UNIT-IV

THE COLLECTIONS FRAMEWORK

What is Collection?

A collection is an object which can store a group of other object.

What is Framework?

A Framework provides ready-made architecture and represents a group of classes and interface .

What is Collection Framework?

1. Collection framework is implemented in java.util package.
2. **Collection** framework provides a set of interfaces and classes to implement various data structures and algorithms.
3. Collection classes in java used to manage the data very efficiently like inserting, deleting, updating, retrieving, sorting the data etc.
4. It is one of the standardized mechanisms which allow us to group multiple values either of same types or different type or both the types in a single variable with dynamic size in nature. This single variable is known as collection framework variable.

Benefits of Collections Framework in Java

I. Reusability: Java Collections Framework provides common classes and utility methods than can be used with different types of collections. This promotes the reusability of the code. A developer does not have to re-invent the wheel by writing the same method again.

II. Quality: Using Java Collection Framework improves the program quality, since the code is already tested and used by thousands of developers.

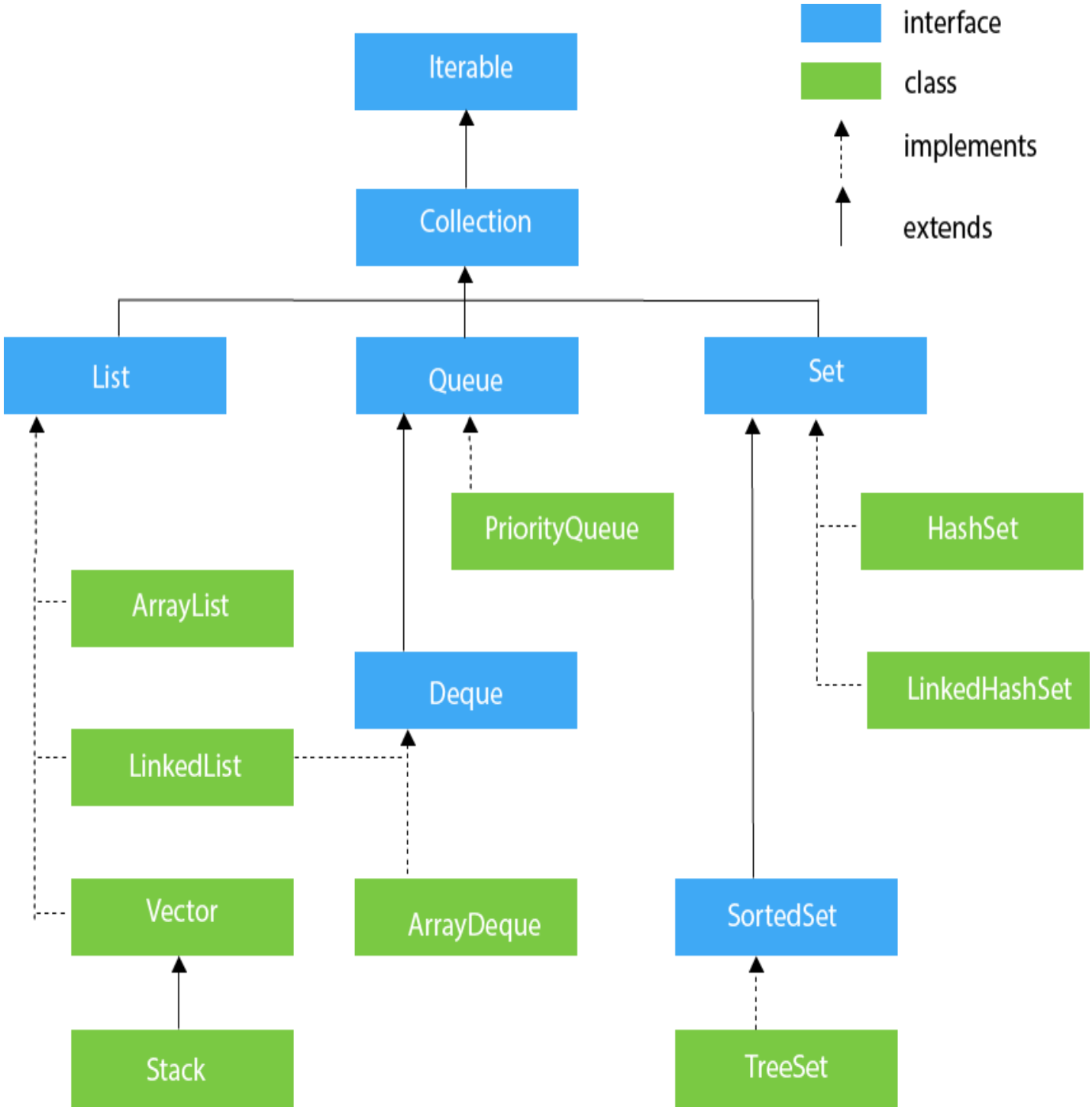
III. Speed: Most of programmers report that their development speed increased since they can focus on core logic and use the generic collections provided by Java framework.

IV. Maintenance: Since most of the Java Collections framework code is open source and API documents is widely available, it is easy to maintain the code written with the help of Java Collections framework. One developer can easily pick the code of previous developer.

V. Reduces effort to design new APIs: This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections instead, they can use standard collection interfaces.

HIERARCHY OF COLLECTION FRAMEWORK

The java.util package contains all the classes and interfaces for the Collection framework.



In the above image, blue part refers to the different interfaces and the green part defines the class.

COLLECTION INTERFACE:

The Collection interface is the root interface of the collections framework hierarchy. The Collection interface is available inside the java.util package. The Collection interface extends Iterable interface. Java does not provide direct implementations of the Collection interface but provides implementations of its subinterfaces like List, Set, and Queue.

Methods of Collection Interface : The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its subinterfaces.

1. **boolean add (element obj) :** This method appends the specified element to the end of the ArrayList. If the element is added successfully then the preceding method returns true.
2. **int size() :** This method returns the number of elements present in the ArrayList.
3. **element remove (int position) :** This method removes the element at the specified position in the ArrayList.
4. **addAll() :** adds all the elements of a collection to another collection
5. **iterator() :** returns an iterator to access elements of the collection
6. **removeAll() :** removes all the elements of the specified collection from the collection
7. **clear()** : removes all the elements of the collection

Collection Interface is divided into three types of interfaces.

1. List Interface
2. Queue Interface
3. Set Interface

1. LIST INTERFACE: The List interface is a child interface of the Collection interface. The List interface is Available inside the java.util package.

- The List interface extends **Collection** interface.
- The List interface allows duplicate elements.
- The List interface preserves the order of insertion.
- The List allows accessing the elements based on the index value that starts with zero.

List interface is implemented by the classes

- Array List
- Linked List
- Stack
- Vector

The Collection classes

ARRAYLIST CLASS:

1. ArrayList is part of Java's collection framework and implements List interface.
2. An ArrayList is like an array.
3. ArrayList can grow in memory dynamically. It means that when we store elements into the ArrayList, depending on the number of elements, the memory is dynamically allotted and re-allotted to accommodate all the elements.
4. ArrayList allows duplicate and null values.
5. ArrayList is an ordered collection. It maintains the insertion order of the elements.
6. You cannot create an ArrayList of primitive types like int, char etc. You need to use boxed types like Integer, Character, Boolean etc.
7. ArrayList is not synchronized. This means that when more than one thread acts simultaneously on the ArrayList object, the results may be incorrect in some cases.

ArrayList Constructor:

The ArrayList class in Java provides the following constructor methods to create the ArrayList.

1. ArrayList a1=new ArrayList();

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

2. ArrayList a1=new ArrayList**(int InitalCapacity);**

   Creates an empty Array List object with specified initial capacity .

ArrayList Class Methods:

ArrayList class includes the following methods.

1. **boolean add (element obj) :** This method adds an element to the arraylist. It returns true if the element is added successfully.
2. **boolean addAll(element obj)** : This method adds each element of the given element at the end of the Array List, It returns True if element is added successfully, and returns false if it is not.
3. **int size() :** This method returns the number of elements present in the list.
4. **boolean contains(Object obj) :** This method returns true if the calling Array list object has the specific element present as given in the argument list, otherwise it returns false.
5. **element remove (int position) :** This method removes an element at the specified position in the Array list.

6. **void clear() :** This method removes all the elements from the Array list.
7. **iterator() :** returns an iterator to access elements of the ArrayList.

Program:

```
import java.util.*;
class ArrayListDemo
{
	public static void main(String args[])
	{
		ArrayList<Integer> alist=new ArrayList< Integer >();
		System.out.println("Initial list of elements: "+allist);
		alist.add(20);
		alist.add(50);
		alist.add(45);
		alist.add(67);
		alist.add(72);
		alist.add(90);
		alist.add(88);
		alist.add(74);
		alist.add(100);
		alist.add(99);
		System.out.println("After invoking add method: "+ alist);
		System.out.println("Elements using Iterator :");
		Iterator<Integer> it = alist.iterator();
		System.out.print("Display the ArrayList: ");
		while(it.hasNext())
		{
			System.out.print(" "+it.next());
		}
		System.out.println("Element to be removed:" + alist.remove(2));
		System.out.println("ArrayList:" + alist);
		System.out.println("Array List size:" + alist.size());
	}
}
```

LINKEDLIST CLASS

Linked List class uses a doubly linked list. A linked List contains a group of elements in the form of nodes. Each node will have three fields.

previous	Data	next

Previous - stores an address of the previous element in the list. It is null for the first element

Next - stores an address of the next element in the list. It is null for the last element

Data - stores the actual data

Properties of Linked List class

1. The underlying data structure is Double Linked List.
2. LinkedList class can contain duplicate elements..
3. LinkedList class maintains insertion order.
4. Heterogeneous objects are allowed.
5. In LinkedList class Insertion of NULL values are allowed.
6. LinkedList class is non synchronized.
7. Linked List class, manipulation is fast because no shifting needs to occur.

Constructors of LinkedList

1. **LinkedList ll = new LinkedList();**

   Creates an empty LinkedList object.

2. **LinkedList ll = new LinkedList(Collection c);**

   Creates an equivalent LinkedList object for the given Collection.

LinkedList Class Methods:

LinkedList class includes the following methods.

1. **boolean add (element obj) :** This method adds an element to the linked list. It returns true if the element is added successfully.
2. **int size() :** This method returns the number of elements present in the list.
3. **void addFirst(element obj) :** This method adds the first element from the linked list
4. **void addLast(element obj) :** This method adds the specified element to the end of the linked list.
5. **element remove (int position) :** This method removes an element at the specified position in the linked list.
6. **element removeFirst() :** This method removes the first element from the linked list and returns it.

7. **element removeLast() :** This method removes the last element from the linked list and returns it.
8. **void clear() :**This method removes all the elements from the linked list.
9. **element getFirst() :** This method returns the first element from the list.
10. **element getLast() :** This method returns the last element from the list.

Program:

```
import java.util.*;
class LinkedListDemo
{
        public static void main(String args[])
        {
                LinkedList<Integer> llist=new LinkedList< Integer >();
                System.out.println("Initial list of elements: "+llist);
                alist.add(20);
                alist.add(50);
                alist.add(45);
                alist.add(67);
                alist.add(72);
                alist.add(90);
                System.out.println("After invoking add method: "+ llist);
                llist.addFirst(10);
                System.out.println("After invoking addFirst method: "+ llist);
                llist.addLast(125);
                System.out.println("After invoking addLast method: "+ llist);
                System.out.println("Elements using Iterator :");
                Iterator<Integer> it = llist.iterator();
                System.out.print("Display the LinkedList: ");
                while(it.hasNext())
                {
                        System.out.print(" "+it.next());
                }
                System.out.println("Element to be removed:" + llist.remove(2));
                System.out.println("LinkedList:" + llist);
                llist.removeFirst();
```

```
        System.out.println("After invoking removeFirst() method: "+llist);
        llist.removeLast();
        System.out.println("After invoking removeLast() method: "+llist);
        System.out.println("Linked List:" + llist);
        System.out.println("linked List size:" + llist.size());
        System.out.println("First Element from the List:"+llist.getFirst());
        System.out.println("Last Element from the List:"+llist.getLast());

    }
}
```

3.Vector Class

1. Vector class is part of Java's collection framework and implements List interface.
2. Vector Class is similar to ArrayList.
3. Vector class can grow in memory dynamically. It means that when we store elements into the Vector, depending on the number of elements, the memory is dynamically allotted and re-allotted to accommodate all the elements.
4. Vector Class allows duplicate and null values.
5. Vector is synchronized. it means even if several threads act on Vector object simultaneously, the results will be reliable.
6. Vector increases its size every time by doubling it. (100 PERCENT)

Note: Vector contains many legacy methods that are not part of collection framework

Vector Class Constructor:

The Vector class in Java provides the following constructor methods to create the Vector.

1. **Vector ():** It creates an empty vector with default initial capacity of 10.

   Syntax 1 : Vector<data_type> list_name = new Vector< data_type >();

   Example: Vector<Integer> v1 = new Vector< Integer >();

Vector Methods In JAVA:

Vector contains many legacy methods that are not part of collection framework .

1. **boolean add (element obj) :** This method adds an element to the vector. It returns true if the element is added successfully.
2. **int size() :** This method returns the number of elements present in the list.

3. **void addElement(Element obj) :** This method adds the specified element at the end of the vector also increasing its size by 1.
4. **int capacity() :** This method gives the capacity of the vector.
5. **Enumeration elements():** This method returns an object of Enumeration class that can be used for traversing over elements of vector.
6. **Object firstElement() :** This method returns the first element at index 0.
7. **Object lastElement() :** This method returns the last element of the vector
8. **boolean isEmpty() :** This method checks weather vector is empty or not .
9. **element remove (int position) :** This method removes an element at the specified position in the linked list.
10. **void clear() :**This method removes all the elements from the Vector.

Program :

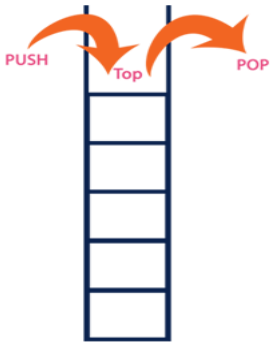
```
import java.util.*;
class VectorDemo
{
        public static void main(String args[])
        {
                Vector<Integer> v = new Vector<Integer>();
                v.add(3);
                v.add(5);
                v.add(4);
                v.add(1);
                v.add(2);
                v.add(8);
                v.add(8);
                System.out.println("Values in Vector Object  :" +v);
                Enumeration e=v.elements();
                System.out.println("Data in enum object :");
                while (e.hasMoreElements())
                {
                        System.out.println("value: " + e.nextElement());
                }
                System.out.println("value at first index :" +v.firstElement());
```

```
                System.out.println("Last element of vector object is : "+v.lastElement());
                System.out.println("value at  index 2: " +v.get(2));
                System.out.println("removed element at index 2  : " +v.remove(2));
                v.set(1, 15);
                System.out.println("Values in vector :" +v);
                System.out.println("Size of the vector is : " +v.size());
                v1.clear();
                System.out.println("Values in vector1 :" +v1);
        }
}
```

4.STACK CLASS

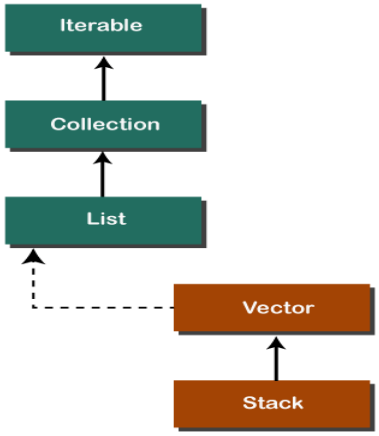
1. A stack represents a group of elements stored in **LIFO (Last In First Out)** principle.
2. Stack is a linear data structure in which the insertion and deletion operations are performed at only **one end**.
3. In a stack, adding and removing of elements are performed at a single position which is known as **"top".**
4. That means, a new element is added at top of the stack and an element is removed from the top of the stack.

Stack Representation is



In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

The stack class arranged in the Collections framework hierarchy, as shown below.



How to create a Stack Class Object

If we want to create a stack, first, import the java.util package and create an object of the Stack class.

Syntax: Stack s = **new** Stack();

[or]

Stack<data type > s = **new** Stack<data type > ();

Methods of the Stack Class

1. **element push(element obj) :** This method pushes an element obj onto the top of the stack and returns that element.
2. **element pop() :** This method pops the top most element from the stack and returns it.
3. **element peek() :** This method returns the top most element from the stack.
4. **boolean empty() :** This method tests whether the stack is empty or not. If the stack is empty then true is returned otherwise false.
5. **int search(Object obj) :** This method returns the position of an element from the top of the stack. If the element is not found in the stack then it returns -1.

Program :

```
import java.util.*;
class MyClass
{
        public static void main (String[] args)
        {
                Stack<Integer> even = new Stack<>();
                s.push(0);
                s.push(2);
                s.push(4);
                s.push(6);
                s.push(8);
                s.push(10);
                s.push(12);
                s.push(14);
                s.push(16);
                System.out.println("Print Stack before pop:"+s);
                System.out.println("pop => " + s.pop());
                System.out.println("pop => " + s.pop());
                System.out.println("Print Stack after pop:"+s);
                System.out.println("Number on top of the stack is => " + s.peek());
                System.out.println("Is stack empty?  Ans:" + s.empty());
                System.out.println("Position of 8 from the top is " + s.search(8));
                System.out.println("Position of 20 from the top is " + s.search(20));
        }
}
```

2.QUEUE INTERFACE: The **Queue** interface is a child interface of the Collection interface. The Queue interface is available inside the **java.util** package.

- The Queue interface extends Collection interface.
- The Queue interface allows duplicate elements.
- The Queue interface preserves the order of insertion.

Queue interface is implemented by the classes

- PriorityQueue
- LinkedList

PriorityQueue Class

1. PriorityQueue is a child class of Queue interface.
2. The elements of PriorityQueue are organized as the elements of queue data structure, but it does not follow FIFO principle.
3. The PriorityQueue elements are organized based on the priority heap.
4. The PriorityQueue class is used to create a dynamic queue of elements that can grow as needed.
5. The PriorityQueue allows to store duplicate data values, but not null values.
6. The PriorityQueue maintains the order of insertion.
7. Removal of elements always takes place from the front end (head) of queue.
8. PriorityQueue is not synchronized. That means it is not thread-safe. For working in a multithreading environment.

PriorityQueue class constructors

1. **PriorityQueue( )** - Creates an empty PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
   Example : PriorityQueue q = new PriorityQueue();

PriorityQueue Methos :

1. **add():** The add() operation is used to insert new element into the PriorityQueue. If it performs insert operation successfully, it returns “true” value. Otherwise it throws java.lang.IllegalStateException.
2. **offer():** The offer() operation is used to insert new element into the PriorityQueue. If it performs insert operation successfully, it returns “true” value. Otherwise it returns “false” value.
3. **element():** The element() operation is used to retrieve an element from the head of the PriorityQueue, without removing it. If it performs examine operation successfully, it returns the head element of the PriorityQueue. Otherwise it throws java.util.NoSuchElementException.

4. **peek():** The peek() operation is used to retrieve an element from the head of the PriorityQueue, without removing it. If it performs examine operation successfully, it returns the head element of the queue. Otherwise it returns null value.
5. **poll():** The poll() operation is used to delete an element from the head of the PriorityQueue. If it performs delete operation successfully, it returns the head element of the PriorityQueue. Otherwise it returns "null" value.
6. **remove():** The remove() operation is used to delete an element from the head of the PriorityQueue. If it performs delete operation successfully, it returns the head element of the PriorityQueue. Otherwise it throws java.util.NoSuchElementException.
7. **size():** This method returns the count of the element available in the PriorityQueue.
8. **void clear() :** This method remove all the elements from the PriorityQueue.
9. **boolean isEmpty( ) :** This method checks whether the PriorityQueue is empty or not. If it is empty this method will return true, else it will return false.

```
import java.util.*;
class PriorityQueueDemo
{
   public static void main(String[] args)
   {
            PriorityQueue q = new PriorityQueue();
                    q.add(20);
                    q.add(40);
                    q.add(60);
                    q.add(80);
                    q.add(100);
                    q.offer(200);
                     q.offer(180);
                    q.offer(120);
            System.out.println("\nTotal item count in PriorityQueue: " + q.size());
            System.out.println("\nItems in PriorityQueue: " + q);
            System.out.println("\nHead item of the PriorityQueue: " + q.element());
            q.remove();
            System.out.println("\nAvailable item in PriorityQueue: " + q);
            System.out.println("\nHead item of the PriorityQueue: " + q.peek());
            q.poll();
            System.out.println("\nAvailable item in PriorityQueue: " + q);
```

```
        System.out.println("\nHead item of the PriorityQueue: " + q.contains(80));
        System.out.println("The PriorityQueue elements through iterator:");
        Iterator it = q.iterator();
        while(it.hasNext())
        {
                System.out.println( "  "+it.next());
        }
        q.clear();
        System.out.println("The PriorityQueue elements are:"+q);
        System.out.println("\nHead item of the PriorityQueue: " + q.peek());
        System.out.println("\nHead item of the PriorityQueue: " + q.poll());
        System.out.println("\nHead item of the PriorityQueue: " + q.element());
        System.out.println("\nHead item of the PriorityQueue: " + q.remove());
    }
}
```

DEQUE INTERFACE: The **Deque** interface is a child interface of the Queue interface. The Deque interface is available inside the **java.util** package.

- The Deque interface extends **Queue** interface.
- The Deque interface allows duplicate elements.
- The Deque interface preserves the order of insertion.

Deque interface is implemented by the classes

- ArrayDeque

ARRAYDEQUE CLASS

1. ArrayDeque in Java is a concrete class that creates a dynamic array (resizable array) to store its elements from both sides of the queue.
2. It provides a resizable-array implementation of deque interface. Therefore, it is also known as array double-ended queue or simply array deck.
3. Since ArrayDeque has no capacity restrictions, ArrayDeque is especially useful for implementing stacks and queues whose sizes are not known in advance.
4. Array deque is not synchronized. That means it is not thread-safe. Multiple threads can access the same ArrayDeque object at the same time.
5. Null elements are restricted in the ArrayDeque.
6. ArrayDeque class performs faster operations than Stack when used as a stack.
7. ArrayDeque class performs faster operations than LinkedList when used as a queue.

Constructors of ArrayDeque class

ArrayDeque(): This constructor creates an empty array deque with starting capacity of 16 elements.

The general syntax to create array deque object is as follows:

ArrayDeque<E> ad = new ArrayDeque<E>();

Methods of ArrayDeque

1. **add():** The add() operation is used to insert new element into the queue. If it performs insert operation successfully, it returns "true" value. Otherwise it throws java.lang.IllegalStateException.
2. **void addFirst(Object o):** It inserts the specified element to the front of the deque.
3. **void addLast(Object o):** It inserts the specified element to the last of deque.
4. **offer():** The offer() operation is used to insert new element into the queue. If it performs insert operation successfully, it returns "true" value. Otherwise it returns "false" value.
5. **boolean offerFirst(Object element):** It inserts the specified element at the front of deque.
6. **boolean offerLast(Object element):** It inserts the specified element at the end of deque.
7. **remove():** The remove() operation is used to delete an element from the head of the queue. If it performs delete operation successfully, it returns the head element of the queue. Otherwise it throws java.util.NoSuchElementException.
8. **boolean removeFirst():** This method is used to retrieve and remove the first element from the deque.
9. **boolean removeLast():** This method is used to retrieve and remove the last element from the deque.
10. **poll():** The poll() operation is used to delete an element from the head of the queue. If it performs delete operation successfully, it returns the head element of the queue. Otherwise it returns "null" value.
11. **Object pollFirst():** The pollFirst() method retrieves and removes the first element of deque. It returns null element if the deque is empty.
12. **Object pollLast():** The pollLast() method retrieves and removes the last element of deque. It returns null element if the deque is empty.
13. **Object getFirst():** This method is used to retrieve the first element of deque. It does not remove element.
14. **Object getLast():** This method is used to retrieve the last element of deque. It does not remove element.
15. **boolean isEmpty():** The isEmpty() method returns true if deque has no elements.

16. **peek():** The peek() operation is used to retrieve an element from the head of the queue, without removing it. If it performs examine operation successfully, it returns the head element of the queue. Otherwise it returns null value.
17. **Object peekFirst():** The peekFirst() method retrieves the first element from the head of deque but does not remove it. It returns null element if the deque is empty.
18. **Object peekLast():** The peekLast() method retrieves the last element from deque but does not remove it. It returns null element if the deque is empty.
19. **element():** The element() operation is used to retrieve an element from the head of the queue, without removing it. If it performs examine operation successfully, it returns the head element of the queue. Otherwise it throws java.util.NoSuchElementException.
20. **void clear():** This method is used to remove all elements from deque.

```
import java.util.*;
class ArrayDequeDemo
{
  public static void main(String[] args)
  {
        ArrayDeque<Integer> ad = new ArrayDeque<Integer> ();
        ad.add(20);
        ad.add(40);
         ad.add(60);
        ad.add(80);
        ad.addFirst(7);
        ad.offer(100);
        ad.offerFirst(35);
        ad.offer(200);
        ad.offer(180);
        ad.offerLast(45);
        ad.offerFirst(55);
        ad.offerLast(95);
        ad.addLast(79);
        System.out.println("\nTotal item count in ArrayDeque: " + ad.size());
        System.out.println("\nItems in ArrayDeque: " + ad);
```

```
        System.out.println("Deleted Element: " + ad.remove());
        System.out.println("Deleted First Element: " + ad.removeFirst());
        System.out.println("Deleted Last Element: " + ad.removeLast());
        System.out.println("\nItems in ArrayDeque: " + ad);
        System.out.println("Deleted poll Element: " + ad.poll());
        System.out.println("Deleted pollFirst Element: " + ad.pollFirst());
        System.out.println("Deleted pollLast Element: " + ad.pollLast());
        System.out.println("First Element is(using peekFirst()): " + ad.peekFirst());
        System.out.println("Last Element is(using PeekLast()): " +ad.peekLast());
        System.out.println("First element is: " + ad.getFirst());
        System.out.println("Last element is: " + ad.getLast());
        System.out.println("\nItems in ArrayDeque: " + ad);
        Iterator i = ad.iterator();
        System.out.printf("The Elements are :");
        while(i.hasNext())
        {
                System.out.print(i.next()+"  ");
        }
    }
}
```

SET INTERFACE

1. Set interface is the child interface of Collection. The Set interface is available inside the java.util package.
2. If we want to represent a group of individual objects as a single entity, where duplicate elements are not allowed and insertion order is not preserved then we should go for Set Interface.

SortedSet interface is implemented by the classes

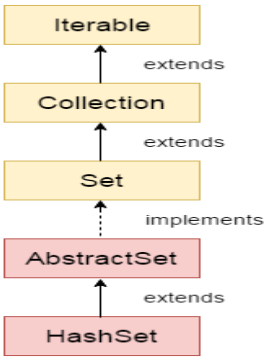
1.HashSet

2.LinkedHashSet

HASHSET CLASS

1. HashSet class Duplicate elements are not allowed. If we trying to insert duplicate elements, we won't get any compiletime or runtime errors. add() method simply returns false.
2. Insertion order is not preserved and all objects will be inserted based on hash-code objects.
3. Heterogeneous objects are allowed.
4. In Hashset Insertion of NULL values are allowed.
5. HashSet implements Serializable and Clonable interfaces but not RandomAcess.
6. HashSet is the best Choice, if our frequent operation is Search operation.
7. HashSet class is non synchronized.

Hierarchy of HashSet class



The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

Constructors of HashSet

HashSet h = new HashSet();

Creates an empty HashSet object with default initial capacity is 16 & default Load Factor is 0.75.

Methods of HashSet Class

1. **boolean add(element obj) :** This method adds an element to the HashSetet. It returns true if the element is added successfully.
2. **int size() :** This method returns the number of elements in the HashSet.
3. **boolean remove(Object o) :** This method removes first Occurrence of the element specified in the argument list from the hashset , and returns true if that element is removed, or false if it is not.
4. **void clear() :** This method remove all the elements from the set
5. **boolean isEmpty( ) :** This method checks whether the hashset is empty or not. If it is empty this method will return true, else it will return false.
6. **boolean contains(Object obj) :** This method returns true if the calling hashset object contains the specific element as given in the argument list, otherwise it returns false

7. Iterator iterator() : This methods returns the iterator, so that we can traverse over the elements of the hashset.

Program:

```
import java.util.*;
class HashSetDemo
{
	public static void main(String args[])
	{
		HashSet<Integer> hs=new HashSet<Integer> ();
		hs.add(80);
		hs.add(20);
		hs.add(90);
		hs.add(45);
		hs.add(30);
		hs.add(10);
		hs.add(20);
		System.out.println("Display the HashSet :"+hs);
		System.out.println("Size of the HashSet :"+hs.size());
		hs.remove(30);
		System.out.println("Updated HashSet: " + hs);
		Iterator it= hs.iterator();
		System.out.print("HashSet using Iterator: ");
		while(it.hasNext())
		{
			System.out.print(" "+it.next());
		}
		System.out.println("Hash Set Contains '10'" :+hs.contains(10));
		System.out.println("Is the set empty: "+hs.isEmpty());
		hs.clear();
		System.out.println("values in HashSet object After using Clear method"+hs );
	}
}
```

SORTEDSET INTERFACE: The Sorted Set interface is a child interface of the Set interface. The Sorted Set interface is available inside the java.util package.

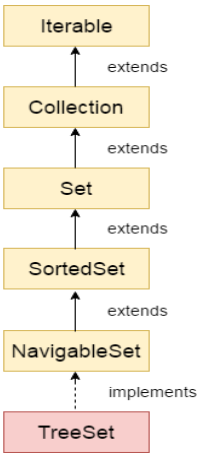
SortedSet interface is implemented by the classes

- TreeSet

TREE SET CLASS

1. TreeSet class is implements Set Interface.
2. In TreeSet all the values are stored in their natural order, like all integer values are stored in ascending order and strings are stored according to Dictionary values.
3. TreeSet is best choice for storing large amount of data.
4. TreeSet class retrieval and access time is very fast, which makes data to be found in no time.
5. Tree Set do not allow null values.
6. TreeSet class do not allow duplicate elements to be stored.
7. TreeSet class is non synchronized.

Hierarchy of TreeSet class



in the above diagram, the Java TreeSet class implements the NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

Constructors of TreeSet

TreeSet ts = new TreeSet();

Creates an empty TreeSet object with default initial capacity is 16

METHODS OF TREESET :

1. **boolean add(Object o):** This methods adds element in the object, which will automatically stores all the elements in increasing order.
2. **Object ceiling(Object o):**This method retrieves the least element which is greater than or equal to the given element in the argument list, or null if there is no such element in the set.
3. **void clear():**This method removes all of the elements from this object.
4. **Object first():**This method returns the lowest element in the Set.
5. **Object floor(Object o):** This method returns the greatest element which is less than or equal to the given element in the argument list, or null if there is no such element.
6. **Object higher(Object o):**This method returns the least element in the TreeSet which is strictly greater than the given element in the argument of this method, or it will return null if there is no such element.
7. **boolean isEmpty():**This method returns true if the set is empty or false if set contains any element.
8. **Iterator iterator():**This method gives an object of Iterator class by which we can retrieve list in increasing order.
9. **Object last():** This method returns the highest element present in the treeset.
10. **Object lower(Object o):**This method returns the greatest element in this treeset which is strictly less than the given element in the set, or it will return null if there is no such element.
11. **Object pollFirst():** This method returns as well as removes the lowest element, or it will returns null if this tree set is empty.
12. **Object pollLast():**This method retrieves as well as removes the highest element, or it will returns null if this set is empty.
13. **boolean remove(Object o):**This method removes the element specified from the treeset, if it is present.
14. **Integer size():**This method gives the size of treeset.

Program :

```
import java.util.*;

class TreSetMethodsDemo
{
        public static void main(String args[])
        {
                TreeSet <Integer>ts1 = new TreeSet<Integer>();
                ts1.add(3);

                ts1.add(5);

                ts1.add(5);

                ts1.add(1);

                ts1.add(2);

                ts1.add(9);

                ts1.add(11);

                ts1.add(4);

                System.out.println("Values in TreeSet object 1: " +ts1);

                System.out.println("Ceiling value of '6' is:" +ts1.ceiling(6));

                System.out.println("Ceiling value of '11' is:" +ts1.ceiling(10));

                System.out.println("Tree Set Object contains element '0' : "+ts1.contains(0));

                System.out.println("Values in TreeSet object 1 are :");

                Iterator itr = ts1.Iterator();

                while (itr.hasNext())
                {

                        System.out.println(itr.next() + " ");

                }
                System.out.println("Lowest element in the tree set is : " +ts1.first());

                System.out.println("Floor value of '6' : " +ts1.floor(6));

                System.out.println("Floor value of '10' : " +ts1.floor(10));

                System.out.println("highest value than 4 :  " + ts1.higher(4) );

                System.out.println("highest value than 5 :  " + ts1.higher(5) );

                System.out.println("Is Tree Set empty ?: " +ts1.isEmpty());

                System.out.println("Last elememt in Tree Set object is: "+ts1.last());

                System.out.println("Greatest Element Less than 5: "+ts1.lower(5));
```

```
                System.out.println("Greatest Element Less than 5: "+ts1.lower(1));
                System.out.println("Removed Element using Pollfirst method :"+ts1.pollFirst());
                System.out.println("Values in Tree Set object: " + ts1);
                System.out.println("Removed Element using PollLast method "+ts1.pollLast());
                System.out.println("Values in Tree Set object: " + ts1);
                System.out.println("Is element '1' removed ? " +ts1.remove(5));
                System.out.println("Values in Tree Set Object: " +ts1);
                System.out.println("Size of the Tree Set Object : " +ts1.size());
        }
}
```

RETRIEVING ELEMENTS FROM COLLECTIONS

If we want to retrieve objects one by one from the collection, then we should go for cursors.

There are 3 types of cursors are available in java

1. Iterator Interface
2. ListIterator Interface
3. Enumeration Interface

1. **Iterator Interface:** Iterator is an Interface that contains methods to retrieve the elements one by one from a collection object. Iterator Interface we can perform both read and remove operations.

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

Syntax : Iterator iterator();

Example : Iterator e = object. iterator ();

Methods of Iterator Interface

1. **boolean hasNext() :** This method returns true if the iterator has more elements.
2. **element next() :** This method returns the next element in the iterator.
3. **void remove() :** This method removes from the collection the last element returned by the iterator.

Program :

```
import java.util.*;
class ArrayListDemo
{
        public static void main(String arg[])
        {
                ArrayList<Integer> ar1=new ArrayList<Integer>();
```

```
                ar1.add(20);
                ar1.add(30);
                ar1.add(40);
                ar1.add(20);
                Iterator it=ar1.iterator();
                System.out.println("Elements using Iterator :");
                while(it.hasNext())
                {
                        System.out.println(it.next());
                }
                it.remove();
                System.out.println("After the remove() method is called :"+ar1);  //[20,30,40]
        }
}
```

2. **ListIterator Interface:** ListIterator is an Interface that contains methods to retrieve the elements from a collection object, both in forward and reverse directions.

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

Syntax : ListIterator listIterator();

Example : ListIterator li = object. listIterator ();

Methods of ListIterator Interface

a) **Forward Direction**

1. **boolean hasNext() :** This method returns true if the ListIterator has more elements when traversing the list in the forward direction.
2. **element next() :** This method returns the next element in the list.
3. **int nextIndex():**This method returns the index of the element that would be returned on calling the *next()* method.

b) **Backward Direction**

1. **boolean hasPrevious() :** This method returns true if the ListIterator has more elements when traversing the list in the reverse direction.
2. **element previous() :** This method returns the previous element in the list.
3. **int previousIndex():** This method returns the index of the element that would be returned on calling the *previous()* method.

c) **Other Method**

1. **void remove() :** This method removes from the list the last element that was returned by the next() or previous() method.

Program:

```
import java.util.*;
class ListIterator1
{
	public static void main(String arg[])
	{
		Vector<Integer> v1=new Vector<Integer>();
		v1.add(20);
		v1.add(30);
		v1.add(40);
		v1.add(50);
		v1.add(60);
		System.out.println("Vector List is :"+v1);
		System.out.println("Elements using ListIterator :");
		ListIterator lit=v1.listIterator ();
		System.out.println("In Forward Direction :");
		while(lit.hasNext())
		{
			System.out.println("Position of Next Element: "+lit.nextIndex());
			System.out.println(lit.next());
		}
		lit.remove();
		System.out.println("After removing method Vector List is :"+v1);
		System.out.println("In Backward Direction :");
		while(lit.hasPrevious())
		{
			System.out.println("Position of Previous Element: "+lit.previousIndex());
			System.out.println(lit.previous());
		}
		lit.remove();
		System.out.println("After removing method Vector List is :"+v1);
	}
}
```

3. **Enumeration Interface:** Enumeration is an Interface is useful to retrieve the elements one by one from a collection object. It is similar to Iterator Interface. The enumeration interface retrieves elements in a *forwarding direction*.

We can create Enumeration object by using elements() method .

Syntax : Enumeration elements();

Example : Enumeration e = object.elements();

Methods of Enumeration Interface

1. **boolean hasMoreElements() :** This method returns true if the Enumeration has any more elements.
2. **element nextElement() :** This method returns the next element is available in Enumeration.

Program :

```
import java.util.*;
class EnumerationDemo
{
        public static void main(String arg[])
        {
                Vector<Integer> v1=new Vector<Integer>();
                v1.add(20);
                v1.add(30);
                v1.add(40);
                v1.add(20);
                System.out.println("Vector List is :"+v1);
                Enumeration e=v1.elements();
                System.out.println("Elements using Enumeration :");
                while (e.hasMoreElements())
                {
                        System.out.println(e.nextElement());
                }
        }
}
```

Map Interface

1. Map interface in Java is defined in java.util.Map package.
2. It is a part of the Java collections framework but it does not extend the collection interface.
3. A map contains values on the basis of key, i.e. key and value pair.
4. Each key and value pair is known as an entry.
5. A Map contains unique keys.
6. A Map is useful if you have to search, update or delete elements on the basis of a key.

Map interface is implemented by the classes

1.HashMap

2. LinkedHashMap

3.TreeMap

HASH MAP CLASS :

1. **HashMap** is an unordered collection that stores elements (objects) in the form of key-value pairs (called entries).
2. Java HashMap contains only unique keys that means no duplicate keys are allowed but values can be duplicated. We retrieve values based on the key.
3. If we try to insert an entry that has a duplicate key, the map replaces the old entry with a new entry.
4. In HashMap, insertion order is not preserved (i.e. maintains no order). Which means we cannot retrieve keys and values in the same order in which they have been inserted into the HashMap. It is based on the Hashcode of keys, not on the hash code of values.
5. Heterogeneous objects are allowed for both keys and values.
6. HashMap may have one null key and multiple null values.
7. HashMap in Java is not synchronized that means while using multiple threads on the HashMap object, we will get unreliable results.

Constructors of HashMap class

HashMap(): It is used to construct an empty HashMap object with the default initial capacity of 16 and the default fill ratio (load factor) is 0.75. The syntax to create a hash map object is as follows:

Sytax : HashMap hmap = new HashMap();

(or)

HashMap<K, V> hmap = new HashMap<K,V>(); // Generic form.

HashMap Methods in Java

1. **put(Object key, Object value):** This method is used To insert the elements or an entry in a Hashmap.
2. **putIfAbsent(K key, V value):** This method adds the specified value associated with the specified key in the map only if it is not already specified.
3. **get(Object key):** This method is used to retrieve the value associated with a key. Its return type is Object.
4. **Collection values():** This method returns a collection view containing all of the values in the map.
5. **remove(Object key):** This method is used to delete an entry for the specified key.
6. **boolean remove(Object key, Object value):** This method removes the specified value associated with specific key from the map.
7. **replace(K key, V value):** This method is used to replace the specified value for a specified key.
8. **int size():** This method returns the number of entries in the map.
9. **boolean isEmpty():** This method is used to check whether the map is empty or not. If there are no key-value mapping present in the hash map then it returns true, else false.
10. **void clear():** It is used to remove entries from the specified map.
11. **Set entrySet():** It is used to return a set view containing all of the key/value pairs in this map.
12. **Set keySet():** This method is used to retrieve a set view of the keys in this map.
13. **boolean containsKey(Object key) :** This method tests whether the specified key k is available in the hashmap.
14. **boolean containsValue(Object key) :** This method tests whether the specified value is available in the hashmap.

```
import java.util.*;
class HashMapDemo
{
        public static void main(String[] args)
        {
                HashMap<String, Integer> hp = new HashMap<>();
                System.out.println("Is HashMap empty: " +hp.isEmpty());
                        hp.put("One", 1);
                        hp.put("Two", 2);
```

```
                hp.put("Three", 3);
                hp.put("Three",20);
                hp.put("null", null);
                hp.put("Four",3);
            System.out.println("HashMap Numbers: " + hp);
            HashMap<String, Integer> hp1 = new HashMap<>();
                hp1.put("Five", 1);
                hp1.put("Six", 22);
            System.out.println("HashMap Numbers1: " + hp1);
                hp.putIfAbsent("Four",4);
                hp.putIfAbsent("Five",5);
                hp.putIfAbsent("Seven",7);
            System.out.println("Updated HashMap Numbers: " + hp);
            System.out.println("Size of HashMap: " + hp.size());
            System.out.println("Three value: "+hp.get("Three"));
            System.out.println("Hash map Values:"+hp.values());
            System.out.println("Removed Entry: " +hp.remove("null"));
                hp.remove("Six",22);
            System.out.println("HashMap Entries after remove: " + hp);
            System.out.println("Replaced value: "+hp.replace("Three",3));
            System.out.println("Replaced value: "+hp.replace("Four",3,4));
            System.out.println("Updated entries in HashMap: "+hp);
            System.out.println(hp.get("Three"));
            System.out.println("Set View: " + hp.entrySet());
            System.out.println("Keys: " + hp.keySet());
            System.out.println(hp.containsKey("Six"));
            System.out.println(hp.containsValue(6));
        }
}
```

LINKEDHASHMAP CLASS :

1. LinkedHashMap class extends the HashMap class and implements Map interface.
2. LinkedHashMap contains unique elements. It contains values based on keys.
3. LinkedHashMap allows only one null key but can have multiple null values.
4. LinkedHashMap in Java is non synchronized. That is, multiple threads can access the same LinkedHashMap object simultaneously.
5. The default initial capacity of LinkedHashMap class is 16 with a load factor of 0.75.

Constructors of LinkedHashMap class

LinkedHashMap(): This constructor is used to create a default LinkedHashMap object. It constructs an empty insertion-ordered LinkedHashMap object with the default initial capacity 16 and load factor 0.75.

Sytax : LinkedHashMap hmap = new LinkedHashMap();

(or)

LinkedHashMap <K, V> hmap = new LinkedHashMap <K,V>(); // Generic form.

LinkedHashMap Methods in Java

1. **put(Object key, Object value):** This method is used To insert the elements or an entry in a Hashmap.
2. **putIfAbsent(K key, V value):** This method adds the specified value associated with the specified key in the map only if it is not already specified.
3. **get(Object key):** This method is used to retrieve the value associated with a key. Its return type is Object.
4. **Collection values():** This method returns a collection view containing all of the values in the map.
5. **remove(Object key):** This method is used to delete an entry for the specified key.
6. **boolean remove(Object key, Object value):** This method removes the specified value associated with specific key from the map.
7. **replace(K key, V value):** This method is used to replace the specified value for a specified key.
8. **int size():** This method returns the number of entries in the map.
9. **boolean isEmpty():** This method is used to check whether the map is empty or not. If there are no key-value mapping present in the hash map then it returns true, else false.
10. **void clear():** It is used to remove entries from the specified map.
11. **Set entrySet():** It is used to return a set view containing all of the key/value pairs in this map.
12. **Set keySet():** This method is used to retrieve a set view of the keys in this map.

13. **boolean containsKey(Object key) :** This method tests whether the specified key k is available in the Linkedhashmap.
14. **boolean containsValue(Object key) :** This method tests whether the specified value is available in the Linkedhashmap.

Program :

```
import java.util.*;
class LinkedHashMapDemo
{
        public static void main(String[] args)
        {
                LinkedHashMap<String, Integer> lhp = new LinkedHashMap<>();
                System.out.println("Is LinkedHashMap empty: " +lhp.isEmpty());
                        lhp.put("One", 1);
                        lhp.put("Two", 2);
                        lhp.put("Three", 3);
                        lhp.put("Three",20);
                        lhp.put("null", null);
                        lhp.put("Four",3);
                System.out.println("LinkedHashMap Numbers: " + lhp);
                LinkedHashMap<String, Integer> lhp1 = new LinkedHashMap<>();
                        lhp1.put("Five", 1);
                        lhp1.put("Six", 22);
                System.out.println("LinkedHashMap Numbers1: " + lhp1);
                System.out.println("Updated LinkedHashMap Numbers: " + lhp);
                        lhp.putIfAbsent("Four",4);
                        lhp.putIfAbsent("Five",5);
                        lhp.putIfAbsent("Two",2);
                System.out.println("Updated HashMap Numbers: " + lhp);
                System.out.println("Size of LinkedHashMap: " + lhp.size());
                System.out.println("Three value: "+lhp.get("Three"));
                System.out.println("LinkedHash map Values:"+lhp.values());
                System.out.println("Removed Entry: " +lhp.remove("null"));
                        lhp.remove("Six",22);
```

```
                System.out.println("LinkedHash map Entries after remove: " + lhp);
                System.out.println("Replaced value: "+lhp.replace("Three",3));
                System.out.println("Replaced value: "+lhp.replace("Four",3,4));
                System.out.println("Updated entries in LinkedHash map: "+lhp);
                System.out.println(lhp.get("Three"));
                System.out.println("Set View: " + lhp.entrySet());
                System.out.println("Keys: " + lhp.keySet());
                System.out.println(lhp.containsKey("Six"));
                System.out.println(lhp.containsValue(6));
    }
}
```

TREE MAP CLASS

1. The underlying data structure of Java TreeMap is a red-black binary search tree.
2. TreeMap contains only unique elements.
3. TreeMap cannot have a null key but can have multiple null values.
4. TreeMap is non synchronized. That means it is not thread-safe.
5. TreeMap in Java maintains ascending order.
6. Java TreeMap determines the order of entries by using either Comparable interface or Comparator class.
7. The iterator returned by TreeMap is fail-fast. That means we cannot modify map during iteration.

Constructors of TreeMap class

TreeMap(): This constructor is used to create a default TreeMap object. It constructs an empty insertion-ordered TreeMap object with the default initial capacity 16 and load factor 0.75.

Sytax : TreeMap hmap = new TreeMap();

(or)

TreeMap <K, V> hmap = new TreeMap <K,V>(); // Generic form.

TreeMap Methods in Java

1. **put(Object key, Object value):** This method is used To insert the elements or an entry in a Hashmap.
2. **putIfAbsent(K key, V value):** This method adds the specified value associated with the specified key in the map only if it is not already specified.

3. **get(Object key):** This method is used to retrieve the value associated with a key. Its return type is Object.
4. **Collection values():** This method returns a collection view containing all of the values in the map.
5. **remove(Object key):** This method is used to delete an entry for the specified key.
6. **boolean remove(Object key, Object value):** This method removes the specified value associated with specific key from the map.
7. **replace(K key, V value):** This method is used to replace the specified value for a specified key.
8. **int size():** This method returns the number of entries in the map.
9. **boolean isEmpty():** This method is used to check whether the map is empty or not. If there are no key-value mapping present in the hash map then it returns true, else false.
10. **void clear():** It is used to remove entries from the specified map.
11. **Set entrySet():** It is used to return a set view containing all of the key/value pairs in this map.
12. **Set keySet():** This method is used to retrieve a set view of the keys in this map.
13. **boolean containsKey(Object key) :** This method tests whether the specified key k is available in the Treemap.
14. **boolean containsValue(Object key) :** This method tests whether the specified value is available in the Treemap.
15. **firstKey():** It is used to retrieve key of first entry in the sorted order from the map. If the tree map is empty, it will throw NoSuchElementException.
16. **lastKey():** It is used to retrieve key of last entry in the sorted order from the map. If the tree map is empty, it will throw NoSuchElementException.
17. **Map.Entry<K,V> pollFirstEntry():** It removes and returns a key-value pair corresponding to the least key in this map, or null if the map is empty.
18. **Map.Entry<K,V> pollLastEntry():** It removes and returns a key-value pair corresponding to the greatest key in this map, or null if the map is empty.

Program:

```
import java.util.*;
class Main
{
        public static void main(String[] args)
        {
                TreeMap<String, Integer> tm = new TreeMap<>();
                System.out.println("Is TreeMap empty: " +tm.isEmpty());
```

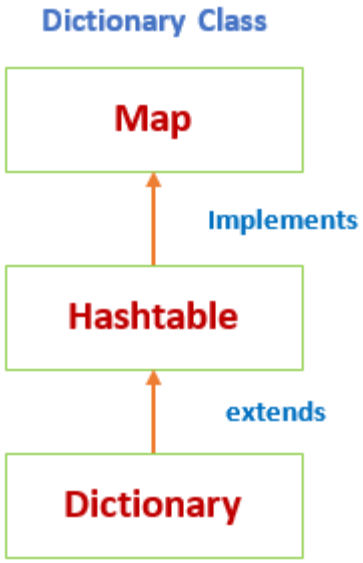
```
        tm.put("One", 1);
        tm.put("Two", 2);
        tm.put("Three", 3);
        tm.put("Three",20);
        tm.put("Null", null);
        tm.put("Four",3);
        System.out.println("TreeMap Numbers: " + tm);
        tm.putIfAbsent("Four",4);
        tm.putIfAbsent("Five",5);
        tm.putIfAbsent("Two",2);
        System.out.println("Updated TreeMap Numbers: " + tm);
        System.out.println("Size of TreeMap: " + tm.size());
        System.out.println("Three value: "+tm.get("Three"));
        System.out.println("TreeMap Values:"+tm.values());
        System.out.println("Removed Entry: " +tm.remove("null"));
            tm.remove("Six",22);
        System.out.println("TreeMap Entries after remove: " + tm);
        System.out.println("Replaced value: "+tm.replace("Three",3));
        System.out.println("Updated entries in TreeMap: "+tm);
        System.out.println(tm.get("Three"));
        System.out.println("Set View: " + tm.entrySet());
        System.out.println("Keys: " + tm.keySet());
        System.out.println(tm.containsKey("Six"));
        System.out.println(tm.containsValue(6));
        System.out.println("firstEntry: " +tm.firstEntry());
        System.out.println("lastEntry: " +tm.lastEntry());
        System.out.println("pollFirstEntry: " +tm.pollFirstEntry());
        System.out.println("pollLastEntry: " +tm.pollLastEntry());
        System.out.println("Updated entries in TreeMap: "+tm);
    }
}
```

LEGACY CLASSES

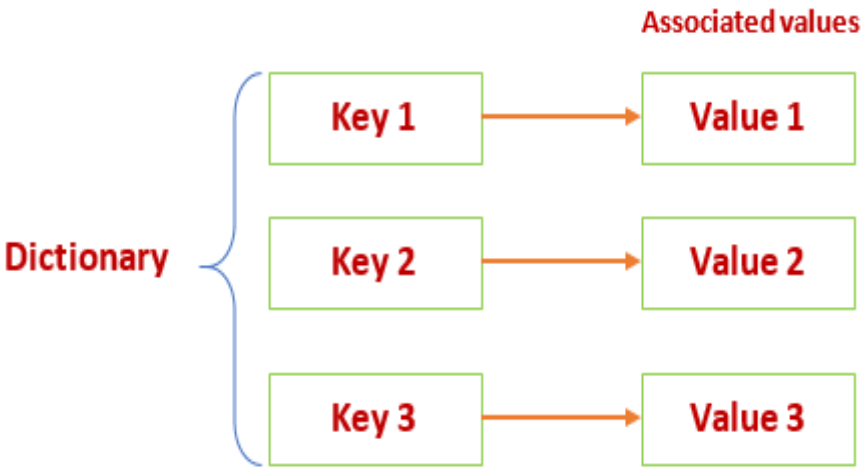
1.Dictionary Class

1. Dictionary Class is an abstract class that stores *key-value* pairs.
2. Given a key, its corresponding value can be stored and retrieved as needed.
3. A dictionary is a list of key-value pairs.
4. As Dictionary is an abstract class we cannot create its object. It needs a child class like Hash table.

The Dictionary class hierarchy is as follows:



Every key is associated with at most one value, as shown in the following figure.



Once the value is stored in a dictionary object, we can retrieve it by using the key.

Dictionary class Methods

1. **Object put(Object key, Object value) :** Inserts a key and its value into the dictionary. Returns null on success; returns the previous value associated with the key if the key is already exist.
2. **int size( ) :** It returns the total number of elements in the dictionary.
3. **Object get(Object key) :** It returns the value associated with given key; Returns null if the key does not exist.
4. **boolean isEmpty( ) :** It returns true if dictionary has no elements otherwise returns false.
5. **Object remove (Object key) :** It returns the value associated with given key and removes the same; Returns null if the key does not exist.
6. **Enumeration keys ( ):** Returns an enumeration of the keys contained in the dictionary.
7. **Enumeration elements ( ):** Returns an enumeration of the elements contained in the dictionary.

Program:

```
import java.util.*;
class DictionaryDemo
{
  public static void main(String args[])
  {
        Dictionary d=new Hashtable ();
        d.put(101,"Apple");
        d.put(102,"Mango");
        d.put(103,"Lemon");
        d.put(104,"Orange");
        d.put(105,"Banana");
        d.put(106,"Graps");
        d.put(107,"Cat");
        d.put(108,"Dog");
        d.put(109,"Elephant");
        d.put(1010,"Monkey");
        System.out.println("Dictionary values are:"+d);
        System.out.println("Size of the Dictionary:"+d.size());
        System.out.println("\nValue at key = 105 :"+d.get(105));
        System.out.println("\nIs the dictionary empty? : "+d.isEmpty());
        System.out.println("The removed value is: "+d.remove(106));
        System.out.println("\nDictionary keys are: \n");
        Enumeration e = d.keys();
```

```
        while(e.hasMoreElements())
        {
                System.out.println(e.nextElement());
        }
        System.out.println("\nDictionary Values are: \n");
        Enumeration e1 = d.elements();
        while(e1.hasMoreElements())
        {
                System.out.println(e1.nextElement());
        }
    }
}
```

2.HASH TABLE :

1. The underlying data structure for Java Hashtable is a hash table only.
2. Insertion order is not preserved. That means it does not maintain insertion order.
3. Duplicate keys are not allowed but values can be duplicated.
4. Heterogeneous objects are allowed for both keys and values.
5. Null is not allowed for both key and values. If we attempt to store null key or value, we will get a RuntimeException named NullPointerException.
6. Java Hashtable implements Serializable and Cloneable interfaces but not random access.
7. Every method present in Hashtable is synchronized. Hence, Hashtable object is thread-safe.
8. Since Hashtable is synchronized, its operations are slower as compared to HashMap in java.

Hierarchy of Java Hashtable

Hashtable class extends Dictionary class and implements Map, Cloneable, and Serializable interfaces.

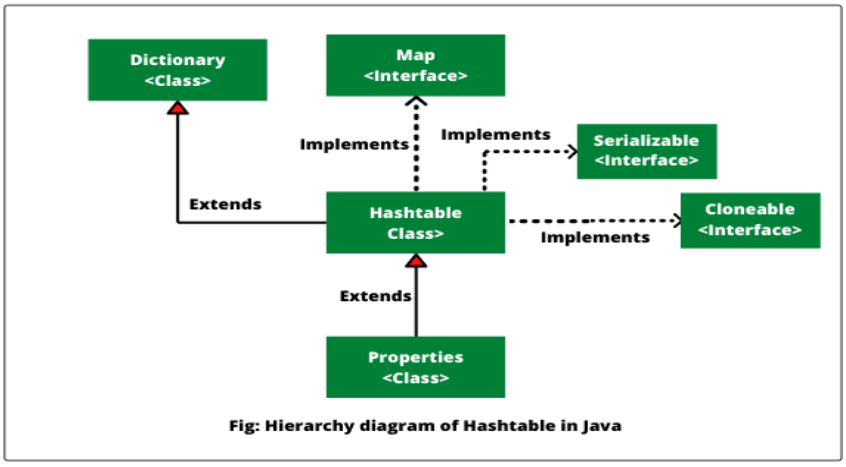


Fig: Hierarchy diagram of Hashtable in Java

Constructors of Hashtable class

Hashtable(): This form of constructor constructs a new, empty hashtable object with a default initial capacity as 11 and load factor as 0.75.

To store a string as key and an integer object as its value, we can create a Hashtable object as:

Hashtable<String,Integer> ht = new Hashtable<>();

HashTable Methods in Java

1. **put(Object key, Object value):** This method is used To insert the elements or an entry in a Hashmap.
2. **putIfAbsent(K key, V value):** This method adds the specified value associated with the specified key in the map only if it is not already specified.
3. **get(Object key):** This method is used to retrieve the value associated with a key. Its return type is Object.
4. **Collection values():** This method returns a collection view containing all of the values in the map.
5. **remove(Object key):** This method is used to delete an entry for the specified key.
6. **boolean remove(Object key, Object value):** This method removes the specified value associated with specific key from the map.
7. **int size():** This method returns the number of entries in the map.
8. **boolean isEmpty():** This method is used to check whether the map is empty or not. If there are no key-value mapping present in the hash map then it returns true, else false.
9. **void clear():** It is used to remove entries from the specified map.
10. **Set entrySet():** It is used to return a set view containing all of the key/value pairs in this map.
11. **Set keySet():** This method is used to retrieve a set view of the keys in this map.
12. **firstKey():** It is used to retrieve key of first entry in the sorted order from the map. If the tree map is empty, it will throw NoSuchElementException.
13. **lastKey():** It is used to retrieve key of last entry in the sorted order from the map. If the tree map is empty, it will throw NoSuchElementException.
14. **Enumeration<V> elements():** This method returns an enumeration of the values in this hashtable.
15. **Enumeration<K> keys():** This method returns an enumeration of the keys in this hashtable.

Program :

```
import java.util.*;
class HashtableDemo
{
	public static void main(String[] args)
	{
		Hashtable<Integer, String> ht = new Hashtable<Integer, String>();
		System.out.println("Is hash table empty: " + ht.isEmpty());
			ht.put(1, "One");
			ht.put(2, "Two");
			ht.put(3, "Three");
			ht.put(4, "Four");
			ht.put(5, "Five");
			ht.put(6, "Six");
		System.out.println("Displaying entries in hash table: " +ht);
		System.out.println("Size of hash table: " +ht.size());
		System.out.println("Removed entry: " +ht.remove(6));
		System.out.println("Updated entries in hash table: " +ht);
		System.out.println("Getting the value of 4: " +ht.get(4));
		System.out.println("Getting the value of 2: " +ht.get(2));
		System.out.println("Iterating keys of hash table:");
		Iterator<Integer> itr = ht.keySet().iterator();
		while(itr.hasNext())
		{
			System.out.println(itr.next());
		}
		System.out.println("Iterating values of hash table:");
		Iterator<String> itrValue = ht.values().iterator();
		while(itrValue.hasNext())
		{
			System.out.println(itrValue.next());
		}
	}
}
```

3.PROPERTIES CLASS

1. **Properties in Java** is a class that is a child class (subclass) of Hashtable.
2. It is mainly used to maintain the list of values in which key and value pairs are represented in the form of strings.
3. In other words, keys and values in Properties objects should be of type String.
4. Properties class extends Hashtable class that extends Dictionary class. It implements Map, Serializable, and Cloneable interfaces.

Constructors of Properties class

properties(): This form of a constructor is used to create an empty property list (map) with no default values.

The general syntax to create Properties object is as follows:

Properties p = new Properties();

Methods of Properties class

1. String getProperties(String key): This method returns the value associated with key. If the key is neither in the list nor in the default property list, it will return null object.

2. String getProperty(String key, String defaultValue): This method returns the value associated with key. If the key is neither in the list nor in the default property list, it will return defaultValue.

3. void list(PrintStream streamOut): This method prints the property list streamOut to the specified output stream.

4. void list(PrintWriter streamOut): This method prints this property list streamOut to the specified output stream.

5. void load(InputStream inStream): This method loads (reads) a property list (key and element pairs) from the InputStream.

6. void load(Reader reader): This method loads a property list (key and element pairs) from the input character stream in a simple line-oriented format.

7. void loadFromXML(InputStream in): This method loads all of the properties represented by the XML document on the specified InputStream into the properties table.

8. Enumeration propertyNames(): It returns an enumeration of all the keys in this property list, including distinct keys in the default property list.

9. Object setProperties(String key, String value): It returns value associated with key. It returns null if value associated with key does not exist.

Program :

db.properties

1. **user=system**
2. **password=oracle**

```
import java.util.*;
import java.io.*;
public class Test
{
	public static void main(String[] args)throws Exception
	{
		FileReader reader=new FileReader("db.properties");
		Properties p=new Properties();
			p.load(reader);
		System.out.println(p.getProperty("user"));
		System.out.println(p.getProperty("password"));
	}
}
```

OUTPUT:

```
system
oracle
```

More Utility classes

String Tokenizer:

1. The StringTokenizer class is available inside the java.util package.
2. StringTokenizer class is useful to break a string into pieces called 'Tokens'.
3. These Tokens are then stored in the StringTokenizer object from where they can be retrived.

Constructors of StringTokenizer :

1. StringTokenizer(String str): It creates StringTokenizer with specified string.

2. StringTokenizer(String str, String delim): It creates StringTokenizer with specified string and delimiter.

3. StringTokenizer(String str, String delim, boolean flag): It creates StringTokenizer with specified string, delimiter and return Value. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods of the StringTokenizer Class

1. **int countTokens() :** This method counts and returns the number of tokens available in a StringTokenizer object.
2. **boolean hasMoreTokens () :** This method tests if there are more tokens available in the StringTokenizer object or not. If next token is there then it returns true.
3. **String nextToken ():** This method returns the next token from the StringTokenizer.

Program:

```
import java.util.*;
class StringTokenizerDemo
{
        public static void main(String args[])
        {
                System.out.println("StringTokenizer Constructor 1 - ");
                StringTokenizer st1 = new StringTokenizer("Hello Geeks How are you", " ");
                while (st1.hasMoreTokens())
                {
                        System.out.println(st1.nextToken());
                }
                System.out.println("StringTokenizer Constructor 2 - ");
                StringTokenizer st2 = new StringTokenizer("JAVA : Code : String", " :");
                while (st2.hasMoreTokens())
                {
                        System.out.println(st2.nextToken());
                }
        }
}
```

DATE CLASS :

1. The class Date represents a specific instant in time, with millisecond precision.
2. The Date class of java.util package implements Serializable, Cloneable and Comparable interface.
3. It provides constructors and methods to deal with date and time with java.

Methods of Date Class :

1.int getMinutes() : This method returns the minutes part of the current date.

2.int getHours() : This method returns the hours part of the current date.

3.int getDate() : This method returns the date part of the current date.

4.int getYear() : This method returns the year part of the current date. (It gives Currentyear-1900)

5.int getSeconds() : This method returns the seconds part of the current date.

6.long getTime() : Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.

Program

```
import java.util.*;
class DateDemo
{
        public static void main(String[] args)
        {
                Date d1 = new Date();
                System.out.println("Current date is " + d1);
                Date d2 = new Date(2323223232L);
                System.out.println("Date represented is "+ d2 );
                System.out.println(d1.getHours());
                System.out.println(d1.getMinutes());
                System.out.println(d1.getSeconds());
                System.out.println(d1.getMonth());
                System.out.println(d1.getDate());
                System.out.println(d1.getDay());
                System.out.println(d1.getYear());
                System.out.println(d1.getTime());
        }
}
```

CALENDAR CLASS

1. **Calendar class in Java** is an abstract super class that is designed to support different kinds of calendar systems. It is used to know system data and time information.
2. Java Calendar class provides some specific functionality that is common to most calendar systems and makes that functionality available to the subclasses.
3. The subclasses of Calendar class are used to calculate calendar related information (such as second, minute, hour, day of month, month, year, and so on) of a particular calendar system.

How to create a Calendar in Java?

To create an object of Calendar class in java, we need to call getInstance() method. Since this method is static in nature so we call it using Calendar class name like this:

Syntax : Calendar cl = Calendar.getInstance();

This Calendar object stores the current system date and time by default.

Methods of Calendar class

1. abstract void add(int field, int amount): This method is used to add or subtract the specified amount of time to the given calendar field, based on the calendar's rules.

2. boolean after(Object calendarObj): This method returns true if the invoking calendar object contains a date that is after (later) than specified calendarObj. Otherwise, it returns false.

3. boolean before(Object calendarObj): This method returns true if the invoking calendar object contains a date that is before (earlier) than specified calendarObj. Otherwise, it returns false.

4. final void clear(): It sets zeros all time components in the invoking calendar object.

5. final void clear(int field): It sets zeros the specified time components in the invoking calendar object.

6. int get(int calendarField): It returns the value of the given calendar field.

7. final Date getTime(): It returns a Date object that is equal to the time of invoking object.

Calendar Field Constants

Calendar class defines the following int constants that are used to get or set components of the calendar. They are listed in the table.

ALL_STYLES	FRIDAY	PM	AM
HOUR	SATURDAY	AM_PM	HOUR_OF_DAY
SECOND	APRIL	JANUARY	SEPTEMBER
AUGUST	JULY	SHORT	DATE
JUNE	SUNDAY	DAY_OF_MONTH	LONG

THURSDAY	DAY_OF_WEEK	MARCH	TUESDAY
DAY_OF_WEEK_IN_MONTH	MAY	UNDECIMBER	DAY_OF_YEAR
MILLISECOND	WEDNESDAY	DECEMBER	MINUTE
WEEK_OF_MONTH	DST_OFFSET	MONDAY	WEEK_OF_YEAR
ERA	MONTH	YEAR	FEBRUARY
NOVEMBER	ZONE_OFFSET	FIELD_COUNT	OCTOBER

```
import java.util.*;
class CalendarDemo
{
	public static void main(String[] args)
	{
		Calendar cl = Calendar.getInstance();
		System.out.print("Current System Date: ");
		int dd = cl.get(Calendar.DATE);
		 int mm = cl.get(Calendar.MONTH);
		int yy = cl.get(Calendar.YEAR);
		System.out.println(dd+ "-" +mm+ "-" +yy);
		System.out.print("Current System Time: ");
		int hr = cl.get(Calendar.HOUR);
		int min = cl.get(Calendar.MINUTE);
		 int sec = cl.get(Calendar.SECOND);
		int secMilli = cl.get(Calendar.MILLISECOND);
		System.out.println(hr+ ":"+min+ ":"+ sec+ ":"+secMilli);
		System.out.println("At present Date and Time: " +cl.getTime());
		int maximum = cl.getMaximum(Calendar.DAY_OF_WEEK);
		System.out.println("Maximum number of days in week: " + maximum);
		maximum = cl.getMaximum(Calendar.WEEK_OF_YEAR);
		System.out.println("Maximum number of weeks in year: " + maximum);
		 int minimum = cl.getMinimum(Calendar.DAY_OF_WEEK);
		System.out.println("Minimum number of days in week: " + minimum);
		minimum = cl.getMinimum(Calendar.WEEK_OF_YEAR);
		System.out.println("Minimum number of weeks in year: " + minimum);
		cl.add(Calendar.DATE, -20);
		System.out.println("20 Days ago, date and time information: " + cl.getTime());
		cl.add(Calendar.MONTH, 5);
		System.out.println("5 months later, date and time information: " + cl.getTime());
		cl.add(Calendar.YEAR, 5);
		System.out.println("5 years later, date and time information: " + cl.getTime());
	}
}
```

SCANNER CLASS :

1. The Scanner class implements Iterator interface.
2. The Scanner class provides the easiest way to read input in a Java program.
3. The System.in parameter is used to take input from the standard input. It works just like taking inputs from the keyboard.
4. We have then used the nextLine() method of the Scanner class to read a line of text from the user.

Methods of Scanner Class :

1 : nextBoolean() : This method is used to Reads a boolean value from the user
2 : nextByte() : This method is used to Reads a byte value from the user
3 : nextDouble() : This method is used to Reads a double value from the user
4 : nextFloat() : This method is used to Reads a float value from the user
5 : nextInt() : This method is used to Reads a int value from the user
6 : nextLine() : This method is used to Reads a String value from the user
7 : nextLong() : This method is used to Reads a long value from the user
8 : nextShort() : This method is used to Reads a short value from the user
9 : next() : This method is used to reads a word from the user
10: close() : This method is used to is used to closes this scanner.

Program :

```
import java.util.*;
class ScannerExample
{
        public static void main(String args[])
        {
                Scanner in = new Scanner(System.in);
                System.out.print("Enter your name: ");
                String name = in.nextLine();
                System.out.println("Name is: " + name);
                System.out.print("Enter your age: ");
                int i = in.nextInt();
                System.out.println("Age: " + i);
                System.out.print("Enter your salary: ");
                double d = in.nextDouble();
                System.out.println("Salary: " + d);
                in.close();
        }
}
```

BITSET CLASS :

1. The BitSet is a built-in class in java used to create a dynamic array of bits represented by boolean values.
2. The BitSet class is available inside the java.util package.
3. The BitSet array can increase in size as needed. This feature makes the BitSet similar to a Vector of bits.
4. The bit values can be accessed by non-negative integers as an index.
5. The default value of the BitSet is boolean false with a representation as 0 (off).
6. BitSet uses 1 bit of memory per each boolean value.

Constructors of BitSet

1.BitSet() : A no-argument constructor to create an empty BitSet object.

Syntax : BitSet b=new BitSet();

2. **BitSet(int no_Of_Bits):** A one-constructor with an integer argument to create an instance of the BitSet class with an initial size of the integer argument representing the number of bits.

Syntax : BitSet b=new BitSet(16);

Methods of BitSet class

1. **void and(BitSet bitSet) :** It performs AND operation on the contents of the invoking BitSet object with those specified by bitSet.
2. **void andNot(BitSet bitSet) :** For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.
3. **void or(BitSet bitSet) :** It performs OR operation on the contents of the invoking BitSet object with those specified by bitSet.
4. **void xor(BitSet bitSet) :** It performs XOR operation on the contents of the invoking BitSet object with those specified by bitSet.
5. **int cardinality( ) :** It returns the number of bits set to true in the invoking BitSet.
6. **void clear( ) :** It sets all the bits to zeros of the invoking BitSet.
7. **void clear(int index) :** It set the bit specified by given index to zero.
8. **void clear(int startIndex, int endIndex) :** It sets all the bits from specified startIndex to endIndex to zero.
9. **boolean equals(Object bitSet) :** It retruns true if both invoking and argumented BitSets are equal, otherwise returns false.
10. **boolean get(int index) :** It retruns the present state of the bit at given index in the invoking BitSet.

11. **boolean intersects(BitSet bitSet) :** It returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1.
12. **boolean isEmpty( ) :** It returns true if all bits in the invoking object are zero, otherwise returns false.
13. **int size( ) :** It returns the total number of bits in the invoking BitSet.
14. **int length( ) :** It returns the total number of bits in the invoking BitSet.
15. **void set(int index)** : It sets the bit specified by index.

Program :

```
import java.util.*;
class BitSetClassExample
{
        public static void main(String[] args)
        {
                BitSet bSet1 = new BitSet();
                BitSet bSet2 = new BitSet(16);
                        bSet1.set(10);
                        bSet1.set(5);
                        bSet1.set(0);
                        bSet1.set(7);
                        bSet1.set(20);
                        bSet2.set(1);
                        bSet2.set(15);
                        bSet2.set(20);
                        bSet2.set(77);
                        bSet2.set(50);
                System.out.println("BitSet_1 => " + bSet1);
                System.out.println("BitSet_2 => " + bSet2);
                bSet1.and(bSet2);
                System.out.println("BitSet_1 after and with bSet_2 => " + bSet1);
                bSet1.andNot(bSet2);
                System.out.println("BitSet_1 after andNot with bSet_2 => " + bSet1);
                System.out.println("Length of the bSet_2 => " + bSet2.length());
                System.out.println("Size of the bSet_2 => " + bSet2.size());
                System.out.println("Bit at index 2 in bSet_2 => " + bSet2.get(2));
                bSet2.set(2);
                System.out.println("Bit at index 2 after set in bSet_2 => " + bSet2.get(2));
        }
}
```

FORMATTER CLASS IN JAVA

1. The **Formatter** is a built-in class in java used for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output in java programming.
2. The Formatter class is defined as final class inside the **java.util** package.
3. The Formatter class implements **Cloneable** and **Flushable** interface.

The Formatter class in java has the following methods.

1. **Formatter format(Locale l, String format, Object... args):** It writes a formatted string to the invoking object's destination using the specified locale, format string, and arguments.
2. **Formatter format(String format, Object... args):** It writes a formatted string to the invoking object's destination using the specified format string and arguments.
3. **void flush() :** It flushes the invoking formatter.
4. **Appendable out() :** It returns the destination for the output.
5. **Locale locale() :** It returns the locale set by the construction of the invoking formatter.
6. **String toString():**It converts the invoking object to string.
7. **IOException ioException() :** It returns the IOException last thrown by the invoking formatter's Appendable.
8. **void close() :** It closes the invoking formatter.

EXAMPLE

```
import java.util.*;

public class FormatterClassExample
{
	public static void main(String[] args)
	{
		Formatter formatter=new Formatter();
		formatter.format("%2$5s %1$5s %3$5s", "Smart", "BTech", "Class");
		System.out.println(formatter);
		formatter = new Formatter();
		formatter.format(Locale.FRANCE,"%.5f", -1325.789);
		System.out.println(formatter);
		String name = "Java";
		formatter = new Formatter();
		formatter.format(Locale.US,"Hello %s !", name);
		System.out.println("" + formatter + " " + formatter.locale());
		formatter = new Formatter();
		formatter.format("%.4f", 123.1234567);
		System.out.println("Decimal floating-point notation to 4 places: " + formatter);
		formatter = new Formatter();
		formatter.format("%010d", 88);
		System.out.println("value in 10 digits: " + formatter);
	}
}
```

RANDOM CLASS IN JAVA

1. The Random is a built-in class in java used to generate a stream of pseudo-random numbers in java programming. The Random class is available inside the java.util package.
2. The Random class implements Serializable, Cloneable and Comparable interface.
3. The Random class is a part of java.util package.
4. The Random class provides several methods to generate random numbers of type integer, double, long, float etc.
5. The Random class is thread-safe.
6. Random number generation algorithm works on the seed value. If not provided, seed value is created from system nano time.

The Random class in java has the following methods.

1. **int next(int bits) :** It generates the next pseudo-random number.
2. **Boolean nextBoolean():**It generates the next uniformly distributed pseudo-random boolean value.
3. **double nextDouble() :** It generates the next pseudo-random double number between 0.0 and 1.0.
4. **void nextBytes(byte[] bytes) :** It places the generated random bytes into an user-supplied byte array.
5. **float nextFloat():**It generates the next pseudo-random float number between 0.0 and 1.0.
6. **int nextInt():**It generates the next pseudo-random int number.
7. **int nextInt(int n) :** It generates the next pseudo-random integer value between zero and n.
8. **long nextLong():**It generates the next pseudo-random, uniformly distributed long value
9. **void setSeed(long seedValue):** It sets the seed of the random number generator using a single long seedValue.

EXAMPLE

```
import java.util.Random;
public classRandomClassExample
{
        publicstaticvoidmain(String[] args)
        {
                Random rand =new Random();
                System.out.println("Integer random number - "+ rand.nextInt());
                System.out.println("Integer random number from 0 to 100 - "+ rand.nextInt(100));
                System.out.println("Boolean random value - "+ rand.nextBoolean());
                System.out.println("Double random number - "+ rand.nextDouble());
                System.out.println("Float random number - "+ rand.nextFloat());
                System.out.println("Long random number - "+ rand.nextLong());
        }
}
```