Unit II-1D and 2D Collections

1D Collections:

Introduction:

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

Limitations of Object[] array:

1. 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 possiblealways.
2. Arrays can hold only homogeneous data elements.

Example:

```
Student[] s=new Student[10000];
s[0]=new Student();//valid
s[1]=new Customer();//invalid(compile time error)
```

Compile time error:

```
Test.java:7: cannot find symbol
Symbol: class Customer Location:
class Test
s[1]=new Customer();
```

3) But we can resolve this problem by using object type array(Object[]).

Example:

```
Object[] o=new Object[10000];
o[0]=new Student();
o[1]=new Customer();
```

4) 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 collections concept.

1. Collections are growable in nature that is based on our requirement we can increase (or) decrease the size hence memory point of view collections concept isrecommended to use.
2. Collections can hold both homogeneous and heterogeneous objects.
3. 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 functionalityon our own.

Differences between Arrays and Collections?

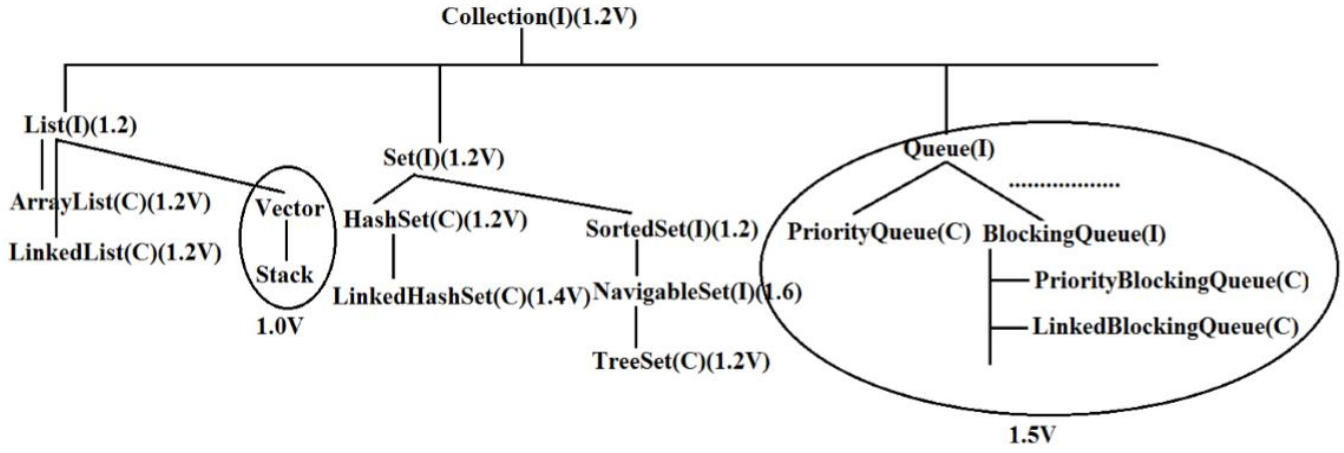
Arrays	Collections
1) Arrays are fixed in size.	1) Collections are growable in nature.
2) Memory point of view arrays are notrecommended to use.	2) Memory point of view collections are highly recommended to use.
3) Performance point of view arrays arerecommended to use.	3) Performance point of view collections 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 ready made method support is available.
6) Arrays can hold both primitives and object types.	6) Collections can hold only objects but not primitives.

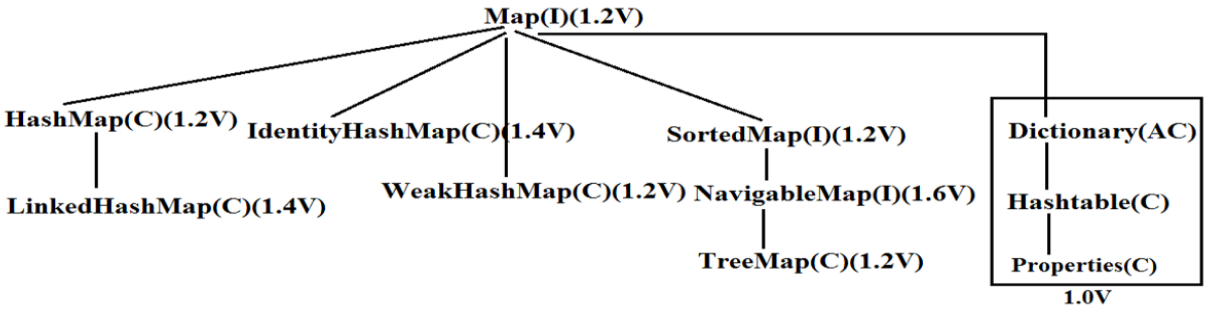
Collection:

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

Collection framework:

It defines several classes and interfaces to represent a group of objects as a single entity.





9(Nine) key interfaces of collection framework:

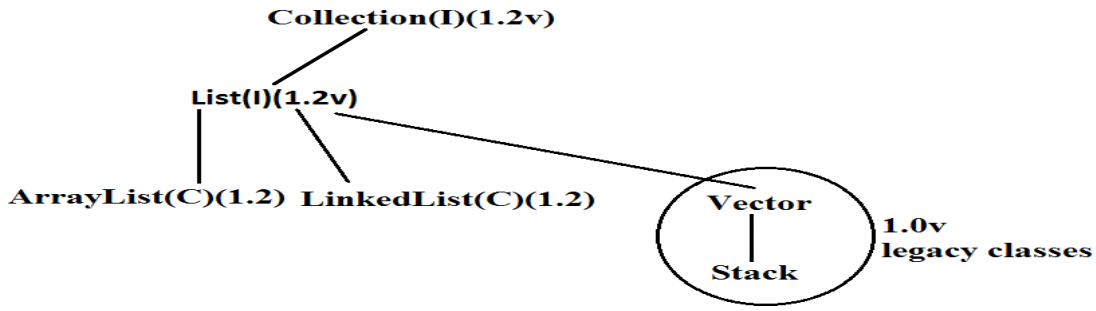
1. Collection
2. List
3. Set
4. SortedSet
5. NavigableSet
6. Queue
7. Map
8. SortedMap
9. NavigableMap

Collection:

1. If we want to represent a group of "individual objects" as a single entity then weshould go for collection.
2. In general we can consider collection as root interface of entire collectionframework.
3. Collection interface defines the most common methods which can be applicablefor any collection object.
4. There is no concrete class which implements Collection interface directly.

List:

1. It is the child interface of Collection.
2. If we want to represent a group of individual objects as a single entity where "duplicates are allow and insertion order must be preserved" then we should gofor List interface.

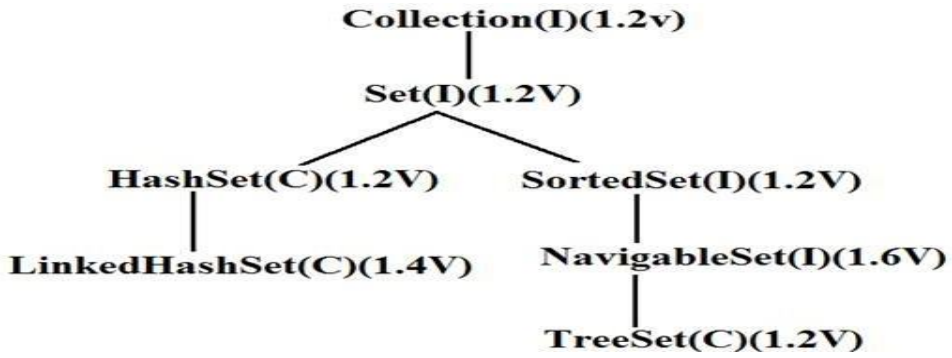


Vector and Stack classes are re-engineered in 1.2 versions to implement List interface.

Set:

1. It is the child interface of Collection.
2. If we want to represent a group of individual objects as single entity "where duplicates are not allow and insertion order is not preserved" then we should gofor Set interface.

SortedSet:



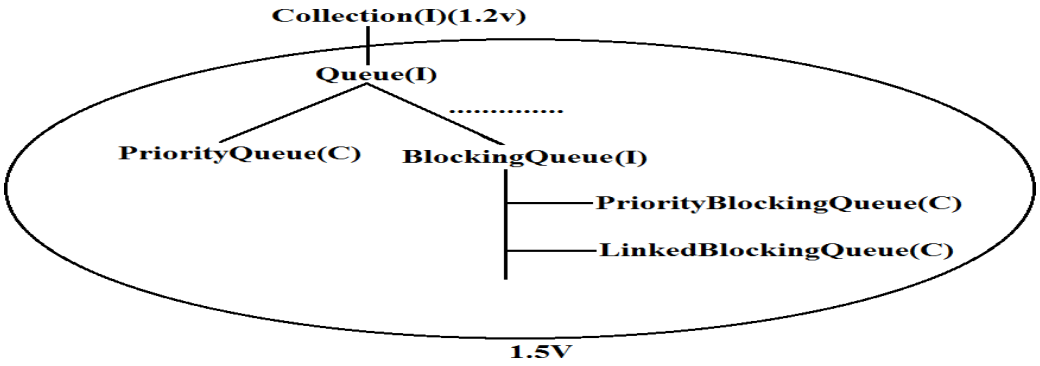
1. It is the child interface of Set.
2. If we want to represent a group of individual objects as single entity "where duplicates are not allowed but all objects will be insertion according to some sortingorder then we should go for SortedSet.
   (or)
3. If we want to represent a group of "unique objects" according to some sortingorder then we should go for SortedSet.

NavigableSet:

1. It is the child interface of SortedSet.
2. It provides several methods for navigation purposes.

Queue:

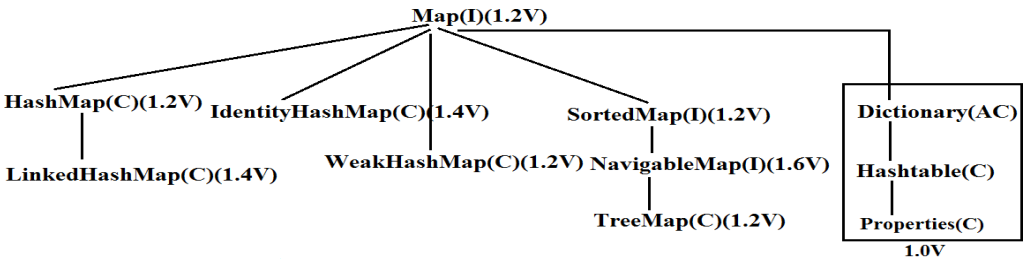
1. It is the child interface of Collection.
2. If we want to represent a group of individual objects prior to processing then weshould go for queue concept.



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.

Map:

1. Map is not child interface of Collection.
2. If we want to represent a group of objects as key-value pairs then we should go forMap interface.
3. Duplicate keys are not allowed but values can be duplicated.



SortedMap:

1. It is the child interface of Map.
2. If we want to represent a group of objects as key value pairs "according to somesorting order of keys" then we should go for SortedMap.

NavigableMap:

1) It is the child interface of SortedMap and defines several methods for navigation purposes.

<u>What is the difference between Collection and Collections ?</u>

"Collection is an "interface" which can be used to represent a group of objects as a single entity. Whereas "Collections is an utility class" present in java.util package to define several utility methods for Collection objects.

Collection----------interface
Collections ----------class

In collection framework the following are legacy characters.

1. Enumeration(I)
2. Dictionary(AC)
3. Vector(C)
4. Stack(C)
5. Hashtable(C)
6. Properties(C)

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.
    1. **boolean add(Object o):** It is used to insert an element in this collection.
    2. **boolean addAll(Collection c)**: It is used to insert the specified collection elements in the invoking collection.

3. **boolean remove(Object o):** It is used to delete an element from the collection.
4. **boolean removeAll(Collection c):** It is used to delete all the elements of the specified collection from the invoking collection.
5. **boolean retainAll(Collection c):** It retains only those elements which are in c and removes the other elements from the collection. It returns true if the elements were removed, else returns false.
6. **Void clear();**It removes the total number of elements from the collection.
7. **boolean contains(Object o):** Checks if the object is present in the collection and returns true if found. Else, it returns false.
8. **boolean containsAll(Collection c):** Returns true if the collection contains all the elements of c, else returns false.
9. **boolean isEmpty();**It checks if collection is empty. It returns true if the list is empty, otherwise false
10. **Int size();**It returns the total number of elements in the collection.
11. **Object[] toArray();**It converts collection into array.
12. **Iterator iterator();**It returns an iterator.

Note: There is no concrete class which implements Collection interface 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 allow and insertion order is preserved. Then we should go for List.
- We can differentiate duplicate objects and we can maintain insertion order by means of index hence "index play very important role in List"

List interface defines the following specific methods.

1. **boolean add(int index,Object o):** It is used to insert the specified element at the specified position in a list.
2. **boolean addAll(int index,Collectio c)**: It is used to append all the elements in the specified collection, starting at the specified position of the list.
3. **Object get(int index):**It is used to fetch the element from the particular position of the list.
4. **Object remove(int index):**I t is used to remove the element present at the specified position in the list.
5. **Object set(int index,Object new):**//to replace
6. **Int indexOf(Object o):** It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
   Returns index of first occurrence of "o".
7. **Int lastIndexOf(Object o):**It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
8. **ListIterator listIterator();**

ArrayList:

- An ArrayList is a re-sizable array, also called a dynamic array. It grows its size to accommodate new elements and shrinks the size when the elements are removed.
- ArrayList internally uses an array to store the elements. Just like arrays, It allows you to retrieve the elements by their index.
- ArrayList allows duplicate and null values.
- ArrayList is an ordered collection. It maintains the insertion order of the elements.
- You cannot create an ArrayList of primitive types like int, char etc. You need to use boxed types like Integer, Character, Boolean etc.
- ArrayList is not synchronized. If multiple threads try to modify an ArrayList at the same time, then the final outcome will be non-deterministic. You must explicitly synchronize access to an ArrayList if multiple threads are gonna modify it.

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. That is to dance between collection objects.

Demo program for ArrayList:

```
import java.util.*;
class A r rayListDemo
{
        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]
         }
}
```

Non-generic Vs. Generic Collection

- Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.
- Java new generic collection allows you to have only one type of object in a collection. Now it is type-safe, so typecasting is not required at runtime.
- Let's see the old non-generic example of creating a Java collection.
  ArrayList list=new ArrayList();//creating old non-generic arraylist
- Let's see the new generic example of creating java collection.
  ArrayList<String> list=new ArrayList<String>();//creating new generic arraylist
- In a generic collection, we specify the type in angular braces. Now ArrayList is forced to have the only specified type of object in it. If you try to add another type of object, it gives a compile-time error.

LinkedList:

Similar to arrays in Java, LinkedList is a linear data structure. However LinkedList elements are not stored in contiguous locations like arrays, they are linked with each other using pointers. Each element of the LinkedList has the reference(address/pointer) to the next element of the LinkedList.

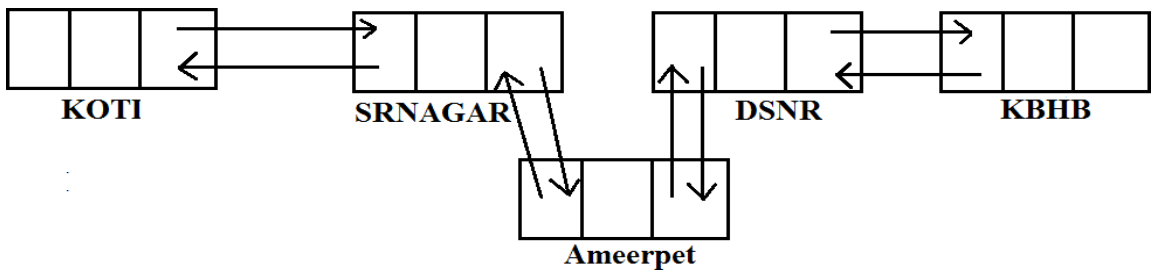
LinkedList representation

Each element in the LinkedList is called the Node. Each Node of the LinkedList contains two items: 1) Content of the element 2) Pointer/Address/Reference to the Next Node in the LinkedList.

Points to Note about LinkedList:

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

Diagram:



Usually we can use LinkedList to implement Stacks and Queues.

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

1. **void addFirst(Object o):** It adds the item (or element) at the first position in the list.
2. **void addLast(Object o):** It inserts the specified item at the end of the list.
3. **Object getFirst()**: It fetches the first item from the list.
4. **Object getLast():**It fetches the last item from the list.
5. **Object removeFirst():**It removes and return the first item from the list.
6. **Object removeLast():**It removes and return the last item of the list.

We can apply these methods only on LinkedList object.

Constructors:

1. LinkedList l=new LinkedList();
   Creates an empty LinkedList object.
2. LinkedList l=new LinkedList(Collection c);
   To create an equivalent LinkedList object for the given collection.

Example:

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

Why do we need a Linked List?

You must be aware of the arrays which is also a linear data structure but arrays have certain limitations such as:

1) Size of the array is fixed which is decided when we create an array so it is hard to predict the number of elements in advance, if the declared size fall short then we cannot increase the size of an array and if we declare a large size array and do not need to store that many elements then it is a waste of memory.

2) Array elements need contiguous memory locations to store their values.

3) Inserting an element in an array is performance wise expensive as we have to shift several elements to make a space for the new element. For example:
Let's say we have an array that has following elements: 10, 12, 15, 20, 4, 5, 100, now if we want to insert a new element 99 after the element that has value 12 then we have to shift all the elements after 12 to their right to make space for new element.

Similarly deleting an element from the array is also a performance wise expensive operation because all the elements after the deleted element have to be shifted left.

These limitations are handled in the Linked List by providing following features:

1. Linked list allows dynamic memory allocation, which means memory allocation is done at the run time by the compiler and we do not need to mention the size of the list during linked list declaration.

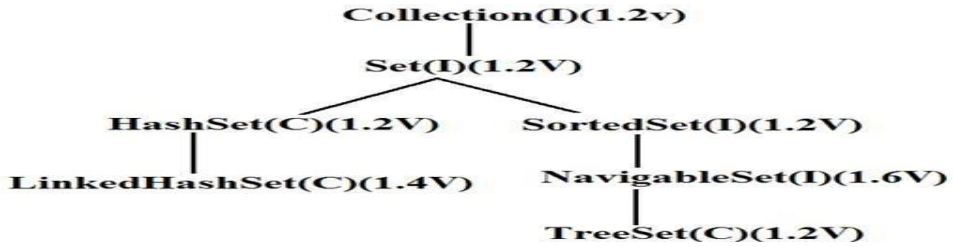
2. Linked list elements don't need contiguous memory locations because elements are linked with each other using the reference part of the node that contains the address of the next node of the list.

3. Insert and delete operations in the Linked list are not performance wise expensive because adding and deleting an element from the linked list does't require element shifting, only the pointer of the previous and the next node requires change.

Set interface:

1. It 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 allow and insertion order is not preserved then we should go forSet interface.

Diagram:



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

HashSet:

- HashSet **internally uses Hashtable** data structure.
- HashSet **doesn't maintain any order**, the elements would be returned in any random order.
- HashSet **doesn't allow duplicates**. If we are trying to insert duplicate objects we won't get compile time error and runtime error add() method simply returns false.
- HashSet **allows null value only once**, however if you insert more than one nulls, it would override the previous null value.
- HashSet **allows** Heterogeneous objects.
- HashSet is best suitable, if our frequent operation is "Search".
- HashSet Implements Serializable and Cloneable interfaces but not RandomAccess.

Constructors:

1. **HashSet h=new HashSet();**
   Creates an empty HashSet object with default initial capacity 16 and default fillratio 0.75(fill ratio is also known as load factor).
2. **HashSet h=new HashSet(int initialcapacity);**
   Creates an empty HashSet object with the specified initial capacity and default fillratio 0.75.
3. **HashSet h=new HashSet(int initialcapacity,float fillratio);**
   Creates an empty HashSet object with the specified initial capacity and fillratio.
4. **HashSet h=new HashSet(Collection c);**
   This doesn't create an empty HashSet. It creates the HashSet with the elements copied from the passed Collection.

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:

LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It contains only unique elements like HashSet. Linked HashSet also provides all optional set operations and maintains insertion order.

- It is the child class of HashSet
- LinkedHashSet **internally uses LinkedList and Hashtable** data structure.
- LinkedHashSet **maintains the insertion order**. Elements gets sorted in the same sequence in which they have been added to the Set.
- LinkedHashSet **doesn't allow duplicates**. If we are trying to insert duplicate objects we won't get compile time error and runtime error add() method simply returns false.
- LinkedHashSet **allows null value only once**, however if you insert more than one nulls, it would override the previous null value.
- LinkedHashSet **allows** Heterogeneous objects.

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.

Constructors:

1. **LinkedHashSet h=new LinkedHashSet();**
   Creates an empty LinkedHashSet object with default initial capacity 16 and default fillratio 0.75(fill ratio is also known as load factor).

2. **LinkedHashSet h=new LinkedHashSet(int initialcapacity);**
   Creates an empty LinkedHashSet object with the specified initial capacity and default fillratio 0.75.
3. **LinkedHashSet h=new LinkedHashSet(int initialcapacity,float fillratio);**
   Creates an empty LinkedHashSet object with the specified initial capacity and fillratio.
4. **LinkedHashSet h=new LinkedHashSet(Collection c);**
   This doesn't create an empty LinkedHashSet. It creates the LinkedHashSet with the elements copied from the passed Collection.

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:

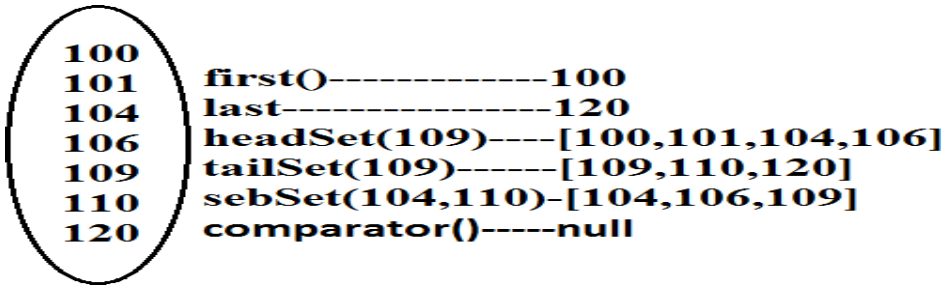
1. It is child interface of Set.
2. If we want to represent a group of "unique objects" where duplicates are not allowed and all objects must be inserting according to some sorting order then weshould go for SortedSet interface.
3. That sorting order can be either default natural sorting (or) customized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

SortedSet interface define the following 6 specific methods.

1. **Object first():**Returns the first (lowest) element in the set.
2. **Object last():**Returns the last (highest) element in the set.
3. **SortedSet headSet(Object obj):**
   Returns the SortedSet whose elements are <obj.

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

Diagram:



Example for SortedSet:

```
import java.util.SortedSet;
import java.util.TreeSet;
public class SortedSetExample {
   public static void main(String[] args) {
      // Create a sorted set using the TreeSet implementation
      SortedSet<Integer> set = new TreeSet<>();
      // Add elements to the set
      set.add(10);
      set.add(20);
      set.add(30);
      set.add(40);
      // Print the elements of the set
      System.out.println(set); // Output: [10, 20, 30, 40]
      // Get the first element in the set
      System.out.println(set.first()); // Output: 10
      // Get the last element in the set
      System.out.println(set.last()); // Output: 40
      // Get a view of the portion of the set that is less than a given element
      System.out.println(set.headSet(30)); // Output: [10, 20]
      // Get a view of the portion of the set that is greater than or equal to a given element
      System.out.println(set.tailSet(30)); // Output: [30, 40]
      // Get a view of the portion of the set that is between two given elements
      System.out.println(set.subSet(20,40)); // Output: [20, 30]
   }
}
```

NavigableSet

NavigableSet is an interface that inherits the SortedSet interface, which extends the Set interface. All of these are available in the java.util package. The SortedSet interface is implemented by the TreeSet class.

The NavigableSet has all the properties of the SortedSet and also adds a feature with navigation properties with the help of navigation methods. The NavigationSet is useful when we need to navigate throughout a SortedSet in a particular manner with the help of the supporting methods, in order to retain the desired output.

NavigableSet interface defines the following methods.

1. **ceiling(e)**: It returns the lowest element which is >=e.
2. **higher(e):** It returns the lowest element which is >e.
3. **floor(e):** It returns highest element which is <=e.
4. **lower(e):** It returns height element which is <e.
5. **pollFirst ():** Remove and return 1st element.
6. **pollLast ():** Remove and return last element.
7. **descendingSet ()**:Returns SortedSet in reverse order.

Syntax: NavigableSet<dataType> name = new TreeSet<dataType>();

Example:

```
import java.util.NavigableSet;
import java.util.TreeSet;
public class NavigableSetExample {
   public static void main(String[] args) {
      // Create a navigable set using the TreeSet implementation
      NavigableSet<Integer> set = new TreeSet<>();
      set.add(10);
      set.add(20);
      set.add(30);
      set.add(40);
      System.out.println(set); // Output: [10, 20, 30, 40]
      // Get the first element in the set
      System.out.println(set.first()); // Output: 10
      // Get the last element in the set
      System.out.println(set.last()); // Output: 40
      // Get the greatest element less than or equal to a given element
      System.out.println(set.floor(25)); // Output: 20
      // Get the least element greater than or equal to a given element
      System.out.println(set.ceiling(35)); // Output: 40
      // Get a reverse order view of the set
     System.out.println(set.descendingSet()); // Output: [40, 30, 20, 10]
    }}
```

TreeSet:

TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

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

Internal Working of The TreeSet Class

TreeSet is being implemented using a binary search tree, which is self-balancing just like a Red-Black Tree. Therefore, operations such as a search, remove, and add consume O(log(N)) time. The reason behind this is there in the self-balancing tree. It is there to ensure that the tree height never exceeds O(log(N)) for all of the mentioned operations. Therefore, it is one of the efficient data structures in order to keep the large data that is sorted and also to do operations on it.

Constructors:

1. **TreeSet t=new TreeSet():**
   Creates an empty TreeSet object where all elements will be inserted according todefault natural sorting order.
2. **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.
3. **TreeSet t=new TreeSet(SortedSet s):**
   It is used to build a TreeSet that contains the elements of the given SortedSet.
4. **TreeSet t=new TreeSet(Collection c):**
   It is used to build a new tree set that contains the elements of the collection c.

Example:

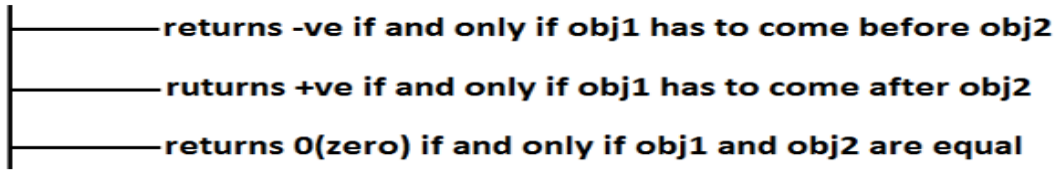
```
import java.util.*;
class TreeSetDemo
{
        public static void main(String[] args)
        {
                TreeSet t=new TreeSet();
                t.add("A");
                t.add("a");
                t.add("B");
                t.add("Z");
                t.add("L");
                //t.add(new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

Comparable Interface:

Comparable is an interface which defines a way to compare an object with other objects of the same type. It helps to sort the objects that have self-tendency to sort themselves, i.e., the objects must know how to order themselves. Eg: Roll number, age, salary. This interface is found in java.lang package and it contains only one method, i.e., compareTo(). Comparable is not capable of sorting the objects on its own, but the interface defines a method int compareTo() which is responsible for sorting.

compareTo ():

This method is used to compare the given object with the current object. The compareTo() method returns an int value. The value can be either positive, negative, or zero.



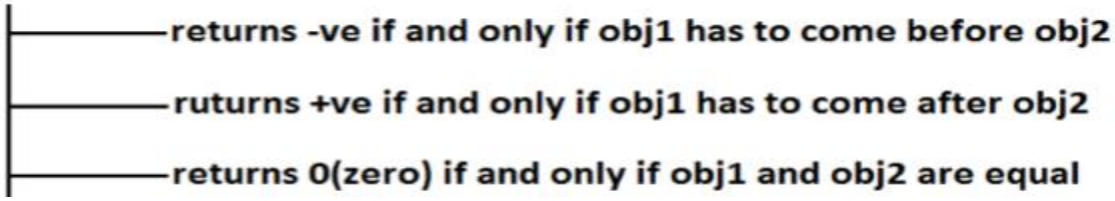
Comparator Interface:

A Comparator interface is used to order the objects of a specific class. This interface is found in java.util package. It contains two methods;

1.compare(Object obj1,Object obj2)
2.equals(Object element).

The first method, compare(Object obj1,Object obj2) compares its two input arguments and showcase the output. It returns a negative integer, zero, or a positive integer to state whether the first argument is less than, equal to, or greater than the second.

The second method, equals(Object element), requires an Object as a parameter and shows if the input object is equal to the comparator. The method will return true, only if the mentioned object is also a Comparator. The order remains the same as that of the Comparator.



Note: 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.

Comparable vs Comparator:

- For predefined Comparable classes default natural sorting order is already available if we are not satisfied with default natural sorting order then we can define our own customized sorting order by Comparator.
- For predefined non Comparable classes [like StringBuffer] default natural sorting order is not available we can define our own sorting order by using Comparator object.
- For our own classes [like Customer, Student, and Employee] we can define default natural sorting order by using 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 order by using Comparator object.

Comparable	Comparator
1) Comparable meant for default natural sorting order.	1) Comparator meant for customized sorting order.
2) Present in java.lang package.	2) Present in java.util package.
3) Contains only one method. compareTo() method.	3) Contains 2 methods. Compare() method. Equals() method.
4) String class and all wrapper Classes implements Comparable interface	4) The only implemented classes of Comparator are Collator and RuleBasedCollator. (used in GUI) Compression

Compression of Set implemented class objects:

Property	HashSet	LinkedHashSet	TreeSet
1) Underlying Data structure.	Hashtable.	LinkedList + Hashtable.	Balanced Tree
2) Insertion order.	Not preserved.	Preserved.	Not preserved (by default).
3) Duplicate objects	Not allowed.	Not allowed	Not allowed.
4) Sorting order	Not applicable.	Not applicable.	applicable.
5) Heterogeneous objects.	Allowed.	Allowed.	Not allowed.
6) Null insertion	Allowed.	Allowed.	Not allowed.

Queue Interface:

Queue interface is part of the Java Collections Framework and represents a data structure that follows the First-In-First-Out (FIFO) principle. A Queue allows elements to be added at the end of the queue and removed from the beginning of the queue.

The Queue interface extends the Collection interface and declares several methods for adding, removing, and examining elements in the queue. Some of the most commonly used methods of the Queue interface include:

1) **add(E e)** - Adds the specified element to the end of the queue. If the queue is full, an exception is thrown.
2) **offer(E e)** - Adds the specified element to the end of the queue. Returns true if the element was added successfully and false if the queue is full.
3) **remove()** - Removes the element at the front of the queue and returns it. If the queue is empty, an exception is thrown.
4) **poll()** - Removes the element at the front of the queue and returns it. Returns null if the queue is empty.
5) **element()** - Returns the element at the front of the queue without removing it. If the queue is empty, an exception is thrown.
6) **peek()** - Returns the element at the front of the queue without removing it. Returns null if the queue is empty.

There are several classes in Java that implement the Queue interface, including LinkedList, PriorityQueue, and ArrayDeque. The choice of implementation depends on the specific needs of your application, such as whether you need a bounded or unbounded queue or if you require priority ordering of elements.

Example:

```
import java.util.Queue;
import java.util.LinkedList;
public class QueueExample {
   public static void main(String[] args) {
      // Create a queue using the LinkedList implementation
      Queue<String> queue = new LinkedList<>();
      // Add elements to the queue
      queue.add("Alice");
      queue.offer("Bob");
      queue.add("Charlie");
      queue.offer("David");
      // Print the elements of the queue
      System.out.println(queue); // Output: [Alice, Bob, Charlie, David]
      // Remove elements from the queue
      queue.remove();
      queue.poll();
      // Print the remaining elements of the queue
      System.out.println(queue); // Output: [Charlie, David]
   }
}
```

Deque interface:

In Java, the Deque (short for "double-ended queue") interface is part of the Java Collections Framework and represents a data structure that allows insertion and removal of elements from both ends. The Deque interface extends the Queue interface and provides additional methods for working with both ends of the queue.

Some of the most commonly used methods of the Deque interface include:

1) **addFirst(E e)** - Inserts the specified element at the beginning of the deque. If the deque is full, an exception is thrown.
2) **addLast(E e)** - Inserts the specified element at the end of the deque. If the deque is full, an exception is thrown.
3) **offerFirst(E e)** - Inserts the specified element at the beginning of the deque. Returns true if the element was added successfully and false if the deque is full.
4) **offerLast(E e)** - Inserts the specified element at the end of the deque. Returns true if the element was added successfully and false if the deque is full.
5) **removeFirst()** - Removes and returns the element at the beginning of the deque. If the deque is empty, an exception is thrown.
6) **removeLast()** - Removes and returns the element at the end of the deque. If the deque is empty, an exception is thrown.
7) **pollFirst()** - Removes and returns the element at the beginning of the deque. Returns null if the deque is empty.
8) **pollLast()** - Removes and returns the element at the end of the deque. Returns null if the deque is empty.
9) **getFirst()** - Returns the element at the beginning of the deque without removing it. If the deque is empty, an exception is thrown.
10) **getLast()** - Returns the element at the end of the deque without removing it. If the deque is empty, an exception is thrown.
11) **peekFirst()** - Returns the element at the beginning of the deque without removing it. Returns null if the deque is empty.
12) **peekLast()** - Returns the element at the end of the deque without removing it. Returns null if the deque is empty**.**

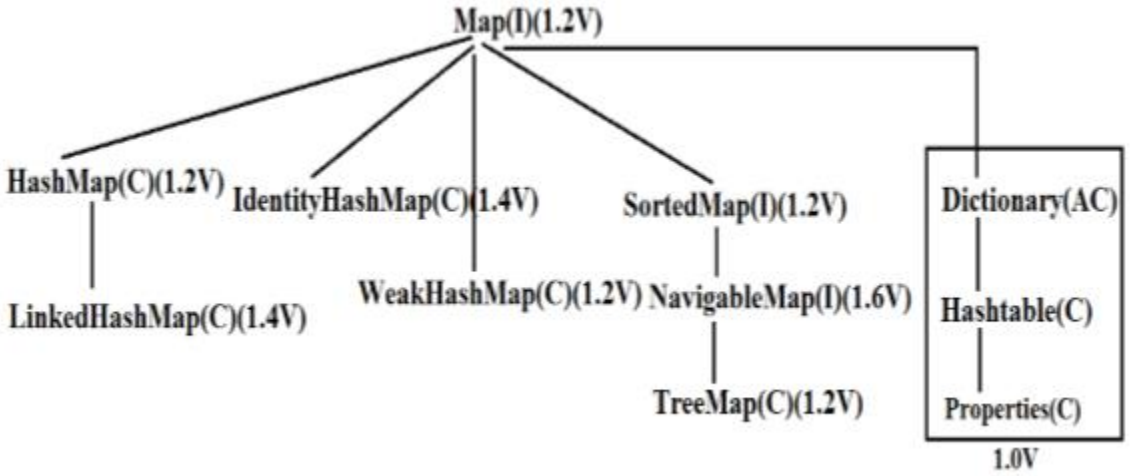
Example

```
import java.util.*;
class DequeDemo{
  public static void main(String[] args){
        Deque<Integer> q=new ArrayDeque<Integer>();
        System.out.println(q); //[]
        //System.out.println(q.removeFirst()); //RE
        System.out.println(q.peek());//null
        System.out.println(q.pollFirst());//null
        q.add(10);
        q.addFirst(90);
        System.out.println(q);//[90,10]
```

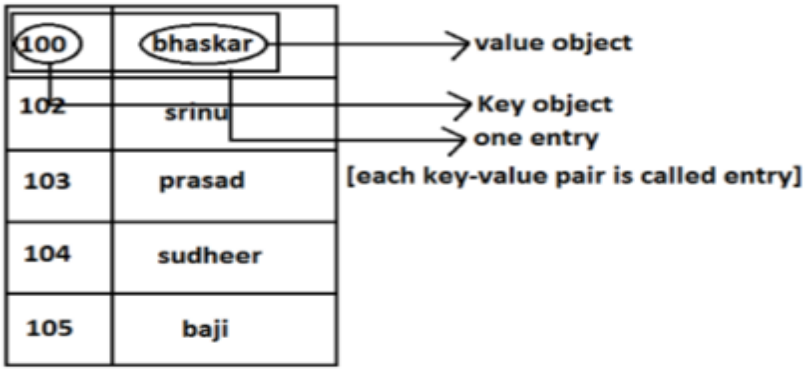
```
        q.addLast(4);//90,10,4
        q.add(11);//90,10,4,11
        q.addFirst(3);//3,90,10,4,11
        System.out.println(q);
        System.out.println(q.offerFirst(20));//true
        System.out.println(q); //20,3,90,10,4,11
        System.out.println(q.peekFirst());//20
        System.out.println(q.peek());//20
        System.out.println(q.peekLast());//11
        System.out.println(q.pollFirst());//20
        System.out.println(q.pollLast());//11
        System.out.println(q);//3,90,10,4,1
    }
}
```

Map Interface

The java collection framework has an interface Map that is available inside the java.util package. The Map interface is not a subtype of Collection interface**.**



1. If we want to represent a group of objects as "key-value" pair then we should go for Map interface.
2. Both key and value are objects only.
3. Duplicate keys are not allowed but values can be duplicated
4. Each key-value pair is called "one entry".



Note: Map interface is not child interface of Collection and hence we can't apply Collection interface methods here.

Map interface defines the following specific methods.

1) **Object put(Object key,Object value):** To add an entry to the Map, if key is already available then the old value replaced with new value and old value will be returned.
2) **void putAll(Map m):** Inserts the specified map into the invoking map.
3) **Object get(Object key):** Returns the value associated with the specified key.
4) **Object remove(Object key):** It removes the entry associated with specified key and returns the corresponding value.
5) **boolean containsKey(Object key):** Returns true if specified key found in the map, else return false.
6) **boolean containsValue(Object value):** Returns true if specified value found in the map, else return false.
7) **boolean isEmpty():**Returns true if the map is empty; otherwise returns false.
8) **Int size():**Returns total number of entries in the invoking Map.
9) **void clear():**Removes all the entries from the map.
10) **Set keySet():** The set of keys we are getting.
11) **Collection values():** The set of values we are getting.
12) **Set entrySet():** The set of entryset we are getting.

Entry interface:

Each key-value pair is called one entry. Hence Map is considered as a group of entry Objects, without existing Map object there is no chance of existing entry object hence interface entry is define inside Map interface(inner interface). Example:

```
interface Map
{       .....;
        .....;
        .....;
        interface Entry  {
                //on Entry we can apply these 3 methods.
                Object getKey();
                Object getValue();
                Object setValue(Object new);
        }
 }
```

HashMap

HashMap is a collection class that implements the Map interface. It is a hash table-based implementation of the Map interface, which means that it uses a hash function to store and retrieve key-value pairs.

A HashMap in Java stores key-value pairs as entries, which are instances of the Map.Entry interface. Each entry contains a key and its corresponding value.

1.The underlying data structure is Hashtable.
2. Duplicate keys are not allowed but values can be duplicated.
3. Insertion order is not preserved and it is based on hash code of the keys.
4. Heterogeneous objects are allowed for both key and value.
5. Null is allowed for keys(only once) and for values(any number of times).
6. It is best suitable for Search operations.

Constructors in HashMap

HashMap in Java has 4 constructors, and each one has public access modifier. The constructors are as follows:

1.HashMap():

Creates an empty HashMap object with default initial capacity 16 and default fill ratio "0.75".

Syntax: HashMap<K, V> hm = new HashMap<K, V>();

2.HashMap(int initialcapacity):

This constructor creates an instance of HashMap with a specified initial capacity and a load factor of 0.75.

Syntax: HashMap<K, V> hm = new HashMap<K, V>(int initialCapacity);

3.HashMap(int initialCapacity, float loadFactor)

This constructor creates an instance of HashMap with a specified initial capacity and specified load factor.

Syntax:

HashMap<K, V> hm = new HashMap<K, V>(int initialCapacity, int loadFactor);

4.HashMap(Map m)

This creates a HashMap instance with the same mappings as the specified map.

Syntax: HashMap<K, V> hm = new HashMap<K, V>(Map map);

Example:

```
import java.util.*;
public class HashMapExample {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put("1", "1"); // put example
        map.put("2", "2");
        map.put("3", "3");
        map.put("4", null); // null value
        map.put(null, "100"); // null key
        String value = map.get("3"); // get example
        System.out.println(map.containsKey(null));
        System.out.println(map.containsValue("100"));
```

```
            System.out.println(map.entrySet());
            System.out.println("map size=" + map.size());
            map.clear();
            System.out.println("map is empty=" + map.isEmpty());
        }
}
```

LinkedHashMap

LinkedHashMap is another implementation of the Map interface that provides a hash table-based implementation with predictable iteration order. It extends the HashMap class and maintains a linked list of entries in the order in which they were added to the map.

Like HashMap, LinkedHashMap stores key-value pairs as entries. Each entry contains a key and its corresponding value.

1.The underlying data structure is LinkedList and Hashtable.
2. Duplicate keys are not allowed but values can be duplicated.
3. Insertion order is preserved and it is based on hash code of the keys.
4. Heterogeneous objects are allowed for both key and value.
5. Null is allowed for keys(only once) and for values(any number of times).
6. It is best suitable for Search operations.

Constructors in LinkedHashMap

HashMap in Java has 4 constructors, and each one has public access modifier. The constructors are as follows:

1. LinkedHashMap():

Creates an empty LinkedHashMap object with default initial capacity 16 and default fill ratio "0.75".

Syntax: LinkedHashMap<K, V> hm = new LinkedHashMap<K, V>();

2. LinkedHashMap(int initialcapacity):

This constructor creates an instance of LinkedHashMap with a specified initial capacity and a load factor of 0.75.

Syntax:

LinkedHashMap<K, V> hm = new LinkedHashMap<K, V>(int initialCapacity);

3. LinkedHashMap(int initialCapacity, float loadFactor)

This constructor creates an instance of LinkedHashMap with a specified initial capacity and specified load factor.

Syntax:

LinkedHashMap<K, V> hm = new LinkedHashMap<K, V>(int initialCapacity, int loadFactor);

4. LinkedHashMap(Map m)

This creates a LinkedHashMap instance with the same mappings as the specified map.

Syntax: LinkedHashMap<K, V> hm = new LinkedHashMap<K, V>(Map m);

Example:

```
import java.util.*;
public class LinkedHashMapExample {
  public static void main(String[] args) {
        Map<String, String> map = new LinkedHashMap<>();
        map.put("1", "1"); // put example
        map.put("2", "2");
        map.put("3", "3");
        map.put("4", null); // null value
        map.put(null, "100"); // null key
        String value = map.get("3"); // get example
        System.out.println(map.containsKey(null));
        System.out.println(map.containsValue("100"));
        System.out.println(map.entrySet());
        System.out.println("map size=" + map.size());
        map.clear();
        System.out.println("map is empty=" + map.isEmpty());
    }
}
```

SortedMap

SortedMap in Java is an interface that is a subinterface of Map interface. The entries in the map are maintained in sorted ascending order based on their keys.

In simple words, it guarantees that the entries are maintained in ascending key order. It allows very efficient manipulations of subsets of a map.
Sorting is possible only based on the keys but not based on values.

SortedMap interface defines the following 6 specific methods.
1. Object firsyKey(): It returns the first (lowest or smallest) key in the invoking map.
2. Object lastKey(): It returns the last (highest or largest) key in the invoking map
3. SortedMap headMap(Object key): This method returns a portion of the map whose keys are strictly less than toKey.
4. SortedMap tailMap(Object key): This method returns a portion of the map whose keys are greater than or equal to fromKey.
5. SortedMap subMap(Object key1,Object key2): Returns a map containing those entries with keys that are greater than or equal to key1 and less than key2.
6. Comparator comparator(): Returns the invoking sorted map's comparator. If the natural ordering is used for the invoking map, null is returned.

NavigableMap Interface:

NavigableMap interface is a subinterface of the SortedMap interface that provides additional navigation methods for accessing and manipulating the entries in the map. It defines methods for finding entries based on their keys and for finding the closest matches for a given key.

A NavigableMap is a sorted map, which means that its entries are sorted by their keys in natural order or using a custom Comparator. In addition to the methods inherited from SortedMap, the NavigableMap interface provides the following navigation methods:

1) **lowerKey(Object key):** Returns the greatest key strictly less than the given key, or null if there is no such key.
2) **lowerEntry(Object key):** Returns a Map.Entry with the greatest key strictly less than the given key, or null if there is no such entry.
3) **floorKey(Object key):** Returns the greatest key less than or equal to the given key, or null if there is no such key.
4) **floorEntry(Object key):** Returns a Map.Entry with the greatest key less than or equal to the given key, or null if there is no such entry.
5) **ceilingKey(Object key):** Returns the least key greater than or equal to the given key, or null if there is no such key.
6) **ceilingEntry(Object key):** Returns a Map.Entry with the least key greater than or equal to the given key, or null if there is no such entry.
7) **higherKey(Object key):** Returns the least key strictly greater than the given key, or null if there is no such key.
8) **higherEntry(Object key):** Returns a Map.Entry with the least key strictly greater than the given key, or null if there is no such entry.
9) **pollFirstEntry():** Removes and returns the first entry in the map, or returns null if the map is empty.
10) **pollLastEntry():** Removes and returns the last entry in the map, or returns null if the map is empty.

TreeMap

TreeMap is a class that implements the NavigableMap interface and stores its elements in a sorted order based on their natural ordering or a custom Comparator. Each element in a TreeMap must have a unique key and the keys cannot be null.

1. The underlying data structure is RED-BLACK Tree.
2. Duplicate keys are not allowed but values can be duplicated.
3. Insertion order is not preserved and all entries will be inserted according to some sorting order of keys.
4. If we are depending on default natural sorting order keys should be homogeneous and Comparable otherwise we will get ClassCastException.
5. If we are defining our own sorting order by Comparator then keys can be heterogeneous and non Comparable.
6. There are no restrictions on values they can be heterogeneous and non Comparable.
7. null keys are not allowed but null values are allowed.

Constructors:

1.TreeMap t=new TreeMap(): For default natural sorting order.

This constructor constructs an empty tree map that will be sorted using the natural order of its keys.

2.TreeMap t=new TreeMap(Comparator c): For customized sorting order.

This constructor constructs an empty tree-based map that will be sorted using the Comparator comp

3.TreeMap t=new TreeMap(SortedMap m):

This constructor initializes a tree map with the entries from the SortedMap m, which will be sorted in the same order as m

4.TreeMap t=new TreeMap(Map m):

This constructor initializes a tree map with the entries from m, which will be sorted using the natural order of the keys.

Example for TreeMap:

```
import java.util.TreeMap;
public class TreeMapExample {
   public static void main(String[] args) {
      TreeMap<String, Integer> myMap = new TreeMap<>();
      // Add some key-value pairs to the map
      myMap.put("Alice", 25);
      myMap.put("Bob", 32);
      myMap.put("Charlie", 18);
      myMap.put("David", 45);
      myMap.put("Emily", 29);
      // Print the map
      System.out.println(myMap);
       // Get the value associated with a key
      System.out.println("Value for key Bob: " + myMap.get("Bob"));
      // Remove a key-value pair
      myMap.remove("Charlie");
       // Print the updated map
      System.out.println(myMap);
   }
}
```

HashMap	LinkedHashMap
1) The underlying data structure is Hashtable.	1) The underlying data structure is a combination of Hashtable+ LinkedList.
2) Insertion order is not preserved	2) Insertion order is preserved.
3) introduced in 1.2.v	3) Introduced in 1.4v.