

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.

DURGASOFT

What is Collection?

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

What is Collection Framework?

It defines several classes and interfaces which can be used a group of objects as single entity.

DURGASOFT

Java

Collection

Collection Framework

C++

Container

STL (Standard Template Library)

DURGASOFT

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.

DURGASOFT

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.

DURGASOFT

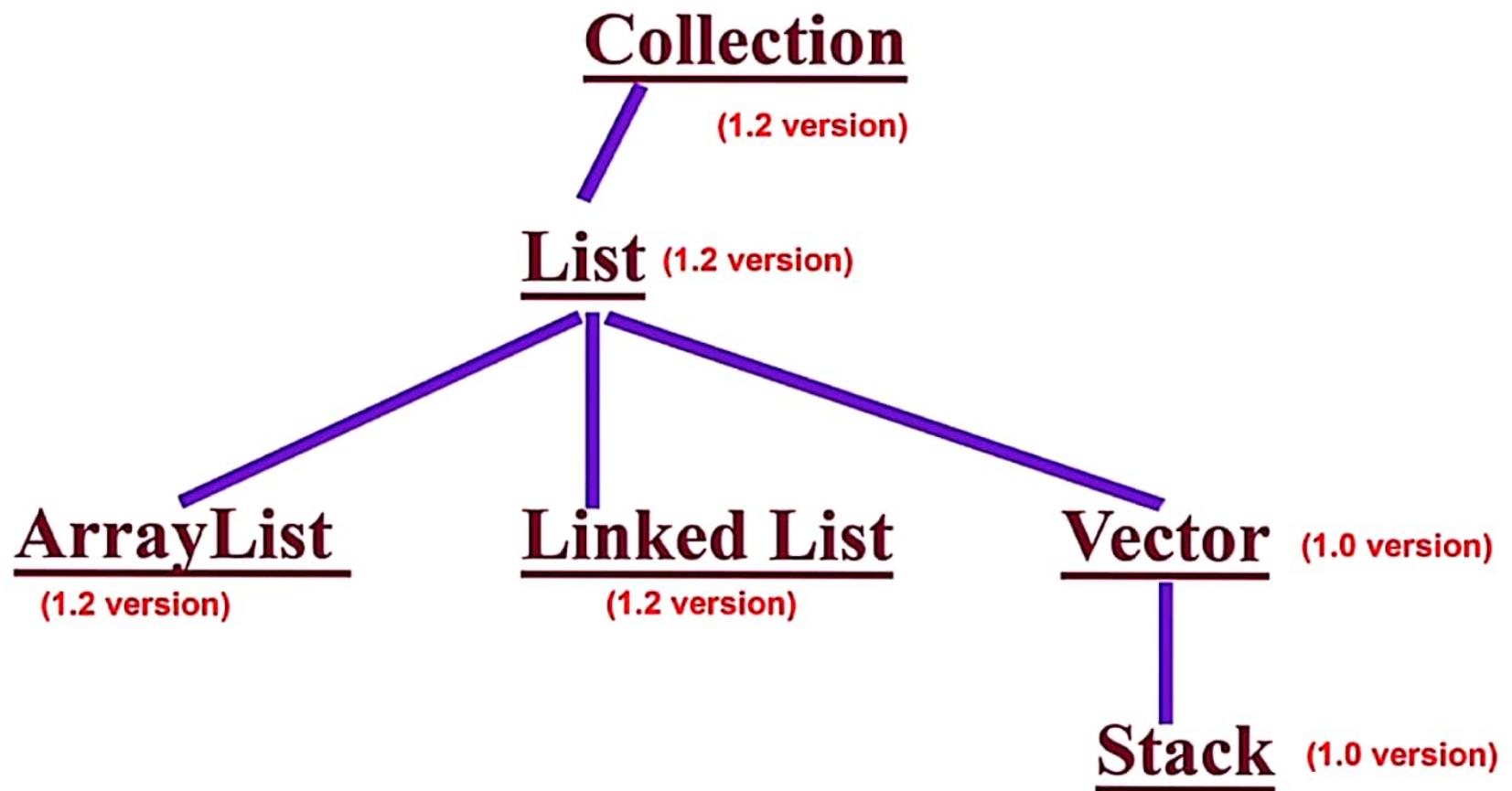
9 key interfaces of Collection Framework

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

ii. List :



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

9 key interfaces of Collection Framework

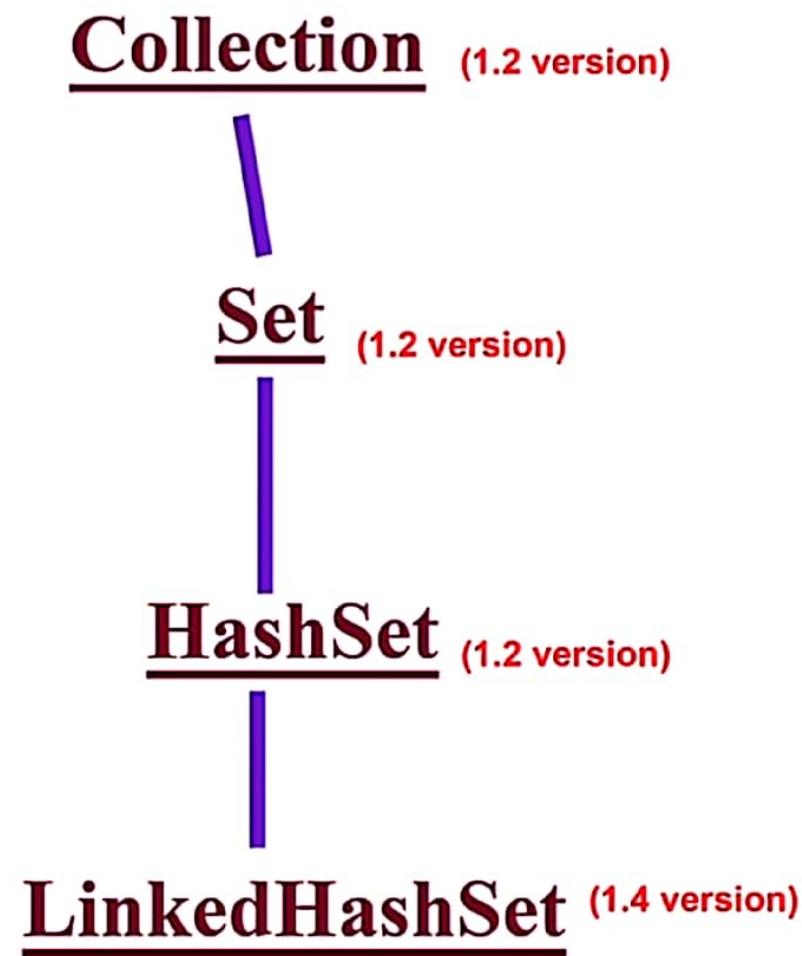
iii. Set:

- * 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.

DURGASOFT

9 key interfaces of Collection Framework

iii. Set :



DURGASOFT

Difference between List & Set

List

- * Duplicates are allowed
- * Insertion order preserved

Set

- * Duplicates are not allowed
- * Insertion order not preserved

DURGASOFT

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 order then we should go for SortedSet.

DURGASOFT

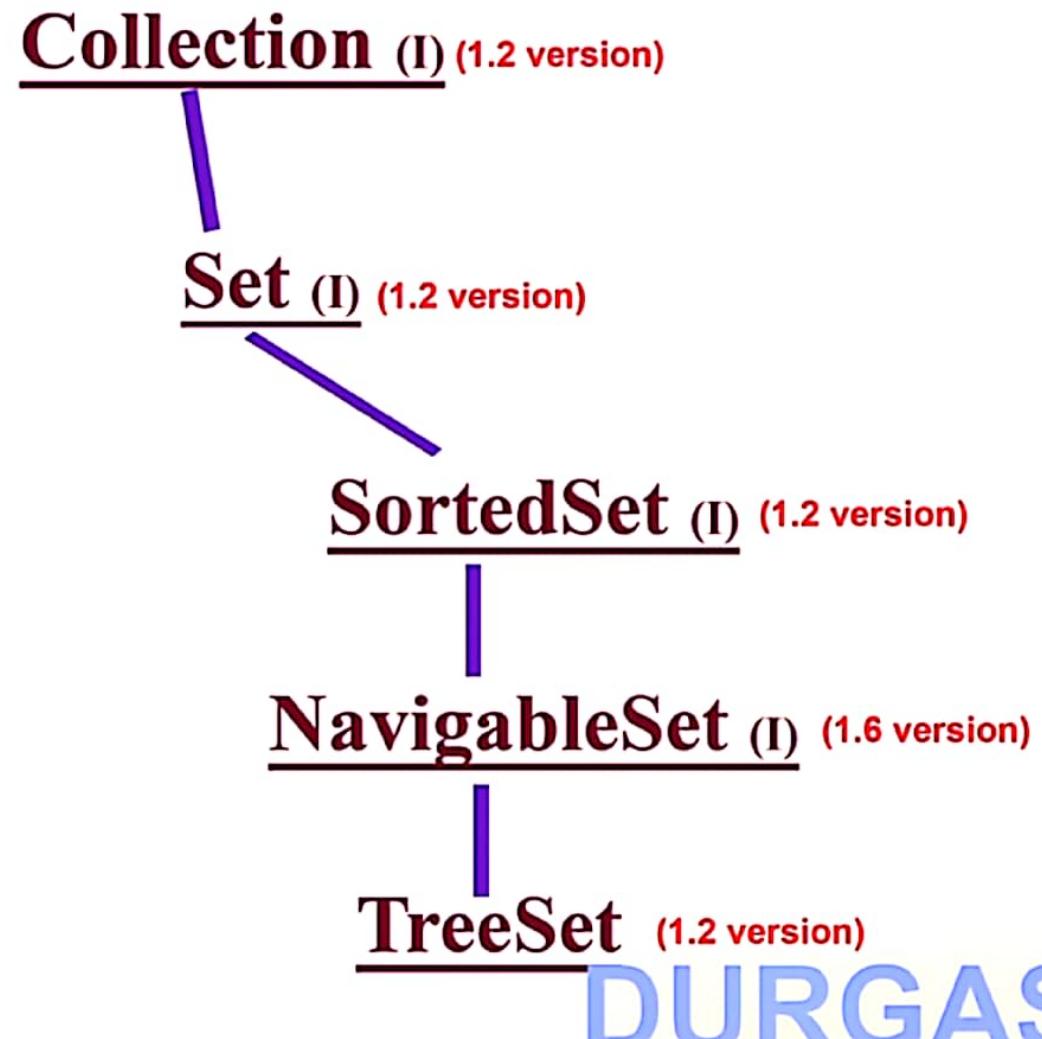
9 key interfaces of Collection Framework

v. NavigableSet:

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

DURGASOFT

9 key interfaces of Collection Framework



9 key interfaces of Collection Framework

vi. Queue :

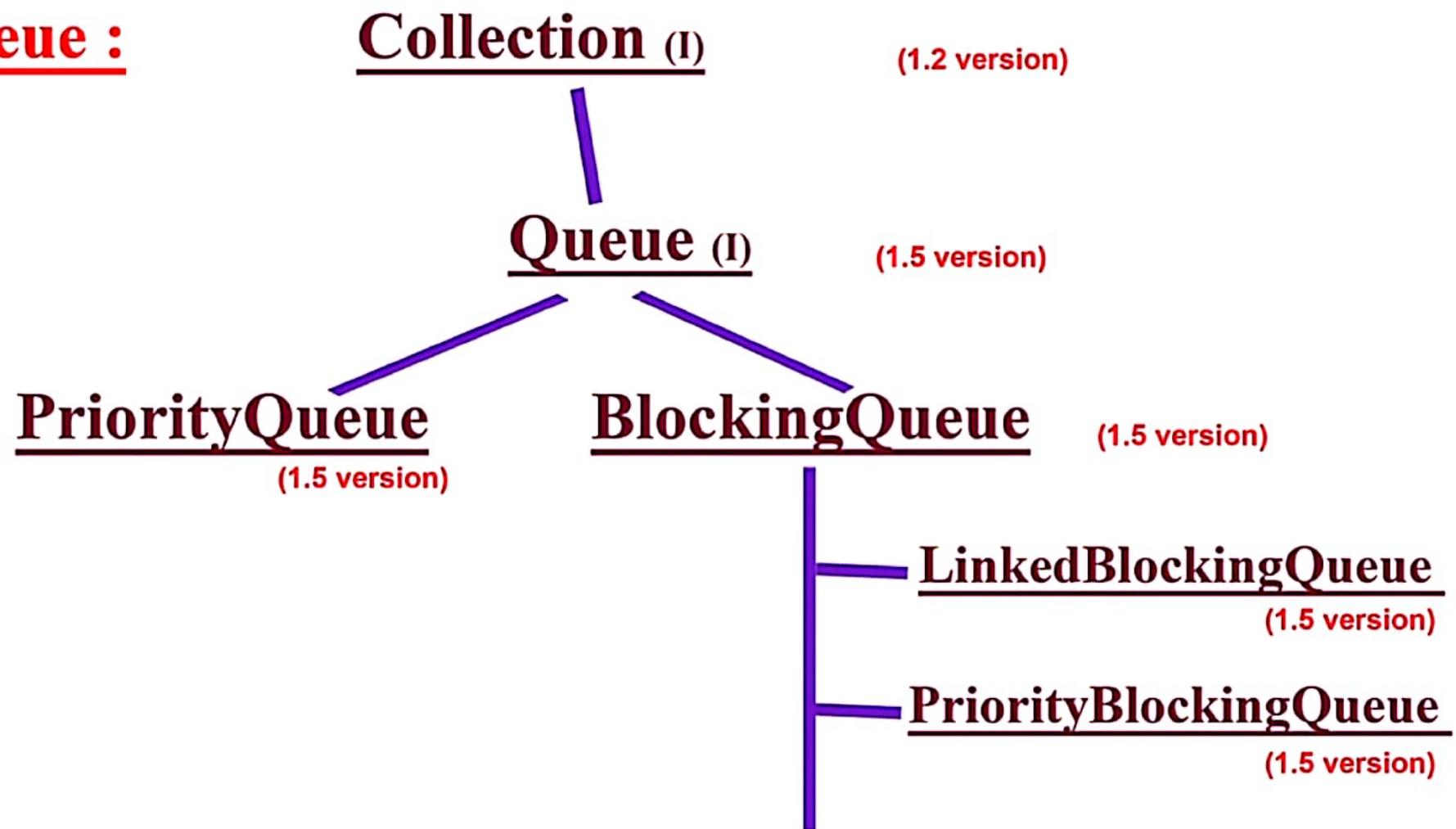
- * It is child interface of Collection.
- * If we want to represent a group of individual objects prior to processing then we should go for Queue.

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.

DURGASOFT

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.

DURGASOFT

9 key interfaces of Collection Framework

vii. Map:

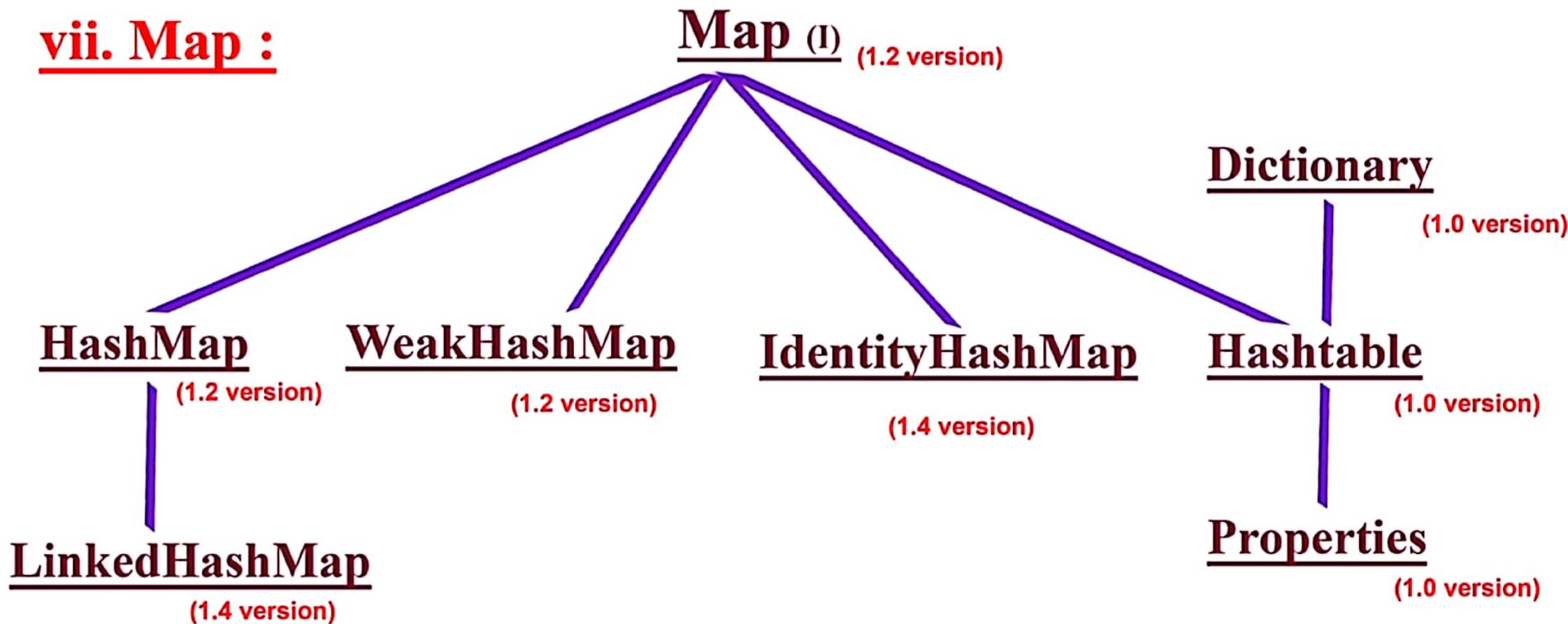
- * Map is not the child interface of Collection.
- * If we want to represent a group of individual objects as key value pairs then should go for Map.

Ex :	Roll No	Name
	101	Durga
	102	Ravi
	103	Venkat

Both key and value are objects, duplicated keys are not allowed but values can be duplicated

9 key interfaces of Collection Framework

vii. Map :



DURGASOFT

viii. SortedMap:

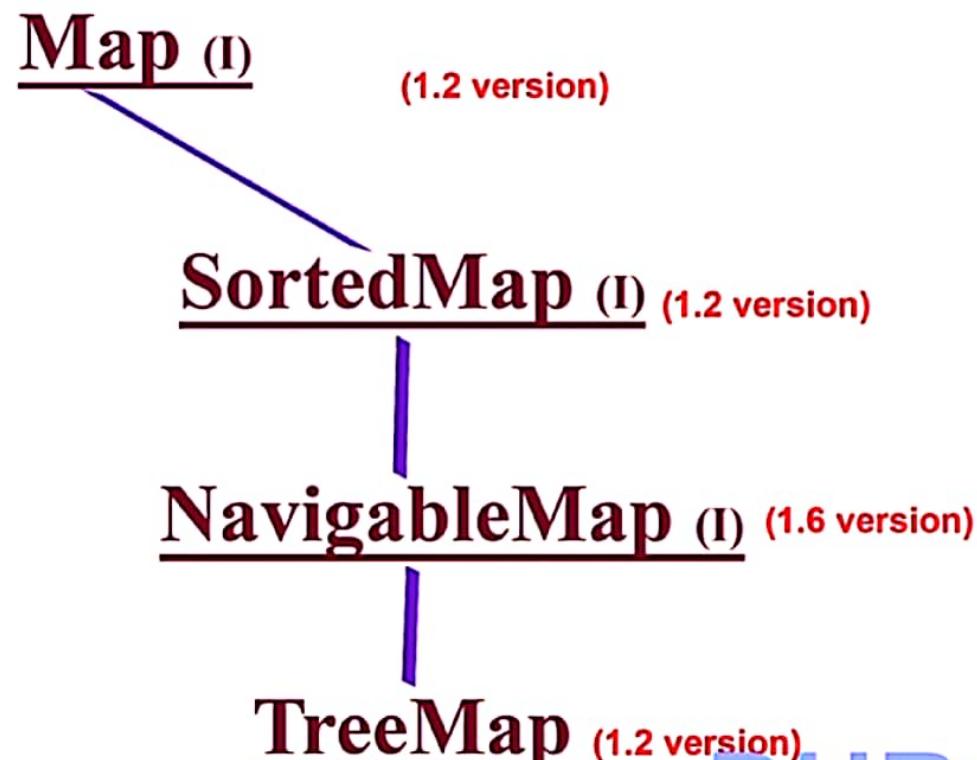
- * It is the child interface of map.
- * If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap



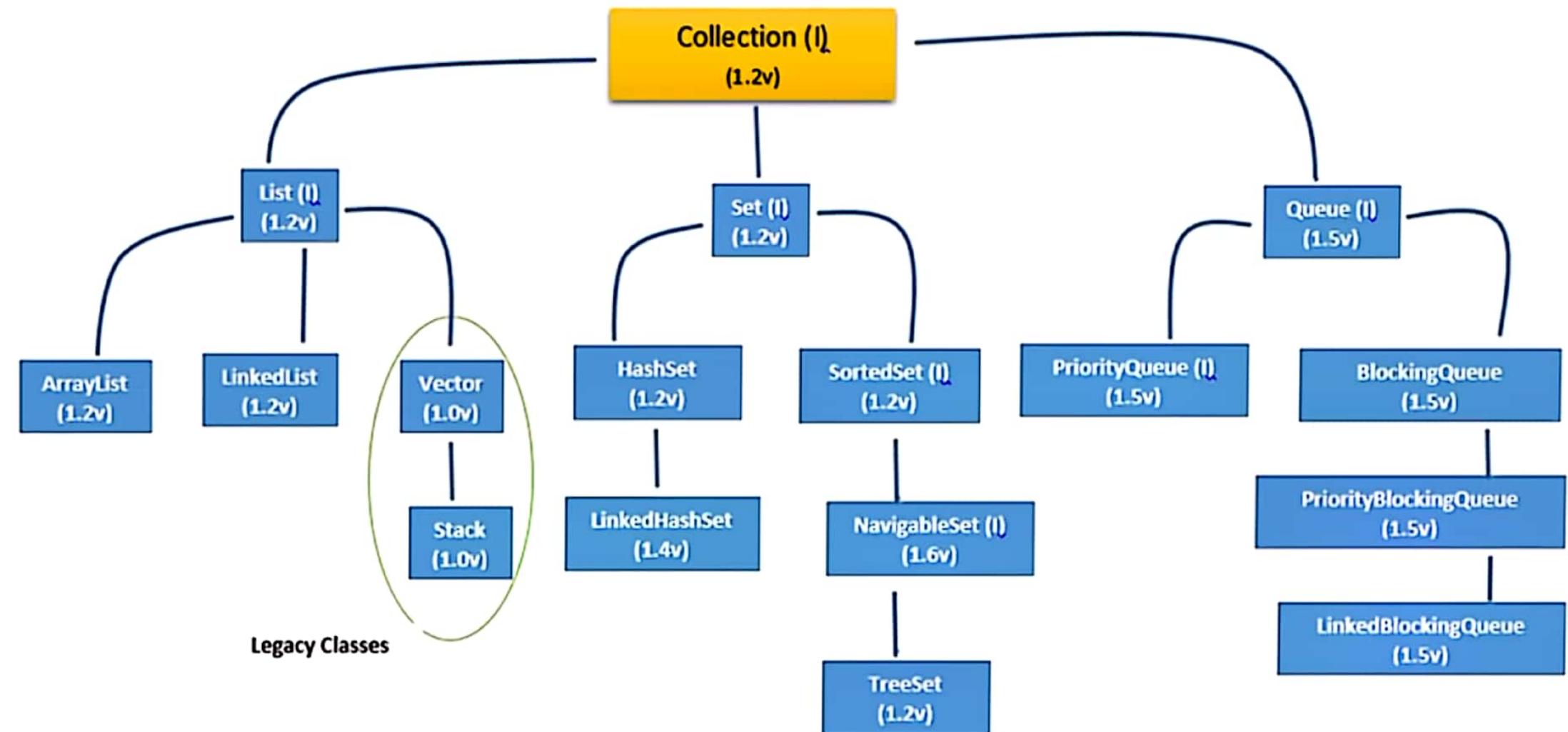
DURGASOFT

ix. NavigableMap:

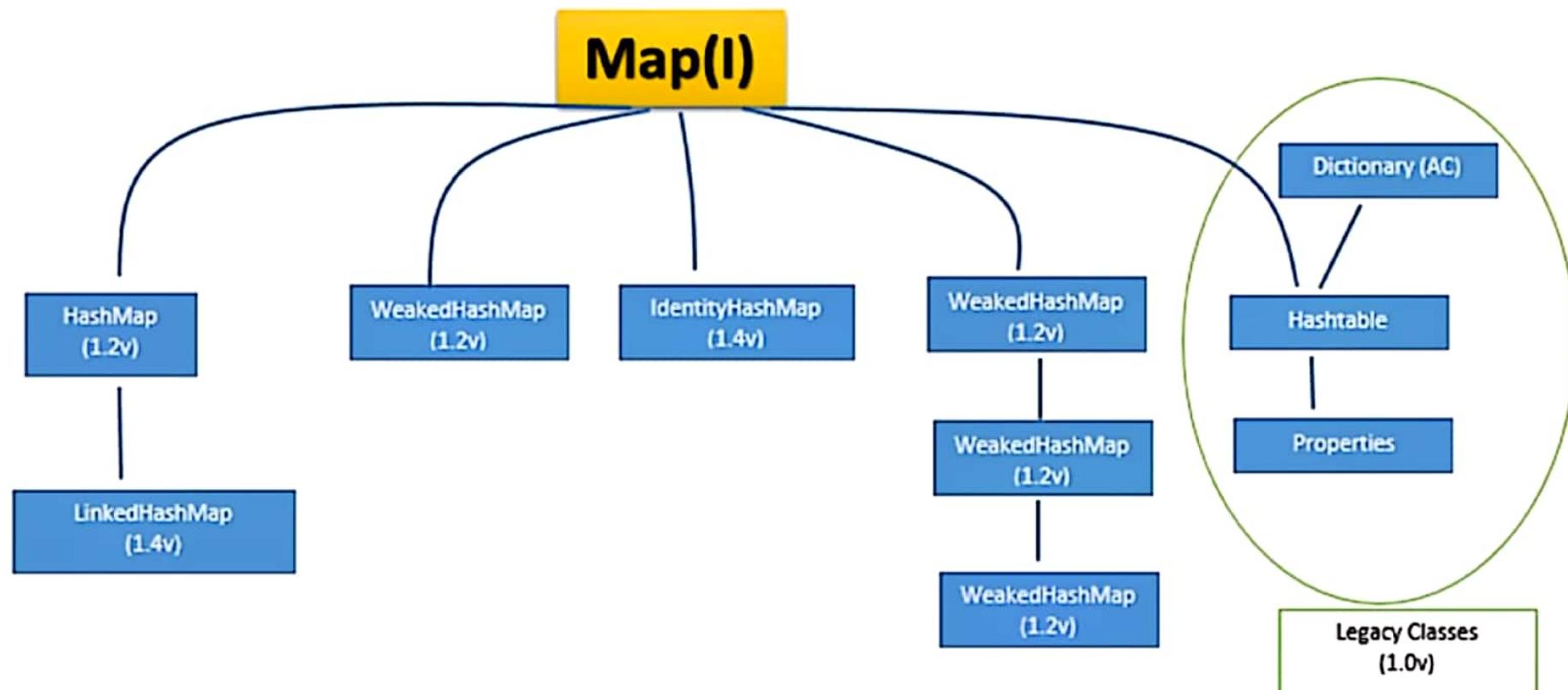
* It is the child interface of sorted map, it defines several utility methods for navigation purpose.



DURGASOFT



DURGASOFT



Sorting:

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

Cursors:

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

Utility Classes:

- 1. Collections
- 2. Arrays

DURGASOFT

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

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

```
175 boolean add(Object o)
176 boolean addAll(Collection c)
177 boolean remove(Object o)
178 boolean removeAll(Collection c)
179 boolean retainAll(Collection c)
180 To remove all objects except those
181 present in c
182 void clear()
183 boolean contains(Object o)
184 boolean containsAll(Collection c)
185 boolean isEmpty()
186 int size();
187 Object[] toArray();
188 Iterator iterator()
189
190
191
```

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.

DURGASOFT

1

2

3

4

5

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)

to replace the element present at specified index with provided Object and returns old object

int indexOf(Object o)

returns index of first occurrence of 'o'

int lastIndexOf(Object o)

ListIterator listIterator();

DURGASOFT

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();

DURGASOFT

ArrayList

- The underlined data structure Resizable Array or Growable Array
- Duplicates are allowed.
- Insertion order is preserved.
- Heterogeneous objects are allowed [except TreeSet & TreeMap everywhere heterogeneous objects are allowed].
- Null insertion is possible.

ArrayList Constructors

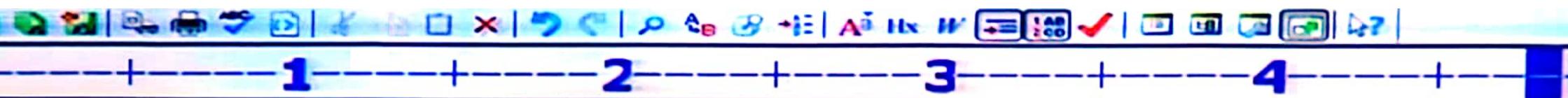
1. **ArrayList al = new ArrayList()**

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);**



```
I.add("A");
I.add(10);
I.add("A");
I.add(null);
System.out.println(I); // [A,10,A,null]
I.remove(2);
System.out.println(I); // [A,10,null]
I.add(2,"M");
I.add("N");
System.out.println(I); // [A,10,M,null,N]
}
```

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.**

RandomAccess

- * Present in `java.util` package.
- * It doesn't contain any methods and it is a Marker interface

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

DURGASOFT

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)

DURGASOFT

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);

DURGASOFT

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);

DURGASOFT

LinkedList

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

DURGASOFT

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.**

DURGASOFT

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();
```

DURGASOFT

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

DURGASOFT

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 (l1);//[ccc,venkey,software,30,null]
    }
}
```

DURGASOFT

Difference between ArrayList & LinkedList

<u>ArrayList</u>	<u>LinkedList</u>
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

DURGASOFT

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.

 DURGASOFT

Vector specific methods

For adding objects :

add (Object o)

[from Collection - List(l)]

add (int index, Object o)

[from List]

addElement (Object o)

[from Vector]

DURGASOFT

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 ();

DURGASOFT

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

DURGASOFT

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);
    }
}
```

DURGASOFT

Stack

- * It is a child class of Vector.
- * It is specially designed class for Last In First Out order(LIFO)

DURGASOFT

Constructor of Stack

Stack s=new Stack();

DURGASOFT

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

DURGASOFT

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]
    }
}
```

DURGASOFT

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

DURGASOFT

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 ();

DURGASOFT

Method of Enumeration

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

DURGASOFT

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]
}
```

DURGASOFT

```
1-----2-----3-----4-----E-----6-----7-----8-----9-----10-----11-----12-----13-----14-----15-----16-----17-----18-----19-----20-----21-----22-----23-----  
4 public static void main(String[] args)  
5 {  
6     Vector v = new Vector();  
7     for(int i = 0; i<=10; i++)  
8     {  
9         v.addElement(i);  
10    }  
11    System.out.println(v); // [0,1,2,3,...,10]  
12    Enumeration e = v.elements();  
13    while(e.hasMoreElements())  
14    {  
15        Integer I = (Integer)e.nextElement();  
16        if(I%2 == 0)  
17            System.out.println(I); // 0 2 4 6 8 10  
18    }  
19    System.out.println(v); // [0,1,2,3,... 10]  
20 }  
21 }  
22 }  
23 }
```

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.

DURGASOFT

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.

DURGASOFT

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

DURGASOFT

Methods in Iterator

- * **Iterator interface defines the following three methods.**
 - i. **public boolean hasNext ()**
 - ii. **public Object next()**
 - iii. **public void remove()**

DURGASOFT

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]  
}
```

DURGASOFT

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

DURGASOFT

Collection Framework

Three cursors of Java

3. ListIterator

**Mr. DURGA,
DURGASOFT**

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.

DURGASOFT

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

DURGASOFT

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)

DURGASOFT

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]  
    } }
```

DURGASOFT

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.

DURGASOFT

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)

DURGASOFT

Collection Framework

Implementation classes of
the three cursors

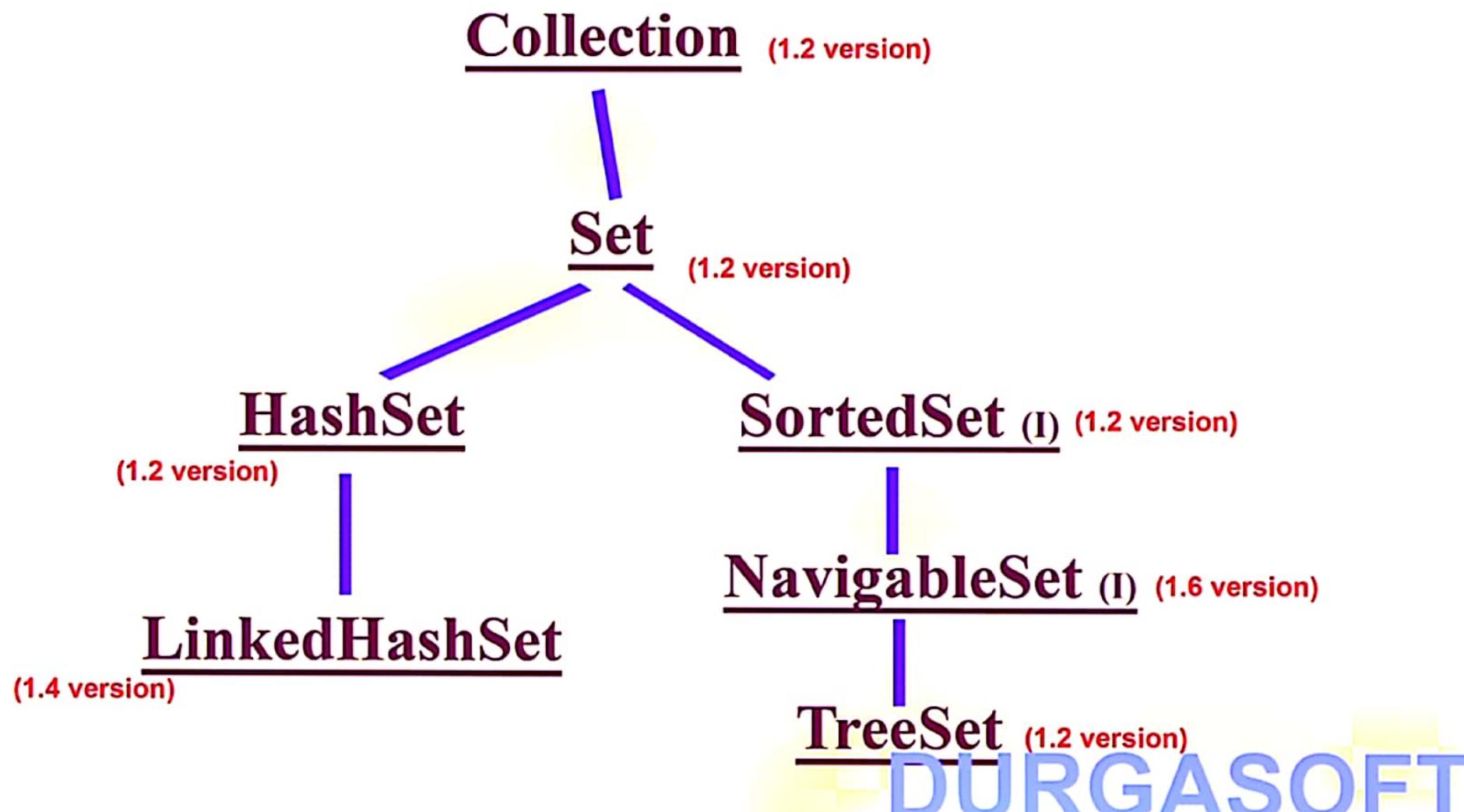
**Mr. DURGA,
DURGASOFT**

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.util.Vector$1  
    System.out.println (itr.getClass (). getName ()); // java.util.Vector$Iter  
    System.out.println (ltr.getClass (). getName ()); // java.util.Vector$ListIter  
}  
}
```

DURGASOFT

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.

DURGASOFT

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.

DURGASOFT

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 initialCapacity);**

- Creates an empty HashSet object with **specified** initial capacity & default Fill Retio **0.75**

3) **HashSet h = new HashSet(int initialCapacity, 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.

DURGASOFT

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]  
    } }
```

DURGASOFT

Collection Framework

LinkedHashSet **(Collection Framework)**

**Mr. DURGA,
DURGASOFT**

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.

DURGASOFT

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]  
    } }
```

DURGASOFT

LinkedHashSet

Note :

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

DURGASOFT

Collection Framework

SortedSet Details
[Collection Framework]

Mr. DURGA,
DURGASOFT

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.

Collection Framework

TreeSet Details [part- 1]

- * Constructors
- * Demo Program

**Mr. DURGA,
DURGASOFT**

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

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]
    }
}
```

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
NullPointerException.**

Collection Framework

TreeSet Details [part- 2]

- * Second demo program
- * Need of Comparable interface

**Mr. DURGA,
DURGASOFT**

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]
```

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();**

DURGASOFT

- * 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.

DURGASOFT

Collection Framework

TreeSet Details [part- 5]

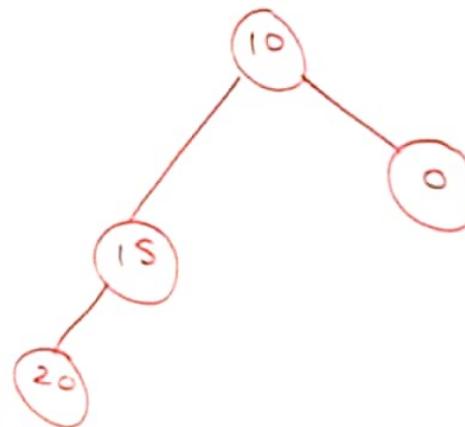
* Demo program for customized sorting

**Mr. DURGA,
DURGASOFT**

Q3: Integer objects into TreeSet, Descending order

```
TreeSet t = new TreeSet(new myComparator());  
t.add(10); ✓  
t.add(0), +ve compare(0, 10)
```

```
t.add(15); -ve compare(15, 10)  
t.add(20); -ve compare(20, 10)  
-ve compare(20, 15)  
t.add(20); -ve compare(20, 10)  
SOPA(+), [20, 15, 10, 0]
```



```
class myComparator implements Comparator  
{  
    public int compare(Object o1, Object o2)  
    {  
        Integer I1 = (Integer) o1;  
        Integer I2 = (Integer) o2;  
        if (I1 < I2)  
            return +1;  
        else if (I1 > I2)  
            return -1;  
        else  
            return 0;  
    }  
}
```

DURGASOFT

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);
    }
}
```

Output: [20,15,10,0]

```
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 -1;
        else
            return 0;
    }
}
```

DURGASOFT

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]

DURGASOFT

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)  
    }  
}
```

DURGASOFT

Collection Framework

TreeSet Details [part- 6]

* Demo program-2 for customized sorting

**Mr. DURGA,
DURGASOFT**

Collection Framework

TreeSet Details [part- 6]

* Demo program-2 for customized sorting

**Mr. DURGA,
DURGASOFT**

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]

DIGI G A S O F T

Collection Framework

TreeSet Details [part- 7]

* Demo program-3 for customized sorting

**Mr. DURGA,
DURGASOFT**

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]

DURGASOFT

Note :

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

DURGASOFT

Collection Framework

TreeSet Details [part- 8]

* Demo program-4 for customized sorting

**Mr. DURGA,
DURGASOFT**

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);
    }
}
```

DURGASOFT

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.

DURGASOFT

Collection Framework

TreeSet Details [part- 9]

* Comparable vs Comparator interface

**Mr. DURGA,
DURGASOFT**

- 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.

DURGASOFT

pre-defined
Comparable
classes
(e.g. String)



Comparable

Pre-defined
non-Comparable
classes
(StringBuffer)



Comparable

Comparable(I)
Default Natural Sorting
order.

The person who is
writing class

our own
classes
(Employee)

The person
who is using
our class

Comparator(I)

Customized Sorting

DURGASOFT

Collection Framework

TreeSet Details [part- 10]

- * Demo program for customized sorting of employee class

**Mr. DURGA,
DURGASOFT**

Demo program for Customized sorting for Employee class : (Page 1 of 3)

```
import java.util.*;
class Employee implements Comparable {
    String name;
    int eid;
    Employee(String name,int eid) {
        this.name = name;
        this.eid = eid;
    }
    public String toString() {
        return name+"--"+eid;
    }
    public int compareTo(Object obj) {
        int eid1 = this.eid;
        Employee e = (Employee)obj;
        int eid2 = e.eid;
        if(eid1 < eid2) {
            return -1;
        } else if (eid1 > eid2) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

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);  
    }  
}
```

Comparison table of Comparable and Comparator

Comparable	Comparator
1. It is meant for default natural sorting order.	1. It is meant for customized sorting order.
2. Present in <code>java.lang</code> package .	2. Present in <code>java.util</code> package
3. This interface defines only one method <code>compareTo()</code> .	3. This interface defines two methods <code>compare()</code> and <code>equals()</code> .
4. All wrapper classes and <code>String</code> class implement <code>comparable</code> interface.	4. The only implemented classes of <code>Comparator</code> are <code>Collator</code> and <code>RuleBasedCollator</code> .

Comparison table of Set(I) implemented classes

Property	HashSet	LinkedHashSet	TreeSet
1. Underlying Data Structure	Hashtable	Hashtable + LinkedList	Balanced Tree
2. Insertion Order	Not Preserved	Preserved	Not Applicable
3. Sorting order	Not Applicable	Not Applicable	Applicable
4. Heterogeneous objects	Allowed	Allowed	Not Allowed
5. Duplicate Objects	Not Allowed	Not Allowed	Not Allowed
6. Null Acceptance	Allowed (only once)	Allowed(only once)	For Empty TreeSet as first element Null is allowed and in all other cases we will get NullPointerException

DURGASOFT

1

2

3

4

5

new, delete
goto, constant
break, continue, return, exit
final, finally, finalize
throw, throws, thrown,
notify, notifyAll
implements, extends, imports
sizeof, instanceof
instanceOf, strictFp
byte, short, Int
None of the above

DURGASOFT
www.durgasoft.com

1 2 3 4 5

new, delete
goto, constant
break, continue, return, exit
final, finally, finalize
throw, throws, thrown,
notify, notifyAll
implements, extends, imports
sizeof, instanceof
instanceOf, strictfp
byte, short, Int
None of the above

DURGASOFT
www.durgasoft.com