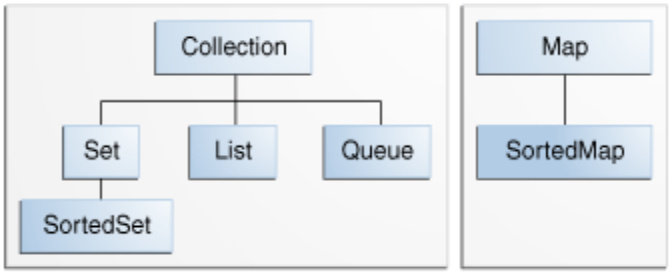
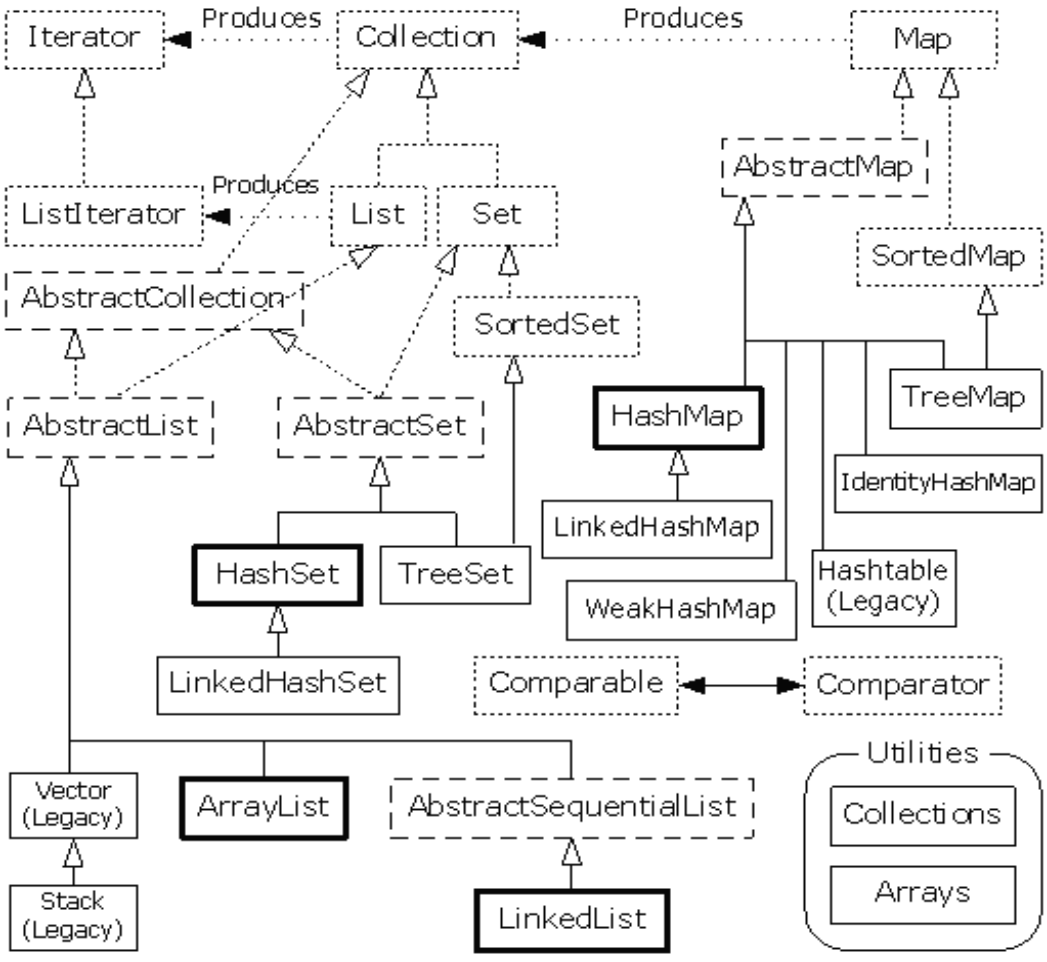
Collections

1. Core collections interfaces are



Here Set is special kind of collections and SortedSet is special kind of Set..it goes like that.

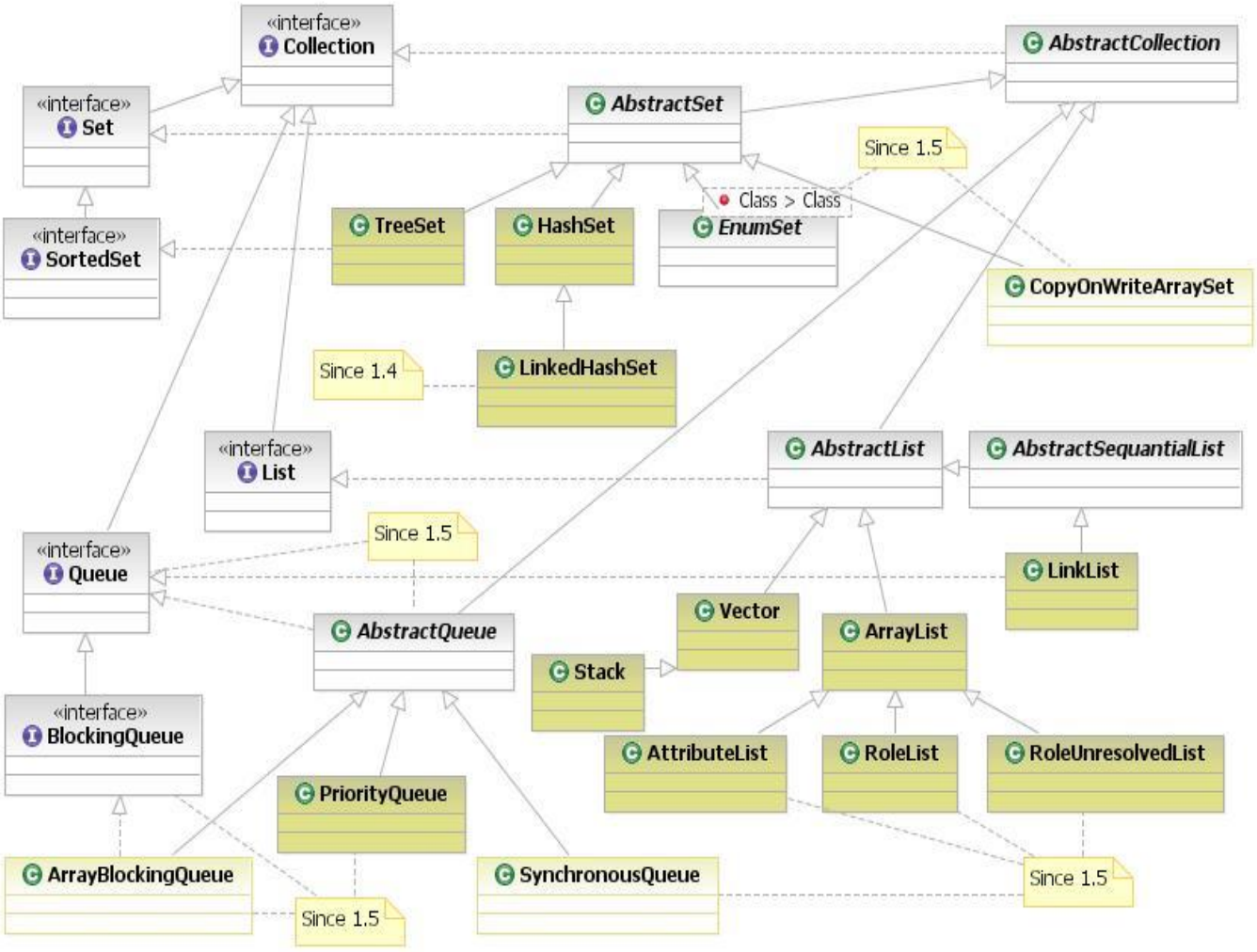
2. Elaborated Collection framework(without queue: Thinking In java 4th edition)



3. Also there are two distinct tree as show in diagram above. The Map is not a true collection.
4. All collections are generic. For example, declaration for collection interface is

```
public interface Collection<E>...
```

- One more collection diagram(including queue).



<---------- Implements

<---------- Extends

5. While declaring a collection instance, one should specify the type of object contained in it. Specifying the type allows compiler to verify the correct type of the object thus reducing errors during runtime.
6. Collection: Collection is the root of collection hierarchy. This is the interface which all collections implement. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific sub-interfaces, such as `Set` and `List`.

   Set: Is the collection which cannot contain duplicate elements. This interface models the mathematical set and is used to represent sets, such as cards comprising a poker hand.

   List: An ordered collection (Sometimes called a sequence). List can contain duplicate elements. A user of the List has precise control over it that where is each element has been inserted and can access them using their integer index (position).

   Queue: A collection which is used to hold multiple elements prior to processing. Besides basic collection operations, a queue provides additional insertion, extraction and inspection operations. Queues typically, but not necessarily, order elements in FIFO (first in, first out) manner. Every queue implementation must specify its ordering properties.

   Map: An object that maps keys to values. A map cannot contain duplicate keys and each key at most map to one value.

   SortedSet: A set that maintains its elements in ascending order. Several other operations have been provided to take the advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

   SortedMap: A map that maintains its ordering in ascending key order. They are used for natural ordered collections of key/value pairs, such as dictionaries and telephone directories.
7. Also we can see in above diagram an abstract class AbstractCollection extends Collection interface. AbstractCollection provides skeleton implementation of Collection interface just to minimize the effort needed to implement it.
8. Then other abstract classes like AbstractSet, AbstractList etc extends AbstractCollection and implements interfaces like Set, List etc. Then from this AbstractSet, AbstractList etc we will going to create our HashSet and Arraylist concrete classes.
9. Optional Operations:

   - To keep the core interfaces manageable , the java platform doesn't provide separate interface for each variant of each collection type(such variants might include immutable, fixed size and append only) . Instead, the modification operation in each interface is designated **optional**.
     Example : For collection interface method add(E e) is marked as optional as shown below.

Modifier and Type	Method and Description
boolean	**add**(**E** e) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(**Collection**<? extends **E**> c) Adds all of the elements in the specified collection to this collection (optional operation).

- A given implementation may elect not to support all operations. If unsupported operation is invoked, a collection throws `UnsupportedOperationException`. These optional operations will be already defined as unsupported by abstract classes like AbstractCollection, AbstractList etc as shown below.

  Example 1: add(E paramE) method of Collection interface is defined in AbstractCollection class as

```
public boolean add(E paramE)
  {
      throw new UnsupportedOperationException();
  }
```

  Example 2 : Method add(int paramInt, E paramE) of the interface List is defined in AbstractList class as

```
public void add(int paramInt, E paramE)
  {
      throw new UnsupportedOperationException();
   }
```

- i.e. Now its completely left to implementation to define(override) these operations in their implementation if they want to support them. All java platform's general purpose implementation (As shown in table below under implementations) supports all optional (i.e. define) operations. Example class ArrayList defines (overrides) both `add(E paramE)` and `add(int paramInt, E paramE)` methods .

10. Collection Interface:

- Any class which defines a collection should implement this interface.
- Collection is a generic interface with declaration
- interface Collection<E>

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[ ] toArray(T *array*[ ])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

- The collection methods can throw following exceptions

i. **UnsupportedOperationException**: When collection cannot be modified.

ii. **ClassCastException**: When attempt is made to add incompatible object to the collection.

iii. **NullPointerException**: When attempt is made to store a null object.

iv. **IllegalArgumentException**: If an invalid argument is used.

v. **IllegalStateException**: If attempt is made to add an element to a fixed length collection which is full.

vi. **ArrayStoreException** : If attempt is made to return an array contains collection elements and type mismatch occurs.

- Traversing the collection

There are two ways to traverse through the collection

i. With for-each

```
for (Object o : collection)
System.out.println(o);
```

ii. Using Iterator

One can get the iterator of a collection by calling iterator method. Following is the Iterator interface.

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

hasNext() – returns true If iterator has more elements

next() – returns next element in the iteration.

remove() – removes the last element that was returned by next

11. Iterator and remove() method :

- The method remove() of interface Iterator removes the element from the collection. But one should call next() Method before calling remove second time else `IllegalStateException` will be thrown. Example for the same is shown below.

```
package collections;

import java.util.*;

public class IteratorRemoveMethod
{
      public static void main(String[] args)
      {
            ArrayList<String> al = new ArrayList<String>();
            al.add("one");
            al.add("two");
            al.add("three");
            System.out.println("al after addn "+al);

            Iterator<String> it = al.iterator();
            while(it.hasNext())
            {
```

```
                    String str = it.next();
                    it.remove();
                    it.remove();
                }
            }
        }
```

Output:

```
al after addn [one, two, three]
Thread [main] (Suspended (exception IllegalStateException))
                ArrayList$Itr.remove() line: 804
                IteratorRemoveMethod.main(String[]) line: 20
```

12. Set Interface:

- It contains methods inherited from *Collection* and adds the restriction that duplicate elements are Prohibited, therefore add() method returns false if attempt is made to add duplicate element to set.
- Set doesn't define any additional method of its own.
- Syntax : `public interface Set<E> extends Collection<E>`

13. SortedSet Interface :

- SortedSet interface extends Set and declares the behaviour of a set sorted in ascending order.
- Syntax : `public interface SortedSet<E> extends Set<E>`
- In addition to method of Set, SortedSet declares some extra methods as listed below.

Method	Description
Comparator<? super E> comparator( )	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, **null** is returned.
E first( )	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E *end*)	Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
E last( )	Returns the last element in the invoking sorted set.
SortedSet<E> subSet(E *start*, E *end*)	Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object.
SortedSet<E> tailSet(E *start*)	Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

14. NavigableSet:

- NavigableSet interface extends SortedSet and declares the behaviour of a collection that supports the retrieval of elements based on the closest match to a given value or values.
- Syntax : `public interface NavigableSet<E> extends SortedSet<E>`

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
E higher(E *obj*)	Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E pollFirst( )	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. **null** is returned if the set is empty.
E pollLast( )	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. **null** is returned if the set is empty.
NavigableSet<E> subSet(E *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E *lowerBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting set is backed by the invoking set.

Note: In above table method higher(E obj) returns the least element in this set strictly greater than the given element, or null if there is no such element.

15. List Interface:

- List interface extends collection and declares the behaviour of a collection that stores a sequence of elements(ordered collection also known as a ***sequence***).

- Apart from methods inherited through Collection list have following additional methods
- List also provides its own richer iterator called ' `ListIterator`'. ListIterator interface has following methods

Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

- Syntax : public interface **List<E>** extends Collection<E>

Method	Description
void add(int *index*, E *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int *index*, Collection<? extends E> *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.

int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified *index*.
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

16. Queue Interface:

- A collection which is used to hold multiple elements prior to processing. Besides basic collection operations, a queue provides additional insertion, extraction and inspection operations. Queues typically, but not necessarily, order elements in FIFO (first in, first out) manner. Every queue implementation must specify its ordering properties.
- Syntax : `public interface Queue<E> extends Collection<E>`
-

Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
boolean offer(E *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
E peek( )	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
E poll( )	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
E remove( )	Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty.

17. Deque Interface:

- The **Deque** interface extends **Queue** and declares the behavior of a double-ended queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks.
- public interface **Deque<E>** extends Queue<E>

Method	Description
void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Adds *obj* to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst( )	Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
E getLast( )	Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
boolean offerFirst(E *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise.
E peekFirst( )	Returns the element at the head of the deque. It returns **null** if the deque is empty. The object is not removed.
E peekLast( )	Returns the element at the tail of the deque. It returns **null** if the deque is empty. The object is not removed.
E pollFirst( )	Returns the element at the head of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pollLast( )	Returns the element at the tail of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pop( )	Returns the element at the head of the deque, removing it in the process. It throws **NoSuchElementException** if the deque is empty.
void push(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeFirstOccurrence(Object *obj*)	Removes the first occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.
E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeLastOccurrence(Object *obj*)	Removes the last occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.

18. Implementations:

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

19. Collection Classes:

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface.
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

20. ArrayList :

- Syntax :

```
public class ArrayList<E> extends AbstractList<E> implements List<E>,
RandomAccess, Cloneable, Serializable
```

- ArrayList class extends AbstractList and implement the list interface.
- ArrayList supports dynamic arrays that can grow as needed. In java we know that arrays are of fix size that means one must know in advance that how many elements an array will hold. But sometimes, you may not know till run time precisely large an array one need. To handle this situation, collections framework defines ArrayList.
- In essence ArrayList is a variable length array of object references.

- Dynamic array is also hold by legacy class Vector.
- ArrayLists are created with initial size, when the size is exceeded the collector will automatically enlarge.
- **ArrayList** has the constructors shown here:

`ArrayList()` Constructs an empty list with an initial capacity of ten.
`ArrayList(Collection<? extends E> c)` Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
`ArrayList(int initialCapacity)` Constructs an empty list with the specified initial capacity.

-Example of ArrayList :

```
package collections;

import java.util.*;

public class ArrayListDemo
{
public static void main(String[] args)
   {
        ArrayList<String> al = new ArrayList<String>();
System.out.println("initial size of the arraylist is "+ al.size());

        //adding elements to arraylist al
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        al.add(1, "A2");

        //size and contents after addition
        System.out.println("Size of arraylist after addition is "+
  al.size());
        System.out.println("content of al "+ al);
        System.out.println();

        //removing contents of al
        al.remove("F");
        al.remove(2);

        //size and contents after deletion
System.out.println("Size of arraylist after deletion is "+ al.size());
        System.out.println("content of al "+ al);
        System.out.println();
```

```
        //creating duplicate arraylist al1
        ArrayList<String> al1 = new ArrayList<String>();
        al1.addAll(al);
        System.out.println("content of al1 "+al1);
        System.out.println();

        //method contains()
        if(al.contains("C"))
        {
              System.out.println("contains");
        }
        else
        {
              System.out.println("dont contains");
        }
        System.out.println();

        //method equals()
        if(al.equals(al1))
        {
              System.out.println("al and al1 are equal");
        }
        else
        {
              System.out.println("al and al1 are unequal");
        }
        System.out.println();

        //hashcode for al
        System.out.println("hash code for al is '"+al.hashCode()+"'");
        System.out.println();

        //Using an iterator
        Iterator<String> it = al.iterator();
        System.out.println("contents of al through iterator:");
        while(it.hasNext())
        {
              String element = it.next();
              System.out.println(element+" ");
        }
        System.out.println();

        //equal works even after removing all elements
        al.removeAll(al);
        al1.removeAll(al1);
        System.out.println("content of al "+ al);
        System.out.println("content of al1 "+ al1);
        if(al.equals(al1))
        {
              System.out.println("al and al1 are equal");
        }
        else
        {
              System.out.println("al and al1 are unequal");
        }
        System.out.println();

        System.out.println(al.isEmpty());

    }
}
```

```
Output:
initial size of the arraylist is 0
Size of arraylist after addition is 7
content of al [C, A2, A, E, B, D, F]

Size of arraylist after deletion is 5
content of al [C, A2, E, B, D]

content of al1 [C, A2, E, B, D]

contains

al and al1 are equal

hash code for al is '152091896'

contents of al through iterator:
C
A2
E
B
D

content of al []
content of al1 []
al and al1 are equal

true
```

- Although the size of ArrayList increases automatically when more elements are added but one can increase the size manually using **ensureCapacity()** method. Signature of the method is void ensureCapacity(int *cap*). This will ensure minimum capacity not maximum limit i.e. if capacity is surpassed then it get increased automatically.
- Conversely if one wanted to reduce the capacity then we have **trimToSize()**. ArrayList backed by an array whose current capacity may be over hence capacity get increased automatically. Example let's say the initial capacity was 10 and if 11 elements are added then during addition of 11^{th} element it will automatically increases capacity to say 15. But as we know only 11 elements have been added the remaining memory of 4 elements just stay there until more elements got added. When one is sure of no more elements needs to be added then he can call **trimToSize()** method on that ArrayList which ensures that its capacity reduced to its current size i.e. 11 in this case.
- Obtaining an array from ArrayList:
    - Because of following (and other) reasons collection needs to be converted in arrays.
        1. To obtain faster processing time for certain operations.
        2. To pass an array to method which is not overloaded to accept a collection.
        3. To integrated collection based code with legacy code that doesn't understand collection.

(Note: If your List is fixed in size — that is, you'll never use remove, add, or any of the bulk operations other than containsAll — then the best option is to use Arrays.asList because arrays are not resizeable).

- There are two versions of toArray()

  object[ ] toArray( )
  <T> T[ ] toArray(T *array*[ ])

  The first returns array of Object and second one returns array of elements that has the same type as **T** which is more convenient.

- Example:

```
package collections;
import java.util.*;

public class ArrayListToArray
{
    public static void main(String[] args)
    {
          ArrayList<Integer> al = new ArrayList<Integer>();

          al.add(1);
          al.add(2);
          al.add(3);
          al.add(4);

          System.out.println("Contents of al : "+al);

          Integer ai[] = new Integer[al.size()];
          ai = al.toArray(ai);

          int sum = 0;

          for(int i : ai)
          sum += i;

          System.out.println("Sum is : "+sum);
    }
}
Output:
Contents of al : [1, 2, 3, 4]
Sum is : 10
```

- Arrays class has a static factory method called 'asList' which allows an array to be viewed as a List.
- <u>Difference between ArrayList and Vector</u>
  - All methods of Vector are synchronized but the methods of array list are not synchronized. Because of this Vector becomes slow and ArrayList is fast.
  - Vector and ArrayList both uses arrays internally as data structure. Therefore they are dynamically resizable, difference is in the way they internally resized. By default Vector doubles the size of its array when size is increased but ArrayList increases by half.

21. LinkedList Class:

- LinkedList Class extends **AbstractSequentialList** and implements **List**, **Queue** and **Deque** interfaces.
- `Syntax : public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable`
- A **LinkedList** is relatively slow for random access, but it has larger feature set than the **ArrayList**

- Constructor

LinkedList( )
LinkedList(Collection<? extends E> *c*)

The first constructor builds an empty linked list. The second constructor builds a linked list that is initialized with the elements of the collection *c*.

- Example 2:

```
package collections;

import java.util.LinkedList;

public class LinkedListDemo
{
   public static void main(String[] args)
   {
       LinkedList<String> ll = new LinkedList<String>();

       ll.add("F");
       ll.add("B");
       ll.add("D");
       ll.add("E");
       ll.add("C");
       ll.addLast("Z");
       ll.addFirst("A");
       ll.addFirst("A");

       ll.add(2, "A2");

       System.out.println("Original content of ll:"+ll);

       //search operation
       System.out.println("index of A2:"+ll.indexOf("A2"));
       System.out.println("index of A:"+ll.indexOf("A"));
       System.out.println("index of A:"+ll.lastIndexOf("A"));

       System.out.println("first element is "+ll.getFirst());
       System.out.println("last element is "+ll.getLast());

       //rang view operation
       System.out.println("sublist :"+ll.subList(0, 2));

       //LinkedList implements cloneable interface
```

```
        System.out.println("clone linkedlist is "+ll.clone());

        ll.remove("F");
        ll.remove(2);

        System.out.println("Content of ll after deletion:"+ll);

        ll.removeFirst();
        ll.removeLast();

        System.out.println("ll after removing first and last:"+ll);

        String val = ll.get(2);
        ll.set(2, val+" changed");

        System.out.println("Content of ll :"+ll);
    }
}
Output:
Original content of ll:[A, A, A2, F, B, D, E, C, Z]
index of A2:2
index of A:0
index of A:1
first element is A
last element is Z
sublist :[A, A]
clone linkedlist is [A, A, A2, F, B, D, E, C, Z]
Content of ll after deletion:[A, A, B, D, E, C, Z]
ll after removing first and last:[A, B, D, E, C]
Content of ll :[A, B, D changed, E, C]
```

- Above we can see, we have ll.clone() method also because LinkedList implements `Cloneable` interface which actually has no clone() method but interface is an indication for object.clone() method. Clone method here Returns a shallow copy of this LinkedList. (The elements themselves are not cloned).

22. Set Implementation:
    - Java has three general purpose Set implementations, they are HashSet, TreeSet and LinkedHashset.
    - HashSet which stores elements in hash table, is best performing implementation but it don't give any guarantee for order of iteration.
    - TreeSet which stores its elements in red-black tree, orders its elements based on their values. It is slower than HashSet.
    - LinkedHashSet which implements hash table with linked list running through it, orders its elements based on order they have been inserted into it(insertion order).

    Example of different flavours of sets :

```
package collections;

import java.util.*;

public class DifferentFlavorsOfSets {
  public static void main(String [] args)
  {
```

```
        ArrayList<String> Al = new ArrayList<String>();
        Al.add("D");
        Al.add("X");
        Al.add("B");
        Al.add("C");
        Al.add("E");

        System.out.println("contents of Array List are :"+Al);

        HashSet<String> Hs = new HashSet<String>(Al);

        System.out.println("contents of Hash set are :"+Hs);

        LinkedHashSet<String> Lhs = new LinkedHashSet<String>(Al);
        System.out.println("contents of Linked Hash set are :"+Lhs);

        //if an element is re-inserted then insertion order is not
affected
        Lhs.add("Y");
        System.out.println("contents of Linked Hash set are :"+Lhs);

        TreeSet<String> ts = new TreeSet<String>(Lhs);
        System.out.println("contents of tree set are "+ts);
  }
  }
  Output:
  contents of Array List are :[D, X, B, C, E]
  contents of Hash set are :[D, E, B, C, X]
  contents of Linked Hash set are :[D, X, B, C, E]
  contents of Linked Hash set are :[D, X, B, C, E, Y]
  contents of tree set are [B, C, D, E, X, Y]
```

Above we can see that how ArrayList allows duplicate while Hash set don't accept duplicates. Next it can be seen that Linked Hash Set maintains insertion order(not natural ordering) and put D in the last place. And finally tree set copies same elements of linked hash set but maintains natural ordering of its elements.

23. HashSet Class:

- Syntax : `public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, java.io.Serializable`
- HashMap is backed by HashTable (Actually a HashMap instance, if one will open the HashSet source code file then find the use of an HashMap instance and HashMap has same code as HashTable only difference is that HashMap methods are not synchronized and permits null values). Similarly HashSet methods calls corresponding HashMap methods using HashMap instance. Example

```
public int size()
  {
      return map.size();
  }
```

- HashSet doesn't define any additional methods beyond those provided by its super classes and interfaces.

- Constructors:

Constructors	
Constructor and Description	
HashSet()	Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).
HashSet(**Collection**<? extends **E**> c)	Constructs a new set containing the elements in the specified collection.
HashSet(int initialCapacity)	Constructs a new, empty set; the backing HashMap instance has the specified initial capacity (default capacity is 16 and load factor 0.75).
HashSet(int initialCapacity, float loadFactor)	Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and the specified load factor. The load factor should be between 0.0 to 1.0.

- Example of HashSet

```
package collections;

import java.util.*;

public class HashSetDemo
{
	public static void main(String[] args)
	{
		HashSet<Integer> hs = new HashSet<Integer>();
		hs.add(20);
		hs.add(10);
		hs.add(15);
		hs.add(15);
		hs.add(5);

		System.out.println("hs after additions "+hs);

		hs.remove(5);
		System.out.println("hs after deletion "+hs);

	System.out.println("hs contains object? "+hs.contains(15));
	System.out.println("hs is empty? "+hs.isEmpty());

	HashSet<Integer> hs1 = new HashSet<Integer>();
	hs1.add(2);
	hs1.add(4);
```

```
            hs1.add(10);

            HashSet<Integer> hs2 = new HashSet<Integer>();
            hs2.add(15);

            System.out.println("hs1 after addtn "+hs1);
            System.out.println("hs2 after addtn "+hs2);

            //equal - should be exactly same
            System.out.println("hs equal hs1 ?"+hs.equals(hs1));
            System.out.println("hs equal hs2 ?"+hs.equals(hs2));

            //containsAll
            System.out.println("hs contains all of hs1?
"+hs.containsAll(hs1));
            System.out.println("hs contains all of hs2?
"+hs.containsAll(hs2));

            //removeAll - remove which is common and retain others
            hs.removeAll(hs1);
            System.out.println("hs after removeAll "+hs);

            //retainAll - retain which is common and remove others
            hs.add(10);
            hs.retainAll(hs1);
              System.out.println("hs after retainAll "+hs);
      }
}
Output:
```

```
hs after additions [5, 20, 10, 15]
hs after deletion [20, 10, 15]
hs contains object? true
hs is empty? false
hs1 after addtn [2, 4, 10]
hs2 after addtn [15]
hs equal hs1 ?false
hs equal hs2 ?false
hs contains all of hs1? false
hs contains all of hs2? true
hs after removeAll [20, 15]
hs after retainAll [10]
```

Above programme we can see the function equals only work for exactly equal hash sets. Function removeAll removes the common elements from the invoking set and retain other elements and the opposite is true for retainAll which retail all the common elements in invoking set and removes the others.

24. LinkedHashSet:

- `public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable`
- It is hash table and linked list implementation of set interface.
- It extends HashSet and implements set. One can say LinkedHashSet is nothing but Hash set with linked list running through it as name suggest. i.e. **LinkedHashSet** maintains a linked list of

the entries in the set, in the order in which they were inserted. In another word LinkedHashSet is nothing but HashSet which maintains insertion order of the elements.

- Also it defines four constructors as shown below.

Constructors	
Constructor and Description	
`LinkedHashSet()`	Constructs a new, empty linked hash set with the default initial capacity (16) and load factor (0.75).
`LinkedHashSet(Collection<? extends E> c)`	Constructs a new linked hash set with the same elements as the specified collection.
`LinkedHashSet(int initialCapacity)`	Constructs a new, empty linked hash set with the specified initial capacity and the default load factor (0.75).
`LinkedHashSet(int initialCapacity, float loadFactor)`	Constructs a new, empty linked hash set with the specified initial capacity and load factor

- How LinkedHashSet implements LinkedList ? i.e. how it take care of order of insertion ? Because its syntax shows its extends HashSet and implements Set both of which don't provide such behaviour. Here is how it works.

  If we check the source for LinkedHashSet file, we find that it contains only implementation of above four constrcutors. But each implementation of the constructor is calling a super constructor.

  Example :

```
public LinkedHashSet(int paramInt, float paramFloat)
   {
         super(paramInt, paramFloat, true);
    }
```

  Which is the fourth constructor listed below. Here we can see that super constructor taking a Boolean value 'true' as one of the parameter. But SuperClass HashSet don't have such constructor. Actually HashSet.java file defines one constructor only for the purpose of LinkedHashSet which is shown below.

```
HashSet(int paramInt, float paramFloat, boolean paramBoolean)
   {
         this.map = new LinkedHashMap(paramInt, paramFloat);
   }
```

From above its clear that LinkedHashSet like HashSet internally used instance of LinkedHashMap as a has-a-relationship. That is the key.

25. TreeSet :

- Syntax : `public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable`
- Elements of TreeSet are arranged in their natural order(due to comparable interface as explained below) or according to the comparator provided during set creation.
- Constructors

Constructor and Description
TreeSet() Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
TreeSet(**Collection**<? extends **E**> c) Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements.
TreeSet(**Comparator**<? super **E**> comparator) Constructs a new, empty tree set, sorted according to the specified comparator.
TreeSet(**SortedSet**<**E**> s) Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

- TreeSet is not synchronized and its better to synchronize it during creation if more than one thread going to access it. The better way to synchronize is to use `Collections.synchronizedSortedSet`
- Example :

```
package collections;

import java.util.*;

public class TreeSetClass
{
```

```
public static void main(String[] args)
{
	TreeSet<Integer> tsc = new TreeSet<Integer>();

	//adding element to first treeset tsc
	tsc.add(5);
	tsc.add(7);
	tsc.add(3);
	tsc.add(2);
	tsc.add(6);
	tsc.add(3);

	System.out.println("Elements of tree set "+tsc);

	//methods related to SortedSet
	System.out.println("--------SortedSet Methods-------");

	//first and last
	System.out.println("first element in tsc is "+tsc.first());
	System.out.println("last element in tsc is "+tsc.last());

	//headSet, tailSet and subSet
	System.out.println("head set for 6 "+tsc.headSet(6));
	System.out.println("tail set for 5 "+tsc.tailSet(5));
	System.out.println("subset between 2 and 6 "+tsc.subSet(2,6));


	//methods related to NavigableSet
	System.out.println("--------NavigableSet Methods-------");

	//descendingIterator and decendingSet
	System.out.println("printing set using descending iterator");
	Iterator<Integer> it = tsc.descendingIterator();
	while(it.hasNext())
	{
		Integer element = it.next();
		System.out.println(element);
	}
	System.out.println("Descending Set is "+tsc.descendingSet());

	//floor and ceiling
	System.out.println("ceiling value of 4 "+tsc.ceiling(4));
	System.out.println("ceiling value of 8 "+tsc.ceiling(8));
	System.out.println("floor value of 2 "+tsc.floor(2));
	System.out.println("floor value of 1 "+tsc.floor(1));

	//headSet, tailSet and subset
	System.out.println("head set for 5 "+tsc.headSet(5, true));
	System.out.println("head set for 5 "+tsc.headSet(5, false));
	System.out.println("tail set for 3 "+tsc.tailSet(3, true));
	System.out.println("tail set for 3 "+tsc.tailSet(3, false));
	    System.out.println("subset for 3 and 7 is "+tsc.subSet(3,
	    true, 7, false));

	//higher and lower
	System.out.println("higher element than 6 is "+tsc.higher(6));
	System.out.println("higher element than 7 is "+tsc.higher(7));
	System.out.println("lower element than 4 is "+tsc.lower(4));
	System.out.println("lower element than 2 is "+tsc.lower(2));

	//pollFirst and pollLast
```

```
            tsc.pollFirst();
            System.out.println("after pollFirst "+tsc);
            tsc.pollLast();
            System.out.println("after pollLast "+tsc);
        }
    }
```

Output:

```
Elements of tree set [2, 3, 5, 6, 7]
--------SortedSet Methods-------
first element in tsc is 2
last element in tsc is 7
head set for 6 [2, 3, 5]
tail set for 5 [5, 6, 7]
subset between 2 and 6 [2, 3, 5]
--------NavigableSet Methods-------
printing set using descending iterator
7
6
5
3
2
Descending Set is [7, 6, 5, 3, 2]
ceiling value of 4 5
ceiling value of 8 null
floor value of 2 2
floor value of 1 null
head set for 5 [2, 3, 5]
head set for 5 [2, 3]
tail set for 3 [3, 5, 6, 7]
tail set for 3 [5, 6, 7]
subset for 3 and 7 is [3, 5, 6]
higher element than 6 is 7
higher element than 7 is null
lower element than 4 is 3
lower element than 2 is null
after pollFirst [3, 5, 6, 7]
after pollLast [3, 5, 6]
```

Comparable<T> and natural ordering :

- If there is a list 'l' then it can be sorted as below
  Collections.sort(l);
- If list consists of String elements then it will be sorted in alphabetical order, if its Date elements then it will be sorted in Chronological order etc which are called as natural ordering of elements.
- How Collections.Sort() works ?
  1. When it is called then it converts list to arrays by calling toArray() method on list.
  2. When array 'a' is obtained which contains all list elements then Arrays.sort(a) method is called.
  3. Arrays class contains sort methods for almost all kinds of primitives arrays.
  4. So whichever is the element type of list, example String that particular sort method is called.
  5. Now another requirement is each element type should implement Comparable<T> interface. This interface contains single method compareTo(T o) which defines the natural ordering of the element.

6. So now inside Arrays.sort() method particular compareTo() method is called to get their natural ordering.
7. Below are list of classes which implements Comparable<T> interface with their natural ordering.

Class	Natural Ordering
Byte	Signed numerical
Character	Unsigned numerical
Long	Signed numerical
Integer	Signed numerical
Short	Signed numerical
Double	Signed numerical
Float	Signed numerical
BigInteger	Signed numerical
BigDecimal	Signed numerical
Boolean	Boolean.FALSE < Boolean.TRUE
File	System-dependent lexicographic on path name
String	Lexicographic
Date	Chronological
CollationKey	Locale-specific lexicographic

- If one want to use ordering other than natural ordering then one can use another interface Comparator.

```
public interface Comparator<T>
   {
     int compare(T o1, T o2);
   }
```

- compare() method has two arguments returning negative, zero or positive integer same as compareTo().

Cloneable interface :

- This interface doesn't contain any method but interface itself is a indication of Object.clone() method. The interface Cloneable under java.lang is as

```
public interface Cloneable
 {

 }
```

i.e. in reality class which implements Cloneable actually override Object.clone() method. By convention, classes that implement this interface should override Object.clone (which is protected) with a public method.

26. Map Interfaces :

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends **Map** so that the keys are maintained in ascending order.

- Map Interface

```
public interface Map<K,V>
```

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.
Set<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map.
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map** and contains the same entries. Otherwise, returns **false**.
V get(Object *k*)	Returns the value associated with the key *k*. Returns **null** if the key is not found.
int hashCode( )	Returns the hash code for the invoking map.

boolean isEmpty( )	Returns **true** if the invoking map is empty. Otherwise, returns **false**.
Set<K> keySet( )	Returns a **Set** that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
V put(K *k*, V *v*)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are *k* and *v*, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> *m*)	Puts all the entries from *m* into this map.
V remove(Object *k*)	Removes the entry whose key equals *k*.
int size( )	Returns the number of key/value pairs in the map.
Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

- Map stores entries in key/value pairs where both keys and values are objects. Keys must be unique where values can be duplicated.
- Maps don't implement iterable interface hence its not possible to iterate through maps. However, after getting collection-view of maps its possible to use either iterator or use for-loop.
- Maps are part of collection framework although they don't implement collection interface. One can have collection-view of map using entrySet(), keyset() and values() methods as mentioned above.

27. SortedMap :

- `public interface SortedMap<K,V> extends Map<K,V>`
- SortedMap extends Map and ensures that all its entries are maintained in ascending order according to keys natural ordering, or according to a comparator provided at the time of SortedMap creation.
- Methods defined by SortedMap are summarized below

Method	Description
Comparator<? super K> comparator( )	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, **null** is returned.
K firstKey( )	Returns the first key in the invoking map.
SortedMap<K, V> headMap(K *end*)	Returns a sorted map for those map entries with keys that are less than *end*.
K lastKey( )	Returns the last key in the invoking map.
SortedMap<K, V> subMap(K *start*, K *end*)	Returns a map containing those entries with keys that are greater than or equal to *start* and less than *end*.
SortedMap<K, V> tailMap(K *start*)	Returns a map containing those entries with keys that are greater than or equal to *start*.

28. NavigableMap :

- `public interface` **`NavigableMap<K,V>`** `extends SortedMap<K,V>`
- NavigableMap declares the behaviour of a map that supports the retrieval of entries based on the closest match to a given key or keys.
-

Method	Description
Map.Entry<K,V> ceilingEntry(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K ceilingKey(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> descendingKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.
NavigableMap<K,V> descendingMap( )	Returns a **NavigableMap** that is the reverse of the invoking map. The resulting map is backed by the invoking map.
Map.Entry<K,V> firstEntry( )	Returns the first entry in the map. This is the entry with the least key.
Map.Entry<K,V> floorEntry(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K floorKey(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting map is backed by the invoking map.
Map.Entry<K,V> higherEntry(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K higherKey(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
Map.Entry<K,V> lastEntry( )	Returns the last entry in the map. This is the entry with the largest key.
Map.Entry<K,V> lowerEntry(K obj)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K lowerKey(K *obj*)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.

NavigableSet<K> navigableKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map. The resulting set is backed by the invoking map.
Map.Entry<K,V> pollFirstEntry( )	Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. **null** is returned if the map is empty.
Map.Entry<K,V> pollLastEntry( )	Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. **null** is returned if the map is empty.
NavigableMap<K,V> subMap(K *lowerBound*, boolean *lowIncl*, K *upperBound* boolean *highIncl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *highIncl* is included. The resulting map is backed by the invoking map.
NavigableMap<K,V> tailMap(K *lowerBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting map is backed by the invoking map.

29. Map.Entry<K, V> Interface :

- public static interface **Map.Entry<K,V>**
- This interface enables one to work with Map entries. Recall the entrySet() method declared by map returns collection view of the map, whose elements are of this class.
- The only way to obtain reference to a map entry is from the iterator of this collection-view. These Map.Entry objects are valid only for the duration of the iteration i.e. the behaviour of the map entry is undefined if the backing map has been modified after the entry was returned by the iterator, except through the setValue operation of the map entry.
- Methods defined by Map.Entry interface are as below.

Method	Description
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map.Entry** whose key and value are equal to that of the invoking object.
K getKey( )	Returns the key for this map entry.
V getValue( )	Returns the value for this map entry.
int hashCode( )	Returns the hash code for this map entry.
V setValue(V *v*)	Sets the value for this map entry to *v*. A **ClassCastException** is thrown if *v* is not the correct type for the map. An **IllegalArgumentException** is thrown if there is a problem with *v*. A **NullPointerException** is thrown if *v* is **null** and the map does not permit **null** keys. An **UnsupportedOperationException** is thrown if the map cannot be changed.

30. Map Classes :

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.

AbstractMap is a super class of all the concrete map class implementation.

- There are three general purpose map classes HashMap, LinkedHashMap and TreeMap. HashMap doesn't guarantee about the order of its elements, LinkedHashMap is nothing but a hash map with a linked list running through it. TreeMap like tree set order its elements in their natural order or according to the comparator provided.
- Example :

```
package collections;

import java.util.*;

public class DifferentFlavoursOfMaps {

   public static void main(String[] args)
   {
          HashMap<String, String> hm = new HashMap<String, String>();
          hm.put("jj", "nine");
          hm.put("jay", "five");
          hm.put("how", "seven");
          hm.put("fin", "one");
          hm.put("good", "three");

          System.out.println("Hash map contains "+hm);

          LinkedHashMap<Integer, String> lhm = new LinkedHashMap<Integer,
          String>();
          lhm.put(9, "nine");
          lhm.put(5, "five");
          lhm.put(7, "seven");
          lhm.put(1, "one");
          lhm.put(3, "three");
          System.out.println("Linked hash map contains "+lhm);

          TreeMap<String, String> tm1 = new TreeMap<String, String>();
          tm1.putAll(hm);
          System.out.println("Tree map contains "+tm1);

          TreeMap<Integer, String> tm2 = new TreeMap<Integer, String>();
          tm2.putAll(lhm);
```

```
            System.out.println("Tree map contains "+tm2);
    }
}
Output:
Hash map contains {how=seven, good=three, fin=one, jay=five, jj=nine}
Linked hash map contains {9=nine, 5=five, 7=seven, 1=one, 3=three}
Tree map contains {fin=one, good=three, how=seven, jay=five, jj=nine}
Tree map contains {1=one, 3=three, 5=five, 7=seven, 9=nine}
```

31. HashMap Class :

- `public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable`
- HashMap is HashTable based implementation of the map. HashMap class is roughly equivalent of HashTable except it is unsynchronized and permits null. i.e. HashMap permits both null values and null keys.
- HashMap makes no guarantee about the order of the map.
- Constructors :

Constructor and Description
HashMap()   Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
HashMap(int initialCapacity)   Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).
HashMap(int initialCapacity, float loadFactor)   Constructs an empty HashMap with the specified initial capacity and load factor.
HashMap(**Map**<? extends **K**,? extends **V**> m)   Constructs a new HashMap with the same mappings as the specified Map.

- Capacity is nothing but number of buckets in the Hash table and initial capacity is nothing but capacity of the hash map when it is created. The load factor is measure of how full the hash table is allowed to get before its capacity is automatically increased. By default load factor is 0.75. When number of entries in hash table exceeds the product of load factor and the current capacity then the hash table is rehashed. So that the hash table will have twice the number of buckets.

- Example :

```
package collections;

import java.util.*;

public class HashMapClass
{
   public static void main(String[] args)
   {
         HashMap<Integer,String> hmc = new HashMap<Integer,String>();
         hmc.put(5, "five");
         hmc.put(5, "five");
         hmc.put(1, "one");
         hmc.put(8, "eight");
         hmc.put(3, "three");
         hmc.put(6, "six");
         hmc.put(11, "elevan");
         hmc.put(2, "two");
         hmc.put(null, "Null");
         hmc.put(4, "four");

         System.out.println(hmc);

         //get method
         System.out.println("value for key 8 is "+hmc.get(8));

         //containsKey and containsValue
         System.out.println("contains key 11? "+hmc.containsKey(11));
         System.out.println("contains value Null?
         "+hmc.containsValue("Null"));
         System.out.println("contains value jj?
         "+hmc.containsValue("jj"));

         //size and isEmpty
         System.out.println("size of hash map - "+hmc.size());
         System.out.println("hash map is empty? "+hmc.isEmpty());

         //entrySet, keySet, values
         System.out.println("entry set with key-value "+hmc.entrySet());
         System.out.println("key set "+hmc.keySet());
         System.out.println("only values "+hmc.values());

         //remove and clear
       hmc.remove(4);
       System.out.println("hash map after removing key 4 "+hmc);
       hmc.clear();
       System.out.println("hash map after clear - "+hmc);
   }
}
Output:
{null=Null, 1=one, 2=two, 3=three, 4=four, 5=five, 6=six, 8=eight,
11=elevan}
value for key 8 is eight
contains key 11? true
contains value Null? true
contains value jj? false
size of hash map - 9
hash map is empty? false
entry set with key-value [null=Null, 1=one, 2=two, 3=three, 4=four,
5=five, 6=six, 8=eight, 11=elevan]
```

```
key set [null, 1, 2, 3, 4, 5, 6, 8, 11]
only values [Null, one, two, three, four, five, six, eight, elevan]
hash map after removing key 4 {null=Null, 1=one, 2=two, 3=three,
5=five, 6=six, 8=eight, 11=elevan}
hash map after clear - {}
```

32. TreeMap Example :

```
package collections;

import java.util.*;

public class TreeMapClass
{
   public static void main(String[] args)
   {
         TreeMap<Integer, String> tm = new TreeMap<Integer, String>();
         tm.put(4, "four");
         tm.put(2, "two");
         tm.put(10, "ten");
         tm.put(6, "six");
         tm.put(8, "eight");

         System.out.println("Content of tree map are "+tm);

         //SortedMap methods
         System.out.println("*****SortedMap Methods********");

         //firstKey and lastKey
         System.out.println("-----firstKey and lastKey-----");
         System.out.println("first key in tm is "+tm.firstKey());
         System.out.println("last key in tm is "+tm.lastKey());

         //headMap, tailMap and subMap
         System.out.println("-----headMap, tailMap and subMap-----");
         System.out.println("head map for 6 "+tm.headMap(6));
         System.out.println("tail map for 8 "+tm.tailMap(8));
         System.out.println("sub map for 3 and 8"+tm.subMap(3, 8));

         //NavigableMap methods
         System.out.println("*****NavigableMap Methods*******");

         //descending keyset and descending map
         System.out.println("-----descending keyset and map------");
         System.out.println("descending keySet "+tm.descendingKeySet());
         System.out.println("descending map "+tm.descendingMap());

         //ceiling and floor both keys and entries
         System.out.println("-----ceiling and floor keys and entries----
         -");
         System.out.println("Ceiling key and entry for 2 is
         "+tm.ceilingKey(2)+" and "+tm.ceilingEntry(2));
         System.out.println("Ceiling key and entry for 5 is
         "+tm.ceilingKey(5)+" and "+tm.ceilingEntry(5));
         System.out.println("Floor key and entry for 9 is
         "+tm.floorKey(9)+" and "+tm.floorEntry(9));
```

```
        //higher and lower both keys and entries
        System.out.println("-----higher and lower keys and entries-----
        ");
        System.out.println("highest keys and entries for 5 is
        "+tm.higherKey(5)+" and "+tm.higherEntry(5));
        System.out.println("lowest keys and entries for 9 is
        "+tm.lowerKey(9)+" and "+tm.lowerEntry(9));

        //headMap, tailMap and subMap
        System.out.println("-----headMap, tailMap and subMap-----");
        System.out.println("head map for 7 is "+tm.headMap(7, true));
        System.out.println("tail map for 4 is "+tm.tailMap(4, true));
        System.out.println("sub map for 3 and 10 is "+tm.subMap(3,
        false, 10, false));

        //first and last both keys and entries
        System.out.println("------first and last keys and entries------
        -");
        System.out.println("first key is "+tm.firstKey());
        System.out.println("first entry is "+tm.firstEntry());
        System.out.println("last key is "+tm.lastKey());
        System.out.println("last entry is "+tm.lastEntry());

        //Navigable set
        System.out.println("------Navigable set------");
        System.out.println("Navigable key set "+tm.navigableKeySet());

        //pollfirst and polllast methods
        System.out.println("------pollFirst and pollLast------");
        tm.pollFirstEntry();
        System.out.println("after pollFirst entry "+tm);
        tm.pollLastEntry();
        System.out.println("after pollLast entry "+tm);

    }
}
```

```
Output:
Content of tree map are {2=two, 4=four, 6=six, 8=eight, 10=ten}
*****SortedMap Methods********
-----firstKey and lastKey-----
first key in tm is 2
last key in tm is 10
-----headMap, tailMap and subMap-----
head map for 6 {2=two, 4=four}
tail map for 8 {8=eight, 10=ten}
sub map for 3 and 8{4=four, 6=six}
*****NavigableMap Methods*******
-----descending keyset and map------
descending keySet [10, 8, 6, 4, 2]
descending map {10=ten, 8=eight, 6=six, 4=four, 2=two}
-----ceiling and floor keys and entries-----
Ceiling key and entry for 2 is 2 and 2=two
Ceiling key and entry for 5 is 6 and 6=six
Floor key and entry for 9 is 8 and 8=eight
-----higher and lower keys and entries-----
highest keys and entries for 5 is 6 and 6=six
lowest keys and entries for 9 is 8 and 8=eight
-----headMap, tailMap and subMap-----
head map for 7 is {2=two, 4=four, 6=six}
tail map for 4 is {4=four, 6=six, 8=eight, 10=ten}
sub map for 3 and 10 is {4=four, 6=six, 8=eight}
```

```
------first and last keys and entries-------
first key is 2
first entry is 2=two
last key is 10
last entry is 10=ten
------Navigable set------
Navigable key set [2, 4, 6, 8, 10]
------pollFirst and pollLast------
after pollFirst entry {4=four, 6=six, 8=eight, 10=ten}
after pollLast entry {4=four, 6=six, 8=eight}
```