Need Of Collections

An array is an indexed Collection of fixed number of homogeneous data elements.
The Main advantage of Arrays is we can represent multiple values with a single variable.
So that reusability of the code will be improved.

Limitations of Object type Arrays:
Arrays are fixed in size i.e. once we created an array with some size there is no chance of increasing or decreasing it's size based on our requirement. Hence to use arrays compulsory we should know the size in advance which may not possible always.

2) Arrays can hold only homogeneous data elements.
Ex:-
Student [] s = new Student [10000];
s[0] = new Student; (correct)
s[1] = new Customer(); (wrong)

But We can resolve this problem by using object Arrays.
Object [] o = new Object [10000];
o[0] = new Student();
o[1] = new Customer();

Arrays Concept is not implemented based on some standard data structure hence readymade method support is not available for every requirement we have to write the code explicitly. Which is complexity of programming.

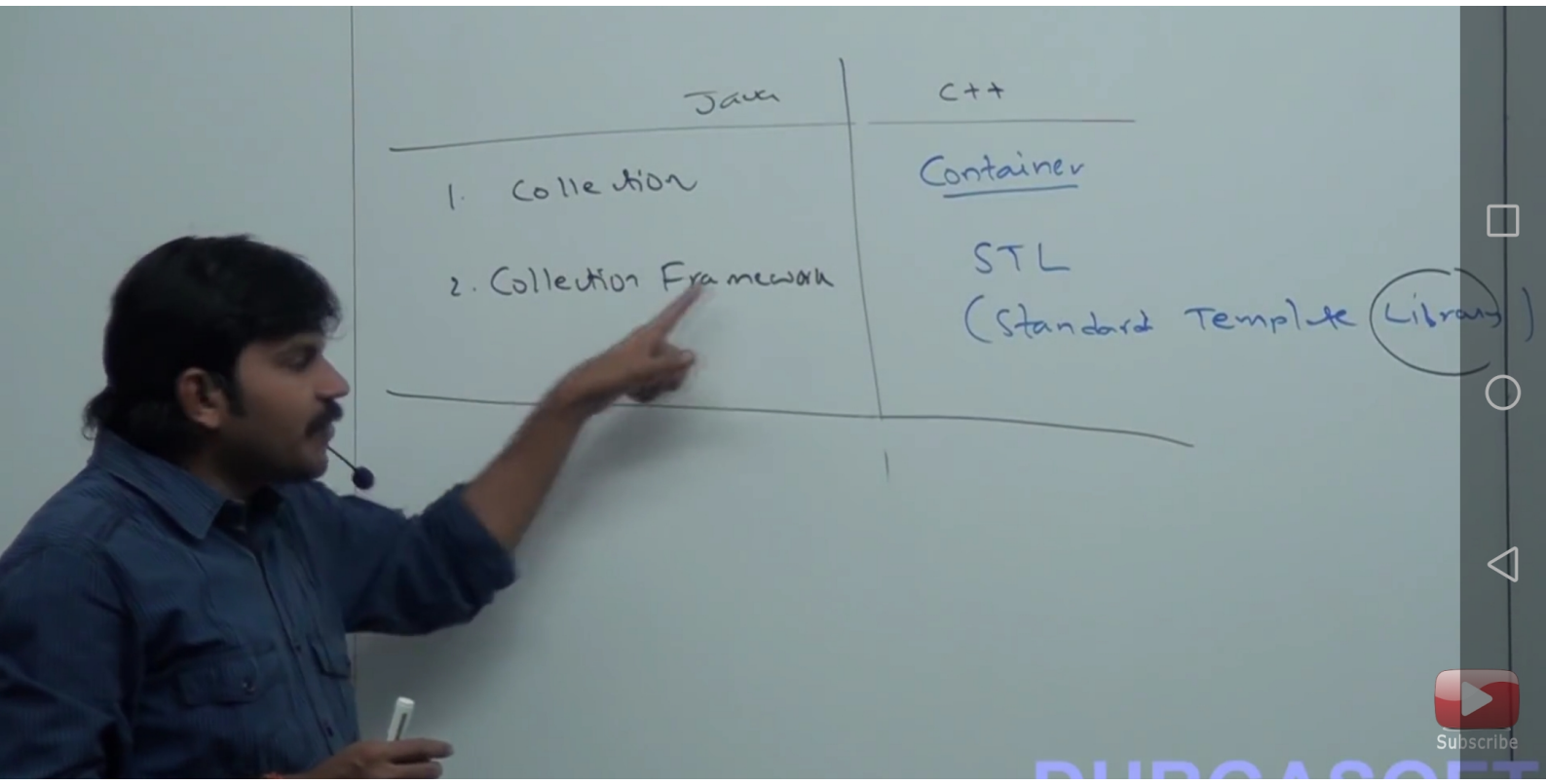
Need Of Collections

To overcome the above limitations of Arrays we should go for Collections.

Collections are growbable in nature. i.e. Based on our requirement we can increase (or) Decrease the size.
Collections can hold both homogeneous & Heterogeneous elements.
Every Collection class is implemented based on some standard data structure. Hence readymade method support is available for every requirement. Being a programmer we have to use this method and we are not responsible to provide implementation.

Difference between Arrays and Collections:

Arrays	Collections
1. Arrays are fixed in size.	1. Collections are growable in nature. I.e. based on our requirement we can increase or decrease the size.
2. Wrt memory arrays are not recommended to use.	2. Wrt to memory collections are recommended to use.
3. Wrt Performance Arrays are recommended to use.	3. Wrt Performance collections are not recommended to use.
4. Array can hold only homogeneous datatype elements	4. Collections can hold both homogeneous and heterogeneous elements.
5. There is no underlying data structure for arrays and hence readymade method support is not available	5. Every Collections class is implemented based on some standard data structure. Hence readymade method support is available for every requirement.
6. Array can hold both primitives and object types	6. Collections can hold only objects but not primitives.


Java
C++
1. Collection
2. Collection Framework
Container
STL
(Standard Template Library)
Subscribe

9-Key Interfaces of Collection Framework

i. Collection :

* If we want to represent a group of individual objects as a single entity then we should go for Collection.

* Collection interface defines the most common methods which are applicable for any Collection object.

* In general collection interface is considered as root interface of Collection Framework.

Note: there is no concrete class which implements collection interface directly.

erence between Collection & Collections

ction is an interface which can be used to represent
up of individual objects as a single entity.

ctions is an utility class present in java.util.package
fine several utility methods (like Sorting, Searching..)
ollection objects.

Difference between Collection & Collections

* Collection is an interface which can be used to represent a group of individual objects as a single entity.

* Collections is an utility class present in java.util.package to define several utility methods (like Sorting, Searching..) for Collection objects.

9 key interfaces of Collection Framework

ii. List :

* List is 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 preserved then we should go for List.

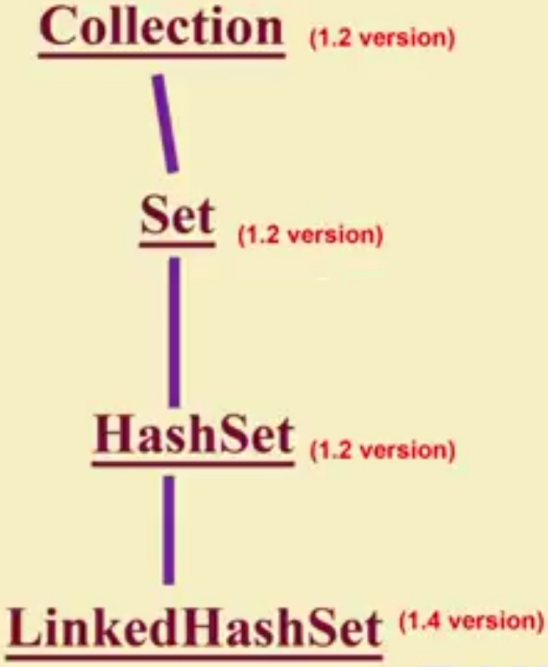
9 key interfaces of Collection Framework

iii. Set:

* 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 and insertion order not preserved then we should go for Set.

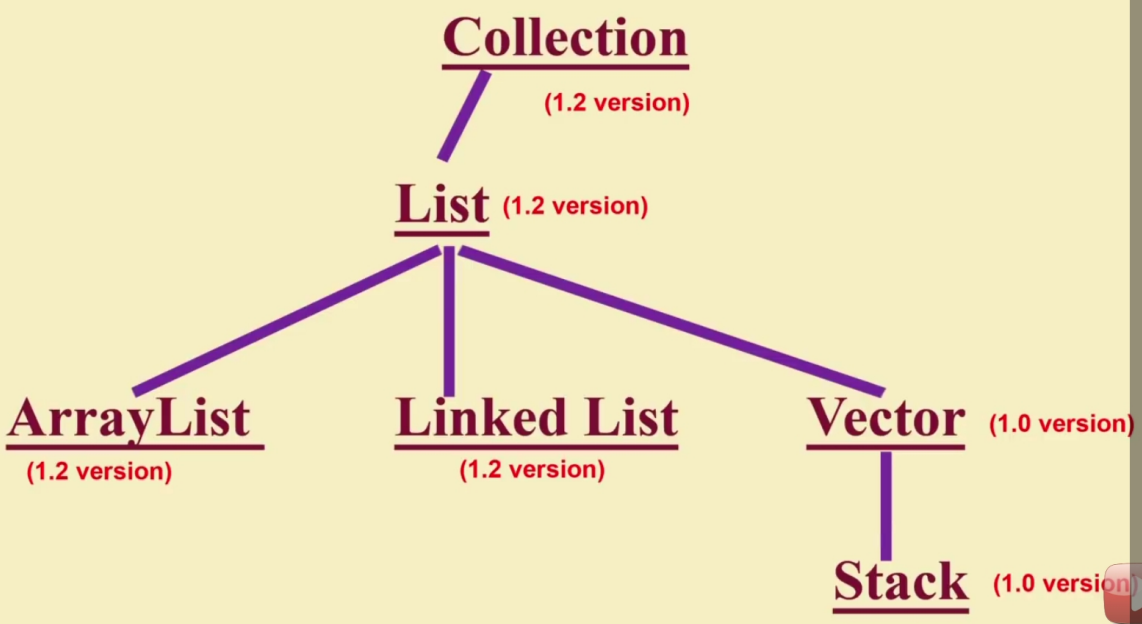
9 key interfaces of Collection Framework

iii. Set :



9 key interfaces of Collection Framework

ii. List :



Note: Vector and Stack classes are re-engineer in 1.2v to implement list interface

Difference between List & Set

List

* Duplicates are allowed
* Insertion order preserved

Set

* Duplicates are not allowed
* Insertion order not preserved

9 key interfaces of Collection Framework

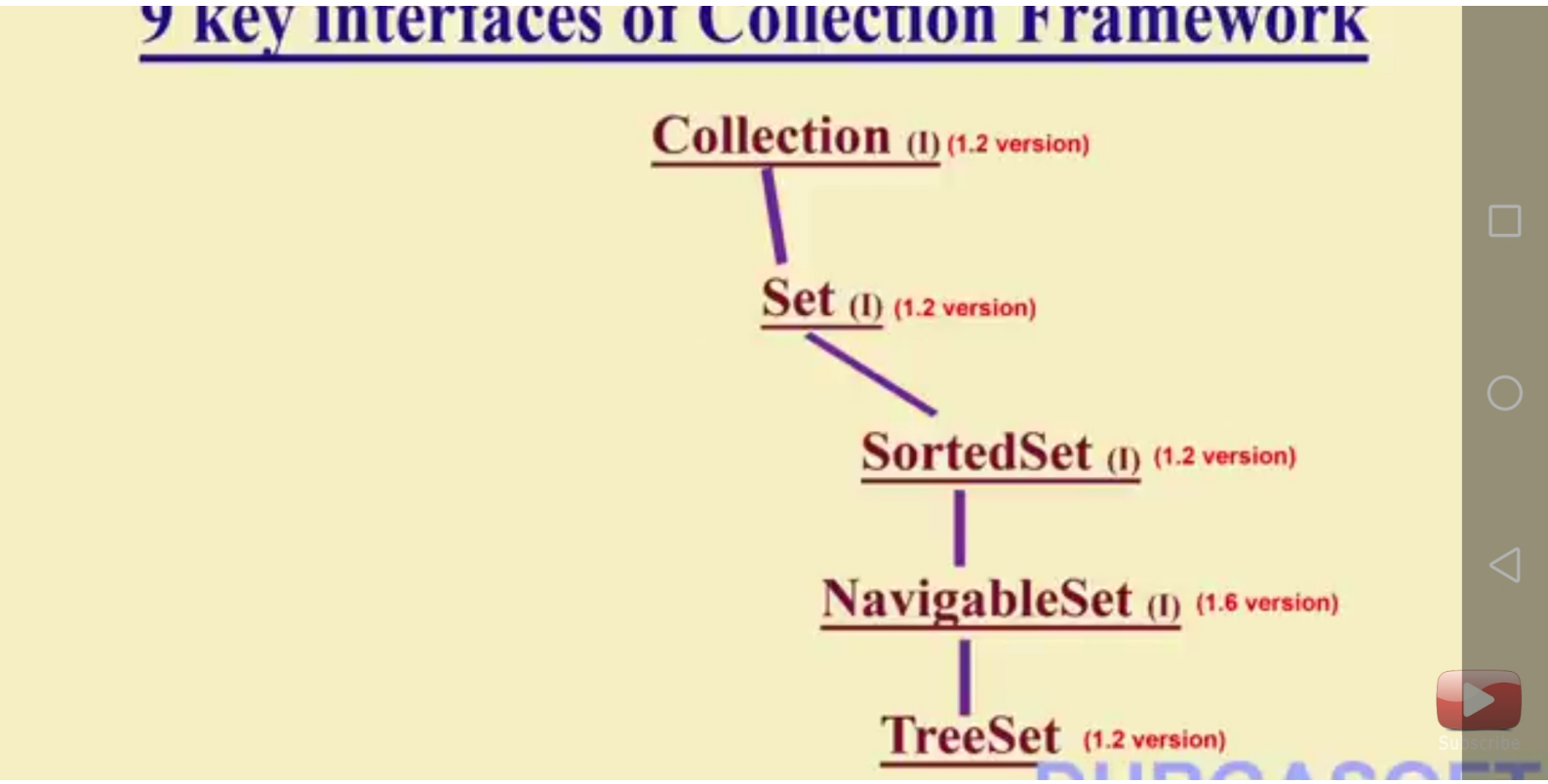
iv. SortedSet:

* It is the child interface of Set.
* If we want to represent a group of individual objects as a single entity where duplicates are not allowed but all objects should be inserted according to some sorting orde then we should go for SortedSet.

9 key interfaces of Collection Framework

v. NavigableSet:

* It is the child interface of SortedSet if defines several methods for navigation purposes.


9 key interfaces of Collection Framework
Collection (I) (1.2 version)
Set (I) (1.2 version)
SortedSet (I) (1.2 version)
NavigableSet (I) (1.6 version)
TreeSet (1.2 version)

9 key interfaces of Collection Framework

vi. Queue :

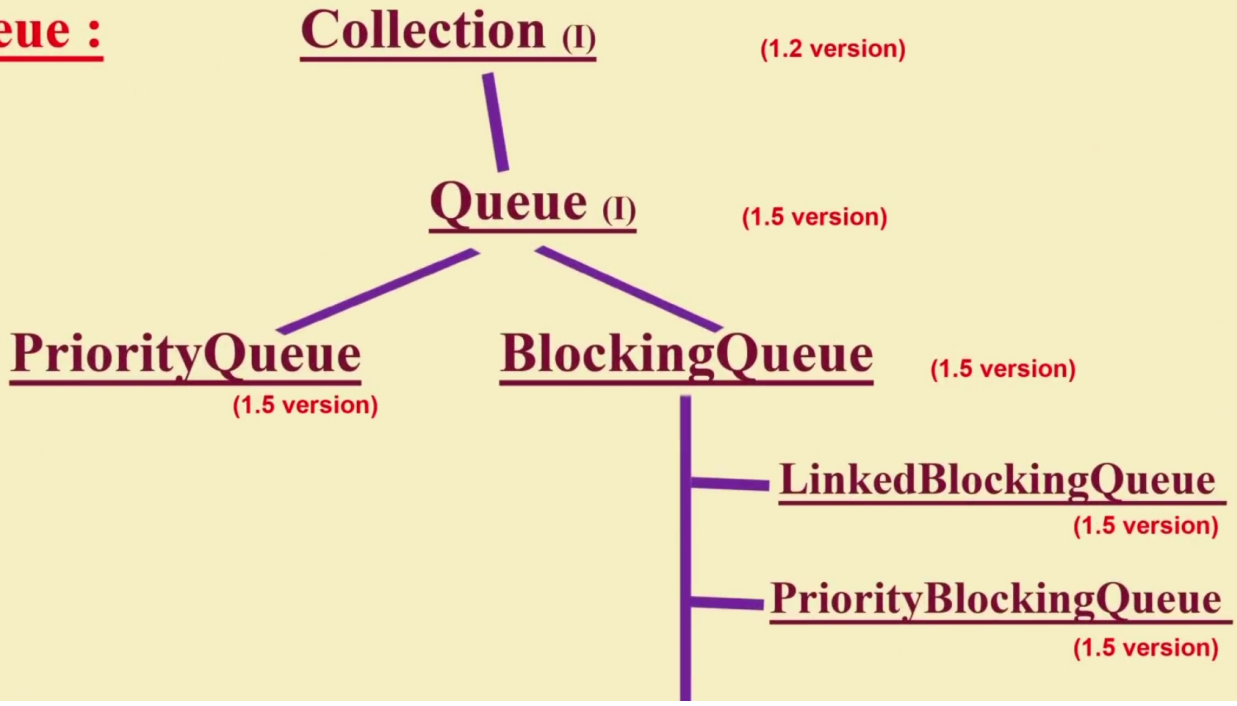
* It is child interface of Collection.

* If we want to represent a group of individual objects prior to processing then we should go for Queue.

Ex: before sending a mail all mail id's we have to store somewhere and in which order we saved in the same order mail's should be delivered (First in First out) for this requirement Queue concept is the best choice.

9 key interfaces of Collection Framework

vi. Queue :



9 key interfaces of Collection Framework

Note :

* All the above interfaces (Collection, List, Set, SortedSet, NavigableSet and Queue) meant for representing a group of individual objects.

* If we want to represent a group of objects as key value pairs then we should go for Map Interface.

9 key interfaces of Collection Framework

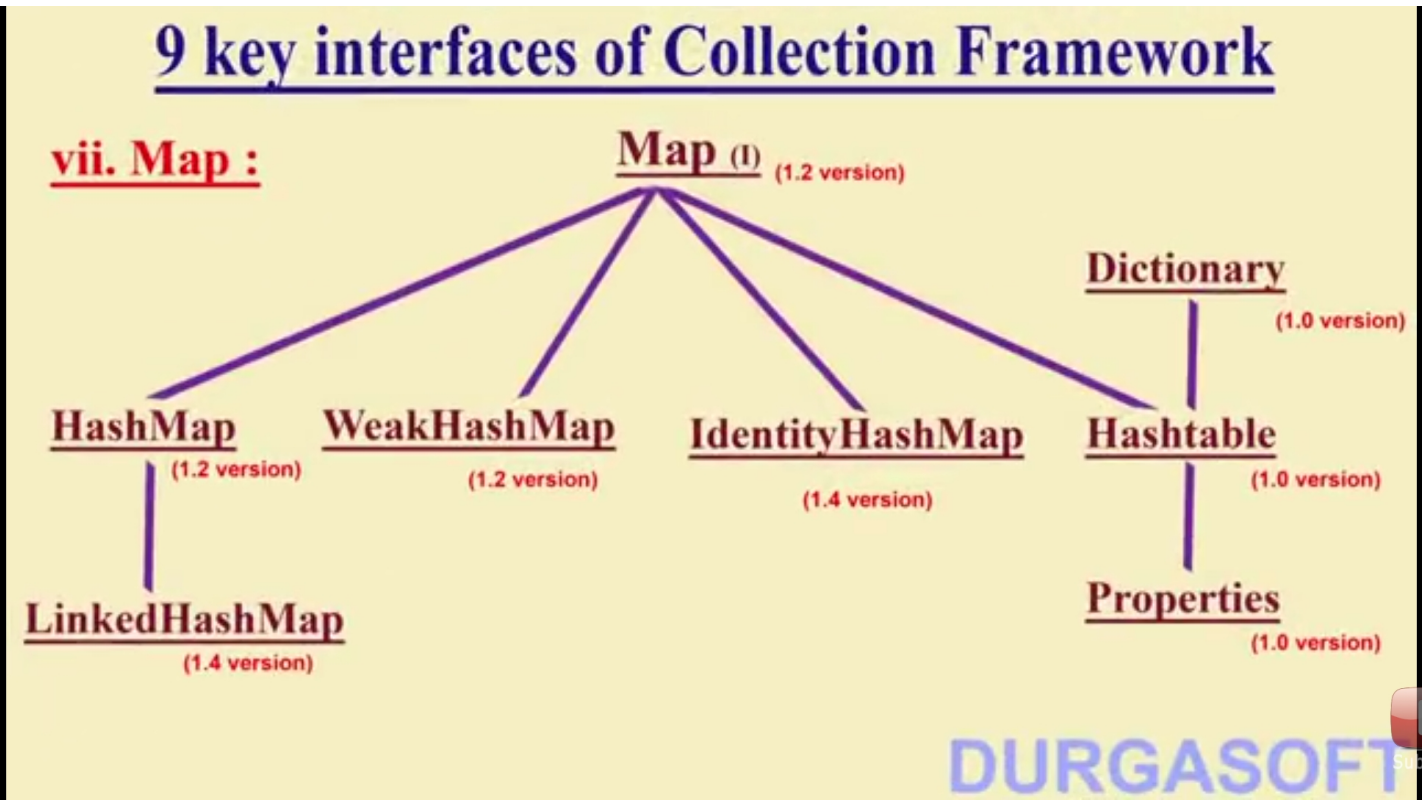
vii. Map:

* Map is not the child interface of Collection.

* If we want to represent a group of individual objects as key value pairs then should go for Map.

Ex :	Roll No	Name
	101	Durga
	102	Ravi
	103	Venkat

Both key and value are objects, duplicated keys are not


9 key interfaces of Collection Framework
vii. Map :
Map (I)
(1.2 version)
Dictionary
(1.0 version)
HashMap
(1.2 version)
WeakHashMap
(1.2 version)
IdentityHashMap
(1.4 version)
Hashtable
(1.0 version)
LinkedHashMap
(1.4 version)
Properties
(1.0 version)
DURGASOFT

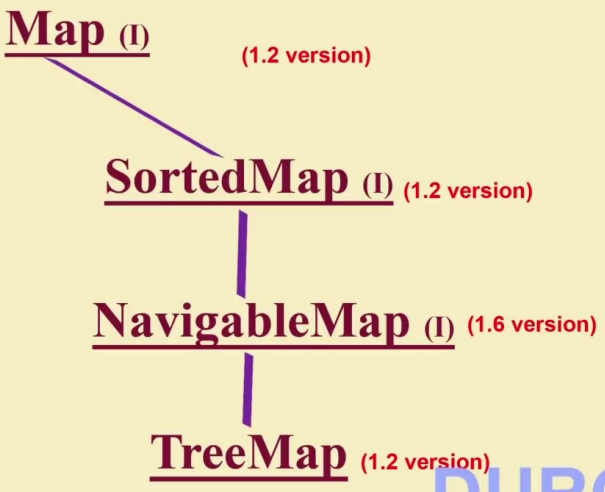
viii. SortedMap:

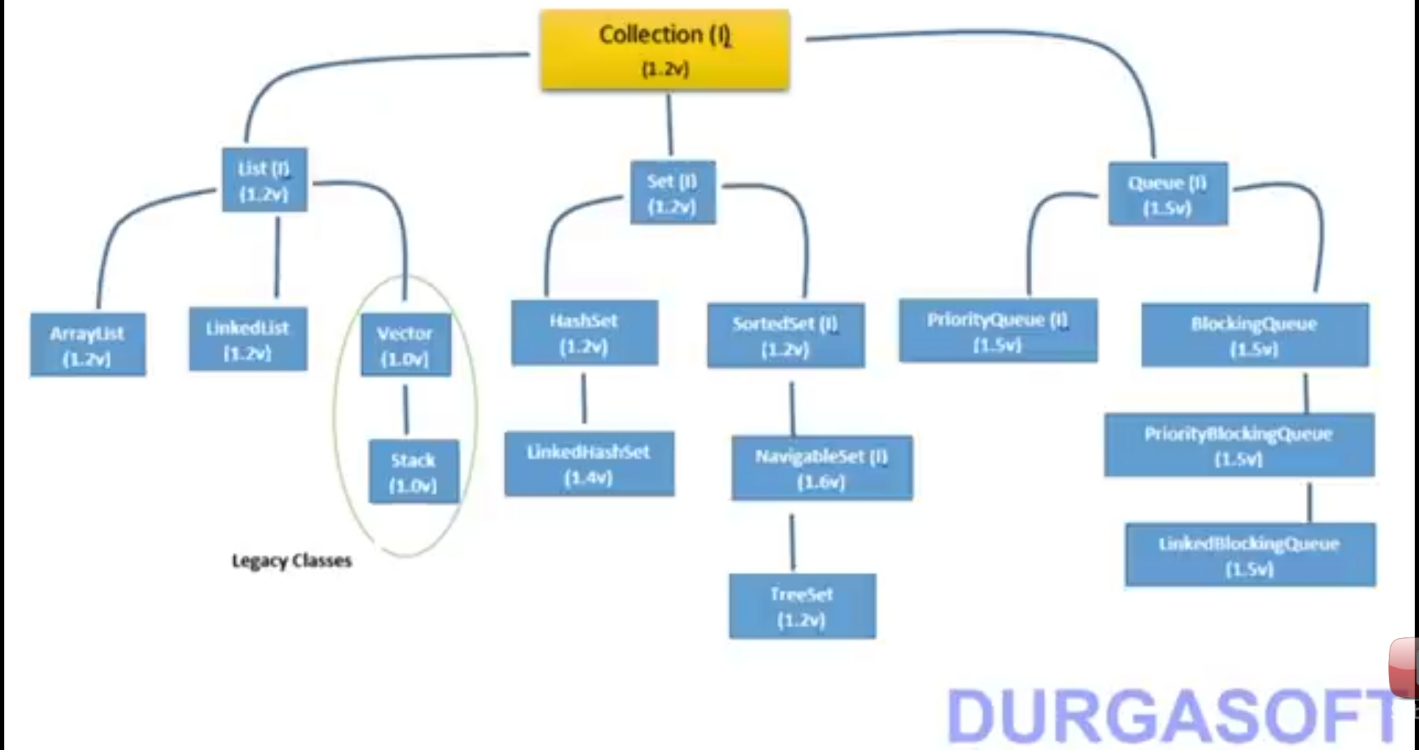
* It is the child interface of map.

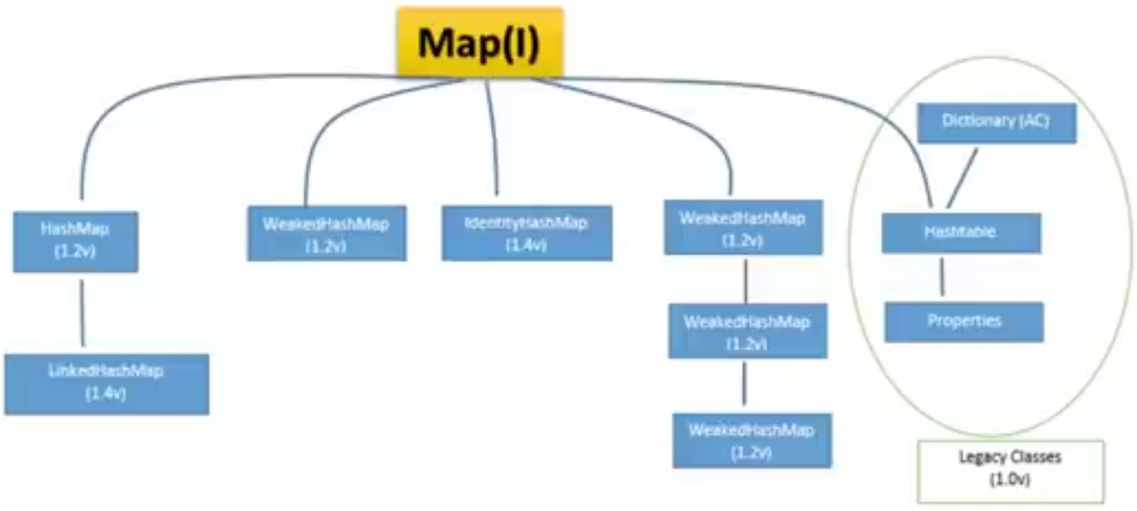
* If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap

ix. NavigableMap:

* It is the child interface of sorted map, it defines several utility methods for navigation purpose.




Collection (I)
(1.2v)
List (I)
(1.2v)
Set (I)
(1.2v)
Queue (I)
(1.5v)
ArrayList
(1.2v)
LinkedList
(1.2v)
Vector
(1.0v)
Stack
(1.0v)
HashSet
(1.2v)
SortedSet (I)
(1.2v)
LinkedHashSet
(1.4v)
NavigableSet (I)
(1.6v)
TreeSet
(1.2v)
PriorityQueue (I)
(1.5v)
BlockingQueue
(1.5v)
PriorityBlockingQueue
(1.5v)
LinkedBlockingQueue
(1.5v)
Legacy Classes
DURGASOFT



Sorting:

1. Comparable(I)
2. Comparator(I)

Cursors:

1. Enumeration(I)
2. Iterator(I)
3. ListIterator(I)

Utility Classes:

1. Collections
2. Arrays

Collection Interface :

* If we want to represent a group of individual objects as a single entity then we should go for Collection.

* In general collection interface is considered as root interface of Collection Framework.

* Collection interface defines the most common methods which are applicable for any collection object

Important methods of Collection Interface

boolean add(Object o)
boolean addAll(Collection c)
boolean remove(Object o)
boolean removeAll(Collection c)
boolean retainAll(Collection c)
void clear()

boolean contains(Object o)
boolean containsAll(Collection c)
boolean isEmpty()
int size()
object[] toArray()
Iterator iterator()

Collection Interface

Note:

Collection interface doesn't contain any method to retrieve objects there is no concrete class which implements collection class directly.

List 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 allowed and insertion order must be preserved then we should go for List.
* We can differentiate duplicates by using index.
* We can preserve insertion order by using index, hence index play very important role in list interface.

List interface specific methods

void add(int index, Object o)
boolean addAll(int index, Collection c)
object get(int index)
object remove(int index)
object set(int index, Object new)
int indexOf(Object o)
int lastIndexOf(Object o)
ListIterator listIterator();

ArrayList Constructors

1. ArrayList al = new ArrayList()
   Creates an empty Array list object with default initial capacity 10. Once Array List reaches its map capacity a new Array List will be created with new capacity = (currentcapacity * 3/2) + 1.

ArrayList Constructors

2. ArrayList al = new ArrayList(int initialCapacity);

3. ArrayList al = new ArrayList(Collection c);

Example for ArrayList

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

* Usually we can use Collections to hold and transfer Objects from one place to another place, to provide support for this requirement every Collection already implements Serializable and Cloneable interfaces.

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

* Hence if our frequent operation is retrieval operation then ArrayList is the best choice.

ArrayList

```
ArrayList l1=new ArrayList ();
LinkedList l2=new LinkedList ();
System.out.println (l1 instanceOf Serializable);    //true
System.out.println (l2 instanceOf Cloneable);       //true
System.out.println (l1 instanceOf RandomAccess);    //true
System.out.println (l2 instanceOf RandomAccess);    //false
```

ArrayList

* ArrayList is best choice if our frequent operation is retrieval operation (Because ArrayList implements RandomAccess interfaces)

* ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle (Because several shift operation are require)

Difference between ArrayList & Vector

ArrayList	Vector
Every method present ArrayList is non synchronize	Every method present in LinkedList is synchronize
At a time multiple threads are allowed to operate on ArrayList Object and hence **ArrayList** is not thread safe	At a time only one thread is allowed to operate on **Vector** Object is thread safe
Threads are not required to wait to operate on ArrayList, hence relatively performance is high.	Threads are required to wait to operate on Vector Object and hence relatively performance is low
Introduced in 1.2 version And it is non legacy class	Introduced in 1.0 version and it is a legacy class

How to get synchronized version of ArrayList Object?

- By default ArrayList is Object is non-synchronized but we can get synchronized version of ArrayList by using Collection class synchronizedList () method.

```
public static List synchronizedList(List l)
```

How to get synchronized version of ArrayList Object?

public static List synchronizedList(List l)

Non-Synchronized

```
ArrayList l1=new ArrayList ();
```

Synchronized

```
List l=Collections.synchronizedList (l1);
```

How to get synchronized version of ArrayList Object?

* Similarly we can get Synchronized version of Set, Map Objects by using the following methods of Collections class.

Public static Set synchronizedSet (Set s);

Public static Set synchronizedMap (Map m);

LinkedList

* The underlying data structure is Double Linked List.
* Insertion order is preserved .
* Duplicates are allowed.
* Heterogeneous Objects are allowed.
* Null insertion is possible.

LinkedList

* LinkedList implements Serializable and Clonable interfaces but not RandomAccess interface.

* LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.

* LinkedList is the worst choice if our frequent operation is retrieval operation.

LinkedList

* Usually we can use LinkedList to implement stacks and queues to provide support for this requirement LinkedList class defines following specific methods.

```
void addFirst();
void addLast();
Object getFirst();
Object getLast();
Object removeFirst();
Object removeLast();
```

LinkedList Constructors

* LinkedList l1=new LinkedList();

  Creates an empty LinkedList Object

* LinkedList l1=new LinkedList(Collection c);

  Creates an equivalent LinkedList Object for the given Collection

LinkedList Demo program

```
import java.util.*;
public class LinkedListDemo {
  public static void main() {
    LinkedList l1=new LinkedList ();
    l1.add (“durga”);
    l1.add (30);
    l1.add (null);
    l1.add (“durga”);
    l1.set (0,”software”);
    l1.add (0,”venkey”);
    l1.addFirst (“ccc”);
    System.out.println (l);//[ccc,venkey,software,30,null]
  }
}
```

Difference between ArrayList & LinkedList

ArrayList	LinkedList
It is the best choice if our frequent operation is retrieval	It is the best choice if our frequent Operation is insertion and deletion
ArrayList is the worst choice if our frequent operation is insertion or deletion	LinkedList is the worst choice if our frequent operation is retrieval operation
Underlying data structure for ArrayList is resizable or growable Array.	Underlying data structure is Double Linked List.
ArrayList implements RandomAccess interface	LinkedList doesn't implement RandomAccess interface

Vector

* The underlying Data structure for the vector is resizable array or growable array.
* Duplicate objects are allowed.
* Insertion order is preserved.
* 'null' insertion is possible.
* Heterogeneous objects are allowed.
* Vector class implemented Serializable, Cloneable and RandomAccess Interfaces.
* Most of the methods present in Vector are synchronized. Hence Vector object is Thread-safe.
* Best choice if the frequent operation is retrieval.

Vector specific methods

For removing Objects :

Remove (Object o)	[from Collection]
removeElement (Object o)	[from Vector]
remove (int index)	[from List]
RemoveElementAt (int index)	[from Vector]
clear ()	[from Collection]
removeAllElements ()	[from Vector]

Vector specific methods

For Accessing Elements :

Object get (int index)	[from Collection]
Object elementAt (int index)	[from Vector]
Object firstElement ()	[from Vector]
Object lastElement ()	[from Vector]

Other Methods:

```
int size();
int capacity ();
Enumeration elements ();
```

Vector specific methods

For removing Objects :

Remove (Object o)	[from Collection]
removeElement (Object o)	[from Vector]
remove (int index)	[from List]
RemoveElementAt (int index)	[from Vector]
clear ()	[from Collection]
removeAllElements ()	[from Vector]

Constructors of vector class

1) Vector v = new Vector();

- Creates an empty vector object with default initial capacity 10, Once vector reaches it's max capacity a new vector Object will be Created with new capacity = 2 * current capacity.

2) Vector v = new Vector(int initialCapacity);

- Creates an empty Vector Object with specified initial capacity

3) Vector v = new Vector(int initialCapacity, int incrementalCapacity);

4) Vector v = new Vector(Collection c);

- Creates an equivalent Vector Object for the given Collection

Demo program for vector

```
import  java.util.*;
class VectorDemo {
   public static void main(String arg[]) {
     Vector v = new Vector ();
     System.out.println (v.capacity ());       //[10]
     for (int i = 0;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);
   }
}
```

Constructor of Stack

```
Stack s=new Stack ();
```

Methods in 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 then it returns -1.

5) Object pop();
   - For inserting an object to the stack

Methods in 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 then it returns -1.

5) Object pop();
   - For inserting an object to the stack

Demo program for Stack

```
import java.util.*;
class StackDemo {
 public static void main (String arg[]) {
   Stack s = new Stack ();
   s.push ("A");
   s.push ("B");
   s.push ("C");
   System.out.println(s);     //[A,B,C]
   System.out.println (s.search ("A"));  //[3]
   System.out.println (s.search("Z"));   //[-1]
 }
}
```

Three cursors of Java

* If we want to retrieve Objects one by one from the Collection, then we should go for Cursors.

* There are three types of cursors are available in java.

    * Enumeration
    * Iterator
    * ListIterator

Enumeration

* Introduced in 1.0 version(for Legacy).
* We can use Enumeration to get Objects one by one from the old Collection Objects(Legacy Collections).
* We can create Enumeration Object by using elements() method of Vector class.

```
Public Enumeration elements ();
```

Example :

```
Enumeration e=v. elements ();
```

Method of Enumeration

* Enumeration defines the following two methods
    * public boolean hasMoreElements();
    * public Object nextElement();

Demo program for Enumeration

```
import java.util.*;
class EnumaretionDemo1 {
  public static void main(String arg[]) {
    Vector v = new Vector ();
    for (int i =0;i<=10 ;i++ ) {
      v.addElement (i);
    }
    System.out.println (v); //[0,1,2,3,4,5….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.println (v);  //[0,1,2,3,4,…10]
    }
}
```

Enumeration

Limitations of Enumeration:

* Enumeration concept is applicable only for legacy classes and hence it is not a universal cursor.
* By using Enumeration we can get only read access and we can't perform remove operation.

Note: To overcome above limitations of Enumeration we should go for Iterator.

Iterator

1. We can apply Iterator concept for any Collection object hence it is universal cursor.

2. By using Iterator we can perform both read and remove operations.

Iterator

* We can create Iterator object by using iterator () method of Collection interface.

```
public Iterator iterator ();
```

Example:

```
Iterator itr=C. iterator();
```

* where C is any Collection Object

Methods in Iterator

* Iterator interface defines the following three methods.

i. public boolean hasNext ()

ii. public Object next()

iii. public void remove()

Demo program for Iterator

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

Limitations of Iterator

1. 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.
2. By using Iterator we can perform only read and remove operations and we can't perform replacement of new Objects.

Note : To overcome above limitations of Iterator we should go for ListIterator

ListIterator

1. By using ListIterator we can move either to the forward direction or to the backward direction, and hence ListIterator is bidirectional cursor.

2. By using ListIterator we can perform replacement and addition of new Objects in addition to read and remove operations.

ListIterator

* We can create ListIterator Object by using listIterator () method of List Interface.

public ListIterator listIterator ()

Example:

ListIterator itr=l. listIterator ();

* where l is any List Object

Methods in ListIterator

* ListIterator is the child interface of Iterator and hence all methods of Iterator by default available to ListIterator.
* ListIterator Interface defines the following 9 methods

forward direction

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

Backward direction

4. public boolean hasPrevious()
5. public void previous()
6. public int previousIndex ()

other capability methods

7. public void remove()
8. public void set(Object new)
9. public void add(object new)

Demo program for ListIterator

```
import java.util.*;
class ListIteratorDemo {
  Public static void main (String arg[]) {
    LinkedList l = new LinkedList ();
    l.add ("balakrishna");
    l.add ("chiru");
    l.add ("venky");
    l.add ("nag");
    System.out.println (l);
 //[balakrishna, venky, chiru, nag]
```

```
    ListIterator ltr = l. listIterator ();
    While (ltr. hasNext ()) {
      String s = (String) ltr.next ();
      if (s. equals (“venky”)) {
        ltr. remove ();
      } else If (s. equals ("nag")) {
        ltr.add ("chaitu");
      } else if (s. equals (“chiru”)) {
        ltr. set (“charan”);
      }
    }
    System.out.println (l);
  //[balakrishna, charan, nag, chaitu]
  } }
```

ListIterator

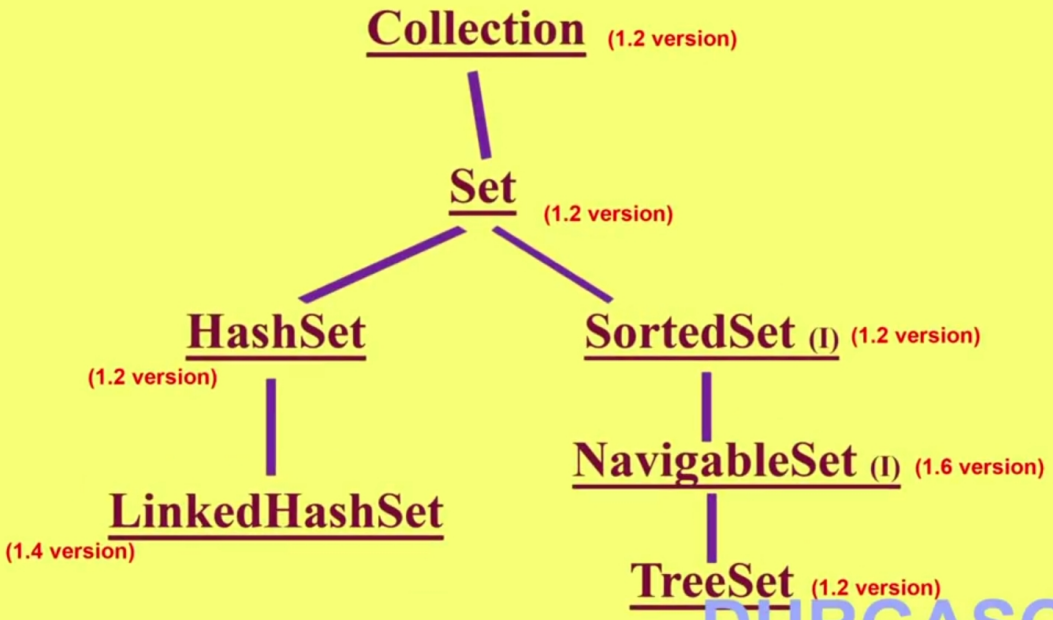
Note : ListIterator is the most powerful cursor but its limitation is, it is applicable only for List implemented class objects and it is not a universal cursor.

Property	Enumeration	Iterator	ListIterator
Applicable for	Only legacy classes	Any Collection classes	Only List classes
Movement	Only forward direction(single direction)	Only forward direction(single direction)	Both forward and backward direction(bi directional)
Accessibility	Only read access	Both read and remove	Read ,remove, replace and addition of new objects
How to get it?	By using elements() method of Vector class	By using iterator() method of Collection interface	By using listIterator() method of List interface
Methods	2 methods hasMoreElements() nextElement()	3 methods hasNext () next() remove()	9 methods
Is it legacy	"yes" (1.0v)	"no" (1.2V)	"no" (1.2V)

Implementation classes of cursors

```
import java.util.*;
class cursorDemo {
public static void main (String [] args) {
   Vector v=new Vector ();
   Enumeration e=v. element ();
   Iterator itr=v.iterator ();
   ListIterator ltr= v.listIterator();
   System.out.println (e.getClass (). getName ());   // java.uitl.Vector$1
   System.out.println (itr.getClass (). getName ());  // java.util.Vector$Itr
   System.out.println (itr.getClass (). getName ());  // java.util.Vector$ListItr
 }
}
```

Set Interface :



Set

1. Set is the child interface of Collection.

2. If we want to represent a group of individual objects as a single entity, where duplicates are not allowed and insertion order is not preserved then we should go for Set.

3. Set interface doesn't contain any new methods. So we have to use only Collection interface methods.

HashSet

* The underlying data structure is Hashtable.
* Duplicates are not allowed. If we are trying to insert duplicates, we won't get any compiletime or runtime errors. add() method simply returns false.
* Insertion order is not preserved and all objects will be inserted based on hash-code of objects.
* Heterogeneous objects are allowed.
* ' null ' insertion is possible.
* implements Serializable and Clonable interfaces but not RandomAccess.
* HashSet is the best choice, if our frequent operation is Search operation.

Constructors of HashSet

1) HashSet h = new HashSet();
   - Creates an empty HashSet object with default initial capacity 16 & default Fill Retio 0.75

2) HashSet h = new HashSet(int initalCapacity);
   - Creates an empty HashSet object with specified initial capacity & default Fill Retio 0.75

3) HashSet h = new HashSet(int initalCapacity, float loadFactor);
   - Creates an empty HashSet object with specified initial capacity & specified Load Factor (or Fill Ratio)

4) HashSet h = new HashSet(Collection c);
   - For inter conversion between Collection objects.

Constructors of HashSet

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.

Demo program for HashSet

```
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.
* Introduced in 1.4 version.
* It is exactly same as HashSet except the following differences.

HashSet	LinkedHashSet
The underlying datastructure is Hash table.	The underlying datastructure is Hash table + Linked List . (that is hybrid data structure)
Insertion order is not preserved	Insertion order is preserved
Introduced in 1.2 version	Introduced in 1.4 version.

Demo program for LinkedHashSet

```
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]
} }
```

LinkedHashSet

Note :

- LinkedHashSet is the best choice to develop cache based applications, where duplicates are not allowed and insertion order must be preserved.

SortedSet (I)

1. It is the child interface of set.

2. 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.

SortedSet Specific methods

Object first() - returns first element of the SortedSet

Object last() - returns last element of the 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 comparator()
- returns Comparator object that describes underlying sorting technique.
If we are using default natural sorting order then we will get null.

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

1. first() --> 100
2. last() --> 115
3. headset(104) --> [100,101,103]
4. tailSet(104) --> [104,107,110,115]
5. subset(103,110) --> [103,104,107]
6. comparator() --> null

Note :

1. Default natural sorting order for numbers Ascending order and for String alphabetical order.
2. We can apply the above methods only on SortedSet implemented class objects. That is on the TreeSet object.

TreeSet

1. The underlying data structure for TreeSet is Balanced Tree.
2. Duplicate objects are not allowed.
3. Insertion order not preserved, but all objects will be inserted according to some sorting order.
4. Heterogeneous objects are not allowed. If we are trying to insert heterogeneous objects then we will get runtime exception saying ClassCastException.
5. Null Insertion is allowed, but only once.

TreeSet Constructors

1. TreeSet t=new TreeSet();
   - Creates an empty TreeSet object where elements will be inserted according to default natural sorting order.

2. TreeSet t=new TreeSet(Comparator c);
   - Creates an empty TreeSet Object where elements will be inserted according to customized sorting order.

3. TreeSet t=new TreeSet(SortedSet s);

4. TreeSet t=new TreeSet(Collection c);

Example

```
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]
   }
}
```

Example

```
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

1. For empty TreeSet as the first element null insertion is possible.
   But After inserting that null if we are trying to insert any another element we will get NullPointerException.

2. For Non empty TreeSet If we are trying to insert Null then we will get NullPointerExcpetion.

Example

```
import java.util.TreeSet;
class TreeSetDemo1 {
    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);      // ClassCastException
    }
}
```

Example

```
import java.util.TreeSet;
class TreeSetDemo1 {
    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);      // ClassCastException
    }
}
```

Note :

1. If we are depending on default natural sorting order then objects should be homogeneous and comparable. Otherwise we will get runtime exception saying ClassCastException.
2. An object is Said to be comparable if and only if the corresponding class implements java.lang.comparable interface.
3. String Class and all wrapper classes already implements comparable interface. But StringBuffer doesn't implement comparable interface.

** Hence in the above program we got ClassCastException **

Comparable Interface :

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

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

Example :

obj1.CompareTo(obj2)

- |---> returns –ve iff obj1 has to come before obj2
- |---> returns +ve iff obj1 has to come after obj2
- |---> returns 0 iff obj1 & obj2 are equal.

Example

```
class Test {
    public static void main(String[] args) {
        System.out.println("A".compareTo("Z"));  // -ve
        System.out.println("Z".compareTo("B"));   // +ve
        System.out.println("A".compareTo("A"));   // 0
        System.out.println("A".compareTo(null));   // NullPointerException

    }
}
```

* If we depending on default natural sorting order internally JVM will call CompareTo() method will inserting 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);       // [A, B, Z]
```

* If we depending on default natural sorting order internally JVM will call CompareTo() method will inserting 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);       // [A, B, Z]
```

Note :

1. 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.

2. Comparable ment for Default Natural Sorting order where as Comparator ment for customized Sorting order.

Comparator Interface

* We can use comparator to define our own sorting (Customized sorting).
* Comparator interface present in java.util package.
* It defines two methods. compare and equals.

1) public int compare(Object obj1, Object obj2)

|---> returns –ve iff obj1 has to come before obj2
|---> returns +ve iff obj1 has to come after obj2
|---> returns 0 iff obj1 & obj2 are equal.

2) public boolean equals();

* When ever we are implementing Comparator interface, compulsory we should provide implementation for compare() method.

* And implementing equals() method is optional, because it is already available in every java class from Object class through inheritance.

Write A Program to insert integer objects into the 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()); — line1
    t.add(10);
    t.add(0);
    t.add(15);
    t.add(20);t
    t.add(20);
    System.out.println(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;
    }
}
```

Output: [20,15,10,0]

Integer Objects Into TreeSet, Desending order:

```
Treeset t=new Treeset(new MyComparator());     ----------> line1
t.add(10);
t.add(0);    -------  +ve  ----> compare(0,10);
t.add(15);   -------  -ve   ----> compare(15,10);

t.add(20);   -------  +ve  ----> compare(20,10);
t.add(20);   -------  -ve   ----> compare(20,15);

t.add(20);   -------  +ve  ----> compare(20,10);
t.add(20);   -------  -ve   ----> compare(20,15);
t.add(20);   -------   0     ----> compare(20,20);
System.out.println(t) ;    // [20,15,10,0]
```

Various possible implementations of compare() method:

```
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        Integer I1=(Integer)obj1;
        Integer i2=(Integer)obj2;
        // return I1 CompareTo(I2);   [0,10,15,20] ascending order
        // return -I1 CompareTo(I2);  [20,15,10,0] descending order
        // return I2 CompareTo(I1);   [20,15,10,0] descending order
        // return -I2 CompareTo(I1);  [0,10,15,20] ascending order
        // return +1                  [10,0,15,20,20] Insertion order
        // return -1                  [20,20,15,0,10] Reverse of Insertion order
        // return 0;                  [10]
         (only first element will be inserted and all the other elements are considered as duplicates)
    }
}
```

Write a program to insert String objects into the TreeSet where sorting order is Reverse of Alphabetical order :

```
import java.util.*;
class TreeSetDemo2 {
  public static void main(String[] args) {
    TreeSet t = new TreeSet(new MyComparator());
    t.add("Roja");
    t.add("ShobhaRani");
    t.add("RajaKumari");
    t.add("GangaBhavani");
    t.add("Ramulamma");
    System.out.println(t);
  }
}
```

```
class MyComparator implements Comparator  {
    public int compare(Object obj1, Object obj2) {
        String s1 = obj1.toString();
        String s2 = (String)obj2;
        // return s2.compareTo(s1);
        return -s1.compareTo(s2);
    }
}
```

Output: [ShobhaRani, Roja, Ramulamma, RajaKumari, GangaBhavani]

Write a program to insert StringBuffer objects into the TreeSet where sorting order is Alphabetical order:

```
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);
    }
}
```

Output: [A, K, L, Z]

Note :

* If we are defining our own sorting by Comparator, the objects need not be Comparable.

Write a program to insert String and StringBuffer objects into the TreeSet where sorting order is increasing length order if two objects having the same length then consider their alphabetical order?

```
import java.util.*;
class TreeSetDemo12 {
  public static void main(String[] args) {
    TreeSet t = new TreeSet(new MyComparator());
    t.add("A");
    t.add(new StringBuffer("ABC"));
    t.add(new StringBuffer("AA"));
    t.add("XX");
    t.add("ABCD");
    t.add("A");
    System.out.println(t);
  }
}
```

Output: [A,AA,XX,ABC,ABCD]

```
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        int l1=s1.length();
        int l2=s2.length();
        if(l1<l2)
            return -1;
        else if(l1>l2)
            return 1;
        else
        return s1.compareTo(s2);
    }
}
```

Note :

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

2. But if we are defining our own sorting by comparator then objects need not be homogeneous and comparable. we can insert heterogeneous non comparable objects also.

1) For predefined Comparable classes like String default natural sorting order already available. If we are not satisfied with that, we can define our own sorting by Comparator object.

2) For Predefined non comparable classes like StringBuffer, default natural sorting order is not already available. We can define required sorting by implementing Comparator interface.

3) For our own classes like Employee, Student, Customer), the person who is writing our own class, he is responsible to define default natural sorting order by implementing Comparable interface.

The person who is using our class, if he is not satisfied with default natural sorting order, then he can define his own sorting by using Comparator.

Demo program for Customized sorting for Employee class : (Page 2 of 3)

```
class CompCompDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee("nag",100);
        Employee e2 = new Employee("balaiah",200);
        Employee e3 = new Employee("chiru",50);
        Employee e4 = new Employee("venki",150);
        Employee e5 = new Employee("nag",100);
        TreeSet t = new TreeSet();
        t.add(e1);
        t.add(e2);
        t.add(e3);
        t.add(e4);
        t.add(e5);
        System.out.println(t);

        TreeSet t1 = new TreeSet(new MyComparator());
        t1.add(e1);
        t1.add(e2);
        t1.add(e3);
        t1.add(e4);
        t1.add(e5);
        System.out.println(t1);
    }
}
```

Demo program for Customized sorting for Employee class : (Page 3 of 3)

```
class MyComparator implements Comparator {
   public int compare(Object obj1, Object obj2) {
         Employee e1 = (Employee)obj1;
         Employee e2 = (Employee)obj2;
         String s1 = e1.name;
         String s2 = e2.name;
         return s1.compareTo(s2);
   }
}
```