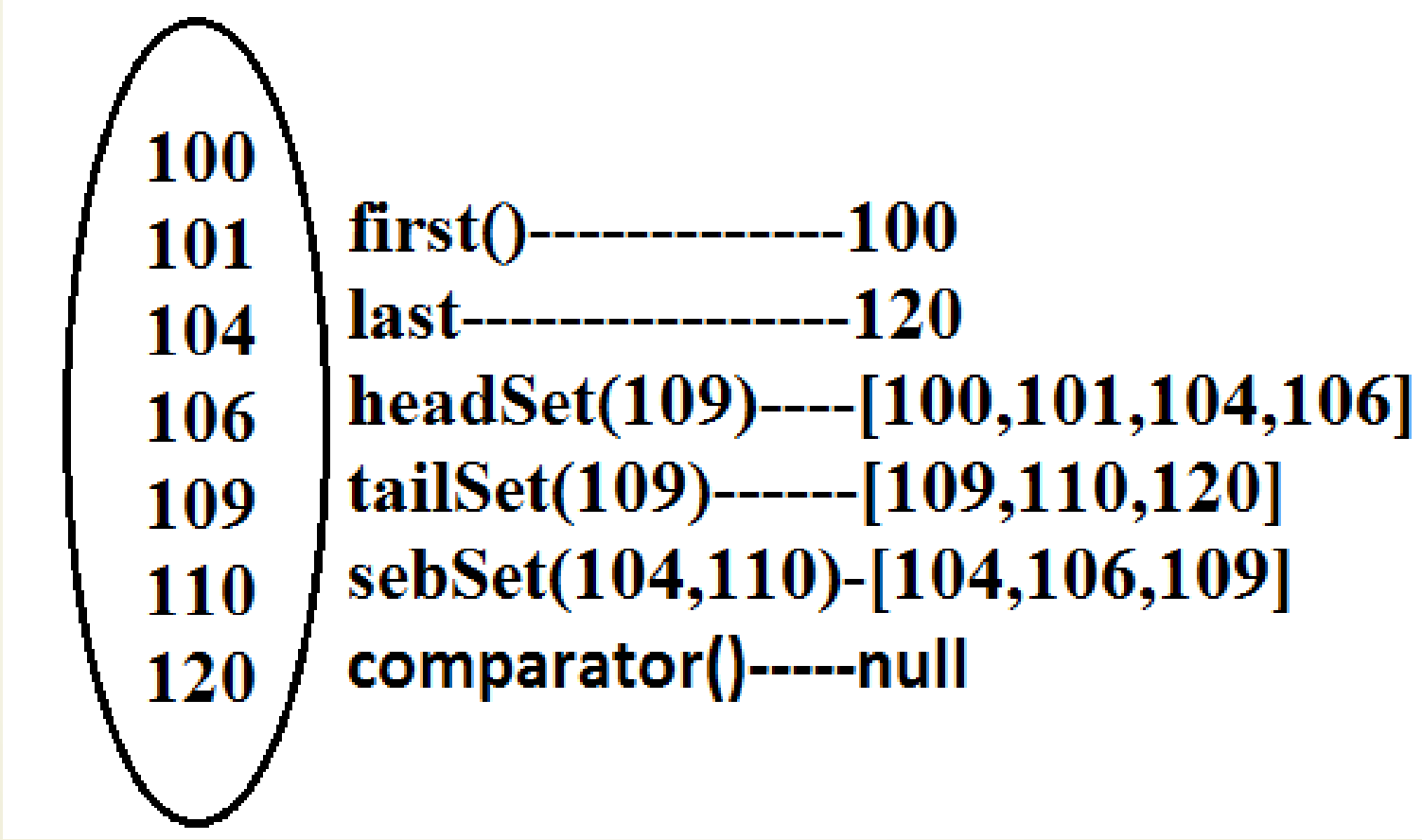
# SortedSet:

- It is child interface of Set.
- If we want to represent a group of "unique objects" where duplicates are not allowed and all objects must be inserting according to some sorting order then we should go for SortedSet interface.
- That sorting order can be either default natural sorting (or) customized sorting order.

## SortedSet interface define the following 6 specific methods.

- Object first();
- Object last();
- SortedSet headSet(Object obj);
  Returns the SortedSet whose elements are <obj.
- SortedSet tailSet(Object obj);
  It returns the SortedSet whose elements are >=obj.
- SortedSet subset(Object o1,Object o2);
  Returns the SortedSet whose elements are >=o1 but <o2.
- Comparator comparator();
  - Returns the Comparator object that describes underlying sorting technique.
  - If we are following default natural sorting order then this method returns null.

# Example


100
101
104
106
109
110
120
first()----------100
last----------120
headSet(109)----[100,101,104,106]
tailSet(109)------[109,110,120]
sebSet(104,110)-[104,106,109]
comparator()-----null

# TreeSet:

- The underlying data structure is balanced tree.
- Duplicate objects are not allowed.
- Insertion order is not preserved and it is based on some sorting order of objects.
- Heterogeneous objects are not allowed if we are trying to insert heterogeneous objects then we will get ClassCastException.
- Null insertion is possible(only once).

# Constructors:

- TreeSet t=new TreeSet();
  Creates an empty TreeSet object where all elements will be inserted according to default natural sorting order.
- TreeSet t=new TreeSet(Comparator c);
  Creates an empty TreeSet object where all objects will be inserted according to customized sorting order specified by Comparator object.
- TreeSet t=new TreeSet(SortedSet s);
- TreeSet t=new TreeSet(Collection c);

```
import java.util.*;
class TreeSetDemo
{
    public static void main(String[] args)
    {
        TreeSet t=new TreeSet();
        t.add("A");
        t.add("a");
        t.add("B");
        t.add("Z");
        t.add("L");
            System.out.println(t);          //[A, B, L, Z, a]
        }
}
```

```
import java.util.*;
class TreeSetDemo
{
    public static void main(String[] args)
    {
        TreeSet t=new TreeSet();
        t.add("A");
        t.add("a");
        t.add("B");
        t.add("Z");
        t.add("L");
        t.add(new Integer(10));          //ClassCastException
        t.add(null);                     //NullPointerException
        System.out.println(t);//[A, B, L, Z, a]
        }
}
```

# Null acceptance:

- For the empty TreeSet as the 1st element "null" insertion is possible but after inserting that null if we are trying to insert any other we will get NullPointerException.
- For the non empty TreeSet if we are trying to insert null then we will get NullPointerException.

# Example 2

```
import java.util.*;
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("B"));
		System.out.println(t);
		}
}
```

Output:

Runtime Exception.

Cannot be cast to java.lang.comparable

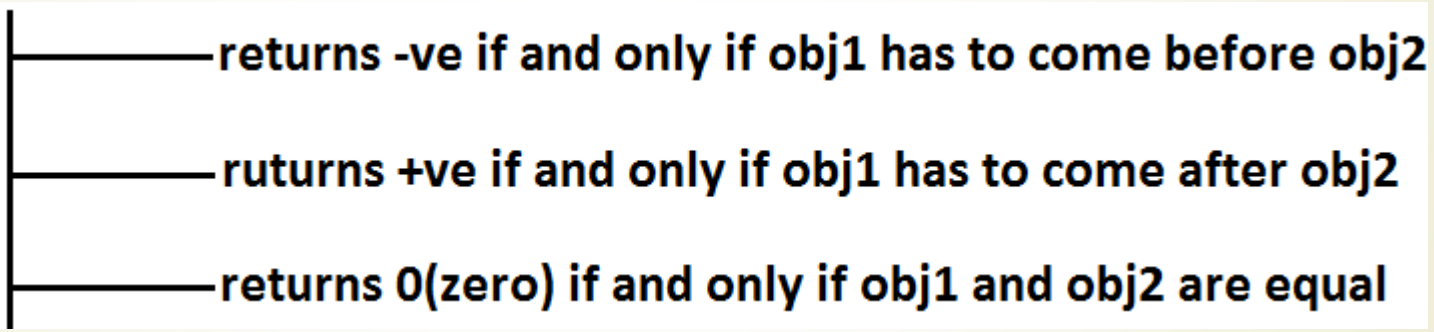
# Note

- Exception in thread "main" java.lang.ClassCastException: java.lang.StringBuffer cannot be cast to java.lang.Comparable
- If we are depending on default natural sorting order compulsory the objects should be homogeneous and Comparable otherwise we will get ClassCastException.
- An object is said to be Comparable if and only if the corresponding class implements Comparable interface.
- String class and all wrapper classes implements Comparable interface but StringBuffer class doesn't implement Comparable interface hence in the above program we are getting ClassCastException.

# Comparable interface:

- Comparable interface present in java.lang package
- Contains only one method compareTo() method.

public int compareTo(Object obj);



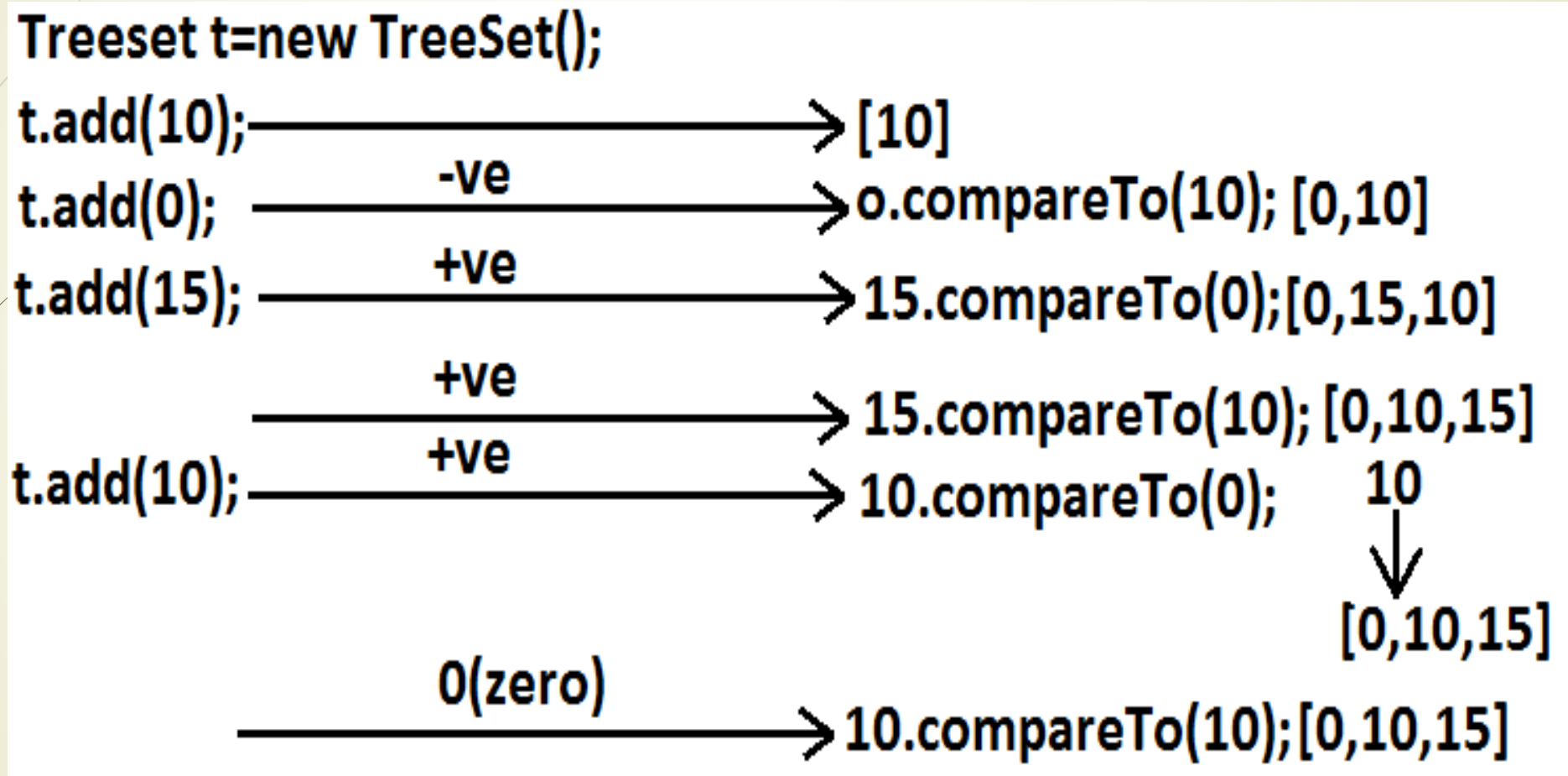
```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("A".compareTo("Z"));   //-25
        System.out.println("Z".compareTo("K"));   //15
        System.out.println("A".compareTo("A"));   //0
        //System.out.println("A".compareTo(new Integer(10)));
            //Test.java:8: compareTo(java.lang.String) in java.lang.String cannot
             be applied to (java.lang.Integer)
        //System.out.println("A".compareTo(null));       }
                                                          //NullPointerException
}
```

- If we depend on the default natural sorting order internally JVM will call compareTo() method which will insert objects to the TreeSet.
- Hence the objects should be Comparable.

```
TreeSet t = new TreeSet();
 t.add("B");
 t.add("Z");     // "Z".compareTo("B"); +Ve
 t.add("A");     // "A".compareTo("B"); -ve
 System.out.println(t);
```

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet t=new TreeSet();
        t.add(10);
        t.add(0);
        t.add(15);
        t.add(10);
        System.out.println(t);//[0, 10, 15]
    }
}
```

# compareTo() method analysis:



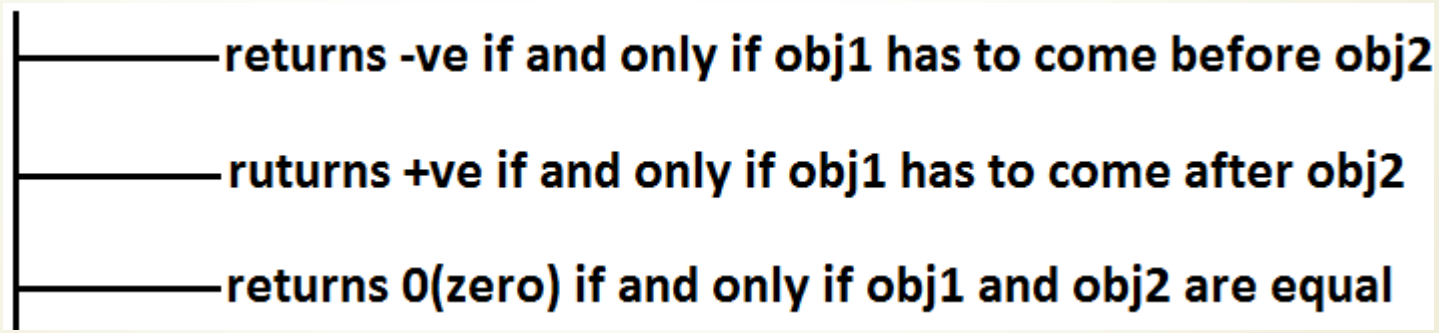
- If we are not satisfying with default natural sorting order (or) if default natural sorting order is not available then we can define our own customized sorting by Comparator object.
- Comparable meant for default natural sorting order.
- Comparator meant for customized sorting order.

# Comparator interface:

- Comparator interface present in java.util package this interface defines the following 2 methods.

**1) public int compare(Object obj1,Object Obj2);**

- 



**2) public boolean equals(Object obj);**

- Whenever we are implementing Comparator interface we have to provide implementation only for compare() method.
- Implementing equals() method is optional because it is already available from Object class through inheritance.

# Program

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        TreeSet t=new TreeSet(new MyComparator());  //---->(1)
        t.add(10);
        t.add(0);
        t.add(15);
        t.add(5);
        t.add(20);
        System.out.println(t);//[20, 15, 10, 5, 0]
    }}
```

```
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 -100;
        else return 0;
    }
}
```

- At line "1" if we are not passing Comparator object then JVM will always calls compareTo() method which is meant for default natural sorting order(ascending order)hence in this case the output is [0, 5, 10, 15, 20].
- At line "1" if we are passing Comparator object then JVM calls compare() method of MyComparator class which is meant for customized sorting order(descending order) hence in this case the output is [20, 15, 10, 5, 0].

# Analysis

```
TreeSet t=new TreeSet(new MyComparator());
t.add(10);     [10]
t.add(0);   +ve  compare(0,10) [10,0]
t.add(15);  -ve  compare(15,10)[15,10,0]
t.add(5);   +ve  compare(5,15) [15,5,10,0]
            +ve  compare(5,10) [15,10,5,0]
            -ve  compare(5,0)  [15,10,5,0]
t.add(20);       compare(20,15) [20,15,10,5,0]
```

# Various alternative implementations of compare() method:

```
public int compare(Object obj1,Object obj2)
    {
        Integer i1=(Integer)obj1;
        Integer i2=(Integer)obj2;
        //return i1.compareTo(i2);//[0, 5, 10, 15, 20]
        //return -i1.compareTo(i2);//[20, 15, 10, 5, 0]
        //return i2.compareTo(i1);//[20, 15, 10, 5, 0]
        //return -i2.compareTo(i1);//[0, 5, 10, 15, 20]
        //return -1;//[20, 5, 15, 0, 10]//reverse of insertion order
        //return +1;//[10, 0, 15, 5, 20]//insertion order
        //return 0;//[10]and all the remaining elements treated as duplicate.
    }
```

# Three cursors of Java

If we want to get objects one by one from the collection then we should go for cursor.

There are 3 types of cursors available in java. They are:

- Enumeration
- Iterator
- ListIterator

# Enumeration:

- We can use Enumeration(1.0v) to get objects one by one from the legacy collection objects.
- We can create Enumeration object by using elements() method.

**public Enumeration elements();**

Enumeration e=v.elements();

using Vector Object

# Enumeration interface defines the following two methods

- public boolean hasMoreElements();
- public Object nextElement();

```
import java.util.*;
class  EnumerationDemo
{       public static void main(String[] args)
        {
                Vector v=new Vector();
                for(int i=0;i<=10;i++)
                {
                        v.addElement(i);
                }
                System.out.println(v);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
                Enumeration e=v.elements();
                while(e.hasMoreElements())
                {
                        Integer i=(Integer)e.nextElement();
                        if(i%2==0)
                                System.out.println(i);//0 2 4 6 8 10
                }
                System.out.print(v);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        }
```

# Limitations of Enumeration:

- We can apply Enumeration concept only for legacy classes and it is not a universal cursor.
- By using Enumeration we can get only read access and we can't perform remove operations.
- To overcome these limitations sun people introduced Iterator concept in 1.2v.

# Iterator:

- We can use Iterator to get objects one by one from any collection object.
- We can apply Iterator concept for any collection object and it is a universal cursor.
- While iterating the objects by Iterator we can perform both read and remove operations.
- We can get Iterator object by using iterator() method of Collection interface.

  - public Iterator iterator();
  - Iterator itr=c.iterator();

# Iterator interface defines the following 3 methods.

- public boolean hasNext();
- public object next();
- public void remove();

```
import java.util.*;
class IteratorDemo {
    public static void main(String[] args)
    {
        ArrayList a=new ArrayList();
        for(int i=0;i<=10;i++)
        {
            a.add(i);
        }
        System.out.println(a);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        Iterator itr=a.iterator();
        while(itr.hasNext())
        {
            Integer i=(Integer)itr.next();
            if(i%2==0)
                System.out.println(i);//0, 2, 4, 6, 8, 10
            else
                itr.remove();
```

# Limitations of Iterator:

- Both enumeration and Iterator are single direction cursors only. That is we can always move only forward direction and we can't move to the backward direction.
- While iterating by Iterator we can perform only read and remove operations and we can't perform replacement and addition of new objects.
- To overcome these limitations sun people introduced listIterator concept.

# ListIterator:

- ListIterator is the child interface of Iterator.
- By using listIterator we can move either to the forward direction (or) to the backward direction that is it is a bi-directional cursor.
- While iterating by listIterator we can perform replacement and addition of new objects in addition to read and remove operations
- By using listIterator method we can create listIterator object.
- **public ListIterator listIterator();**

  **ListIterator itr=l.listIterator();**

  (l is any List object)

# ListIterator interface defines the following 9 methods.

- public boolean hasNext();
- public Object next(); forward
- public int nextIndex();
- public boolean hasPrevious();
- public Object previous(); backward
- public int previousIndex();
- public void remove();
- public void set(Object new);
- public void add(Object new);

```
import java.util.*;
class  ListIteratorDemo
{
	public static void main(String[] args)
	{
		LinkedList l=new LinkedList();
		l.add("balakrishna");
		l.add("venki");
		l.add("chiru");
		l.add("nag");
		System.out.println(l);//[balakrishna, venki, chiru, nag]
		ListIterator itr=l.listIterator();
		while(itr.hasNext())
		{
			String s=(String)itr.next();
			if(s.equals("venki"))
			{
				itr.remove();
			}
		}
		System.out.println(l);//[balakrishna, chiru, nag]
	}
```

The most powerful cursor is listIterator but its limitation is it is applicable only for "List objects".

| Property | Enumeration | Iterator | ListIterator |
|---|---|---|---|
| 1) Is it legacy ? | Yes | no | no |
| 2) It is applicable for ? | Only legacy classes. | Applicable for any collection object. | Applicable for only list objects. |
| 3) Movement? | Single direction cursor(forward) | Single direction cursor(forward) | Bi-directional. |
| 4) How to get it? | By using elements() method. | By using iterator()method. | By using listIterator() method. |
| 5) Accessibility? | Only read. | Both read and remove. | Read/remove/replace /add. |
| 6) Methods | hasMoreElement()<br>nextElement() | hasNext()<br>next()<br>remove() | 9 methods. |