UNIT 4

THE COLLECTIONS FRAMEWORK (java.util)

PART-A
SHORT QUESTIONS WITH SOLUTIONS

What is the benefit of Generics in collections framework?

Answer :

April/May-18(R16), Q1(g)

The Java 1.5 version introduced generics and all the collection interfaces with their implementations. The generics in collection framework permit the user to provide the object type that is available inside the collection. Thus, the compiler will throw a compile-time error when the programmer adds a different type of object. Hence, this ignores the ClassCastException while running a program. Generics do not require any INSTANCEOF and casting operators for conversion. So, the code will be cleaner and clean all the time.

Define collection interfaces.

Answer :

Model Paper-II, Q1(g)

The collection framework of Java is built on a standard set of interfaces called collection interfaces. They are basically defined for implementing collection mechanism. The various interface of collection framework are as follows,

1. Collection interface
2. List interface
3. Queue interface
4. Dequeue interface
5. Set interface
6. Sortedset interface.

What is a Collection Class? Give an example.

Answer :

Nov./Dec.-18(R16), Q1(h)

The subclasses of collection interface are called collection classes. Some of the collection classes provide full implementation to collection interfaces so they can be used as it is, and others are abstract classes. The collection classes defined by collections framework are,

1. AbstractCollection
2. AbstractList
3. AbstractSequentialList
4. LinkedList
5. ArrayList
6. AbstractSet
7. HashSet
8. LinkedHashSet
9. TreeSet
10. Priority Queue
11. Array Deque.

Example

```
import java.util.*,
class ColorsList
{
    public static void main(string args[ ])
    {
        ArrayList color = new ArrayList( );
        Collections.addALL(colors, "red", "yellow", "black");
        colors.foreach(system.out : : println);
    }
}
```

Output

red
yellow
black.

Q4. Write a short notes on Hashset class.

Answer :

The HashSet inherits AbstractSet class and Set interface. The HashSet class is used to store the collection created by it into the hash table that stores the unique key. These keys are converted into hash code by applying a hashing technique. This hash code is then used to retrieve the data associated with the key. The hash table does not store the elements in any sorted order. Hashing is useful to perform the operations such as add( ), remove( ), size( ) and contains( ).

Q5. What are the constructors used in TreeMap?

Answer :

The constructors which are in TreeMap are as follows,

1. **TreeMap( )**
   It creates an empty TreeMap sorted as per the key's natural order(ascending order).
2. **TreeMap(Map m1)**
   It creates a TreeMap object with the elements of another Map class as per key's natural order.
3. **TreeMap(Comparator comp)**
   It creates an empty TreeMap object with the sorting order of the Comparator specified by "comp".
4. **TreeMap(SortedMap m1)**
   It creates a TreeMap object with the same elements of the SortedMap specified by "m1" with the same sorting order of "m1".

This class does not define its own methods.

Q6. What is a Java Priority queue?

Answer :

Priority Queue (Model Paper-I, Q1(g) | Nov./Dec.-18(R16), Q1(g))

This class is derived from abstract queue class to implement priority based queues.

Syntax

class Priority Queue<E1>

Constructor

The following are the constructors defined by priority queue,

(i) Priority Queue( )
(ii) Priority Queue(int capacity)
(iii) Priority Queue(comparator<? super E1>compt)
(iv) Priority Queue(int capacity, comparator<?super E1>compt)
(v) Priority Queue(collection<?extends E1>co)
(vi) Priority Queue(priority queue<?extends E1>co)
(vii) Priority Queue(sortedset<?extends E1>co).

Answer :

In computer vocabulary, iterator means an object with which we can travel all through a range of elements. In Java, Iterator is an interface that allows programmers to access a collection.

Following is the signature of iterator interface,

```
public Interface Iterator
```

Q8. Differentiate between Enumeration and Iterator interface.

Answer : April/May-18(R16), Q1(h)

	Enumeration		Iterator
1.	It allows to iterate through a list of sequentially stored elements in a data structure.	1.	It allows programmers to access a collection. It replaces enumeration interface in collections framework.
2.	It consists of tow abstract methods such as hasMoreElements( ), nextElement( ).	2.	It consists of three abstract methods such as boolean hasNext( ), objectNext( ), void remove( ).
3.	It does not consist of remove( ) method.	3.	It consists of remove( ) method.
4.	It is a legacy interface and used to transverse a vector, Hashtable.	4.	It is not considered as a legacy interface but it can be used to traverse a hashMap, LinkedList, ArrayList, HashSet, TreeMap.
5.	It cannot make changes to collection.	5.	It can make changes or delete the elements from the collection while iteration.

Q9. List out the map interfaces.

Answer :

There are four map interfaces,

1. Map
2. Sorted map
3. Navigable map
4. Map.Entry.

Q10. Write about comparator interface.

Answer : Model Paper-I, Q1(h)

In general, a comparator compares two items of data. Java also means the same. In Java, Comparator is used to compare two stored objects in a Set or a Map etc.

The java.util.Comparator interface is used to dictate in which order we want the sorting of the objects of a data structure. Generally, Comparator is not needed if there's a natural sorting order. Natural sorting order means alphabetical order-A before B or 1 before 2 etc. A TreeSet or a TreeMap, by default, gives a natural sorting order, but if we want a different order, then we require comparator.

Q11. What is Arrays class?

Answer :

Arrays is a class in java.util package introduced from JDK 1.5 version, which provides many static methods to do operations on arrays like filling, comparing, sorting and searching. Earlier to the introduction of Arrays, these operations were done from scratch. Now, the coding has become easier with Arrays. Many static methods are overloaded.

Following is the class signature,

```
public class Arrays extends Object
```

Q12. What is the significance of Legacy class? Give example.

Answer : Nov./Dec.-17(R16), Q1(g)

The data structures introduced with JDK 1.0 version are called legacy classes and interfaces. Even though, the Collections framework is far more superior with its rich set of features, the legacy classes are not deprecated. Instead, some more methods are added and re-engineered to fit into collection framework.

Following table gives a list of legacy classes and interfaces.

Legacy Structure	Type of the Class	Functionality
Enumeration	interface	Used to iterate and print the elements
Vector	class(non-abstract)	Stores all types of objects
Stack	class(non-abstract)	Stores all types of objects(LIFO order)
Dictionary	abstract class	Stores key/value pairs
Hashtable	class(non-abstract)	Stores key/value pairs
Properties	class(non-abstract)	Stores key/value pairs

Q13. Write any two methods of dictionary.

Answer :

The two methods of dictionary are,

1. **elements( )**

The element( ) method returns an enumeration of values in the dictionary. Its syntax is,

```
Enumeration< v >elements( )
```

Where, *v* is the type of the values.

2. **get( )**

The get( ) method returns the object that has the value of the given key. It returns null object if the key is not found in the dictionary. Its syntax is,

```
V get(Objectkey)
```

Q14. What is the purpose of StringTokenizer class? Explain.

Answer :

(Model Paper-II, Q1(h) | Nov./Dec.-17(R16), Q1(h))

StringTokenizer Class separates string/text into tokens delimited by \s,\t,\n,\r and \f. It takes a string as input and parses it into token. This method is known as parsing. StringTokenizer provides lexer/scanner since it is the initial step in the process of parsing. It provides the following constructors.

	Constructor		Description
1.	StringTokenizer(String s)	1.	It takes the string to be tokenized as a parameter and considers white space as default delimiters.
2.	StringTokenizer(String s, String delimiters)	2.	It takes the string to be tokenized as a parameter and also specifies a delimiters.
3.	StringTokenizer(String s, String delimiters, boolean delim AsToken)	3.	It takes the string to be tokenized as a parameter and also specifies a delimiters.In addition to it, it returns s tokens if the value of delim AsToken is true.

PART-B
ESSAY QUESTIONS WITH SOLUTIONS

4.1 COLLECTIONS OVERVIEW, COLLECTION INTERFACES

15. What is Java Collections Framework? List out some benefits of Collections framework and explain.

April/May-18(R16), Q8(a)

OR

What are the best practices related to Java Collections Framework? Discuss.

Nov./Dec.-18(R16), Q9(a)

Answer :

Java Collection Framework

Java collection framework consist, hierarchy, interfaces and classes. This framework provides a standardized way for storing and manipulating group of objects defined in a program. The collection framework has been designed for the following reasons,

To increase the performance of collection like dynamic arrays, linked list, trees and hash tables.

To allow group of collections to work in a similar manner.

To easily extend the collection mechanism.

The Java collection framework is defined in most commonly used package called Java.util.pacakage. This package defines set of interface called collection interfaces and also a set of classes called collection classes. The collections interface is a core on which collection framework is build. They are usually defined for implementing collection classes. Beside this , the Java collection framework also consists of an important collection mechanism called algorithms. They are static methods defined inside collection classes for manipulating collection.

In addition to algorithm, a collections framework consists of a special type of interface called Iterator interface. This interface provides a standardized way to manipulate the element of collection class.

Benefits of Java Collections Framework

Following are the benefits of java collections framework,

i) It reduces programming effort.

ii) It reduces effort for designing new APIs.

iii) It allows inter operability among unrelated APIs.

iv) It increases program speed and quality.

v) It increases software reusability.

vi) It reduces effort for learning and using new APIs.

16. Explain in detail about collection interfaces.

Answer :

Model Paper-II, Q8(a)

Collection Interfaces

The collection framework of Java is built on a standard set of interfaces called collection interfaces. They are basically defined for implementing collection mechanism. Some of the interfaces of collection framework are as follows,

1. Collection interface
2. List interface
3. Dequeue interface
4. Set interface
5. Sorted set interface.

Collection Interface

This interface is the top most element of collection hierarchy. It allows users to work with different types of object.

Syntax: interface collection <E1>

void clear( ), int size, booleanadd(E object) are some of the methods of Collection interface.

2. **List Interface**

This interface is a generic interface which is derived from collection interface and is used to handle a list of elements present in queue. These elements undergo processing based on the location where they exist within the list through a zero-based index. It involves all the copies of elements.

Syntax: interface list<E>

Where, E indicates the type of the objects.

Element( ), Epeek( ) are some of the methods defined by the list interface.

3. **Dequeue Interface**

This interface is derived from Queue interface and is used to handle the double ended queue.

Syntax: interface Dequeue<E>

void push(e object) Eremove first( ), Epop( ) are some of the methods defined by the Dequeue interface.

4. **Set Interface**

This interface is derived from the topmost interface called Collection interface. They are usually defined for handling unique elements present in the set.

Syntax: interface Set<E1>

add(int index –val E 1 object), E get(int index - val) Eremove(int index-val) are some of the methods defined by the Set interface.

5. **SortedSet Interface**

This interface is derived from Set interface inorder to handle the sorted set.

Syntax: interface Set<E>

4.2 THE COLLECTION CLASSES – ARRAY LIST, LINKED LIST, HASH SET, TREE SET, PRIORITY QUEUE, ARRAY DEQUE

Q17. Explain the different types of standard collection classes.

Answer :

Model Paper-II, Q9

Collection Classes

The subclasses of collection interface are called collection classes.

Some of the collection classes provide full implementation to collection interfaces so they can be used as it is, and others are abstract classes. The collection classes defined by collections framework are,

1. AbstractCollection
2. AbstractList
3. AbstractSequentialList
4. LinkedList
5. ArrayList
6. AbstractSet
7. HashSet
8. LinkedHashSet
9. TreeSet
10. PriorityQueue
11. ArrayDeque.

Some of the commonly used collection classes are as follows,

1. **ArrayList Class**

The ArrayList class inherits the functionalities of AbstractList class and List interface. The ArrayList creates the dynamic arrays. The size of an array needs not to be mentioned while creating the array list. The size of an array list grows dynamically to accommodate new objects and shrinks when the objects are removed.

Constructors

ArrayList( )

ArrayList(Collection coll)

ArrayList(int size)

The first form of constructors creates an empty list. The second constructor creates and initializes an array list with the elements of the Collection 'coll'. The third constructor specifies the initial size of arraylist which can grow dynamically when elements are added to it.

This class uses variable-length array (dynamic array) to store the object references. This helps in dynamically increasing/decreasing the size of the collection whenever the elements of array grow or shrink.

(i) **ArrayList( ):** This constructor creates an empty list and initializes it to 0.

(ii) **ArrayList(collections co):** This constructor creates a list to which the user can add elements from some other list.

(iii) **ArrayList(int capacity):** This constructor creates an empty list and initializes it to the specified value. The method of this class are as follows,

1. **set( ):** This method is used to set the values to array elements.
2. **get( ):** This method is used to access/get the array elements.
3. **add( ):** This method is used add elements to the list.
4. **remove( ):** This method is used to remove an element from the list.

Methods

(i) **void ensureCapacity(int size):** This method increases the size of an array list to the size specified by 'size'.

(ii) **void trimToSize( ):** This method reduces the size of an array list.

Program

```
import java.util.*;
class ColorsList
{
    public static void main(String args[ ])
    {
        ArrayList color = new ArrayList( );
        System.out.println("Initial size of Color list:"
                                        +color.size( ));
        color.add("Red");
        color.add("Green");
        color.add("Yellow");
        color.add(0, "Orange");
        System.out.println("Size of color list after
                        addition:" + color.size( ));
        System.out.println("Color list is:" + color);
        color.remove("Red");
        System.out.println("Size of color list after
                        deletion:" + color.size( ));
        System.out.println("The final Color list is:" +
                                        color);
    }
}
```

Output

```
Initial size of color list : 0
Size of color list after addition : 4
Color list is : [Orange, Red, Green, Yellow]
Size of color list after deletion : 3
The final Color list is : [Orange, Green, Yellow]
```

2. LinkedList Class

The LinkedList class inherits AbstractSequentialList class and List interface. It creates a linked list.

Constructors

LinkedList( )
LinkedList(Collection coll)

The first constructor creates an empty linked list while the second constructor initializes the linked list with the elements of the Collection 'coll'.

Methods

(i) **void addFirst(Object obj), void addLast(Object obj):** The addFirst( ) and addLast( ) methods inserts an object 'obj' at the beginning and at the end of the linked list respectively.

(ii) **Object getFirst( ), Object getLast( ):** The getFirst( ) method returns the first element and getLast( ) method returns the last element of the linked list.

(iii) **Object removeFirst( ), Object removeLast( ):** The removeFirst( ) method deletes the first element and removeLast( ) method deletes the last element from the linked list.

Program

```
import java.util.*;
class LListDemo
{
    public static void main(String args[ ])
    {
        LinkedList color = new LinkedList( );
        color.add("Cyan");
        color.add("White");
        color.addFirst("Yellow");
        color.add(2, "Pink");
        color.addLast("Blue");
        color.add("Light green");
        System.out.println("Linked list after
                                addition:" + color);
        color.remove(1);
        color.removeFirst( );
        color.removeLast( );
        System.out.println("Linked list after
                                deletion:" + color);
        Object obj = color.get(2);
        System.out.println("The color at index-2:" +
                                (String) obj);
        color.set(2, "Red");
        System.out.println("After setting the final
                            Linked list is:"+ color);
    }
}
```

Output

List list after addition : [Yellow, Cyan, Pink, White, Blue, Light green]

Linked list after deletion : [Pink, White, Blue]

The color at index-2 : Blue

After setting the final Linked list is : [Pink, White, Red]

3. HashSet Class

The HashSet inherits AbstractSet class and Set interface. The HashSet class is used to store the collection created by it into the hash table that stores the unique key. These keys are converted into hash code by applying a hashing technique. This hash code is then used to retrieve the data associated with the key. The hash table does not store the elements in any sorted order. Hashing is useful to perform the operations such as add( ), remove( ), size( ) and contains( ).

Constructors

HashSet( )

HashSet(Collection coll)

HashSet(int size)

HashSet(int size, float fill_ratio)

The first form of constructor creates an empty hash set. The second form of constructor initializes the hash set with the elements of collection 'coll'. The third form of constructor specifies the size of the hash set. The last form of constructor specifies both the size and the fill ratio. The fill ratio specifies the allowed capacity of hash set that can be full before it is resized. It ranges between 0.0 and 1.0. The default value for fill ratio is 0.75.

The HashSet does not define its own methods.

Program

```
import java.util.*;
class HSet
{
    public static void main(String args[ ])
    {
        HashSet color = new HashSet( );
        color.add("White");
        color.add("Green");
        color.add("Black");
        color.add("Blue");
        System.out.println("The Hash set is :" + color);
    }
}
```

Output

The Hash set is : [White, Blue, Black, Green]

4. TreeSet Class

The TreeSet class inherits Set interface. It creates a collection that uses a tree for storage. Unlike the HashSet class, TreeSet stores the objects in sorted order i.e., in ascending order. Therefore, the large amount of sorted information can be accessed and retrieved very fast.

Constructors

TreeSet( )
TreeSet(Collection coll)
TreeSet(Comparator cp)
TreeSet(SortedSet sort)

The first form of constructor creates a default tree set whose elements will be sorted in ascending order. The second constructor initializes the tree set with the elements of the Collection 'coll'. The third constructor creates an empty tree set whose elements will be sorted in the order specified by the Comparator object 'cp'. The final constructor initializes the tree set with the elements of the SortedSet 'sort'.

Programs

```
import java.util.*;
class TSet
{
    public static void main(String args[ ])
    {
        TreeSet color = new TreeSet( );
        color.add("Brown");
        color.add("White");
        color.add("Yellow");
        color.add("Cyan");
        System.out.println("The tree set is :" + color);
    }
}
```

Output

The tree set is : [Brown, Cyan, White, Yellow]

5. **Priority Queue**

This class is derived from abstract queue class to implement priority based queues.

Syntax

class Priority Queue<E1>

Constructor

The following are the constructors defined by priority queue,

(i) Priority Queue( )
(ii) Priority Queue(int capacity)
(iii) Priority Queue(comparator<? super E1>compt)
(iv) Priority Queue(int capacity, comparator<?super E1>compt)
(v) Priority Queue(collection<?extends E1>co)
(vi) Priority Queue(priority queue<?extends E1>co)
(vii). Priority Queue(sortedset<?extends E1>co)

6. **Array Dequeue**

This class is derived from abstract collection class for implementing Dequeue Interface.

Syntax

class Dequeue<E1>

Constructor

The following are the constructors defined by Array Dequeue,

(i) Array Dequeue( )
(ii) Array Dequeue(int size)
(iii) Array Dequeue(collection <?extends E1>co)

Q18. What is difference between ArrayList and LinkedList in collection framework? Explain.

April/May-18(R16), Q8(b)

Answer :

	ArrayList		LinkedList
1.	Array list is used to store set of elements sequentially using indexes.	1.	Linked list is used to store set of elements sequentially using links.
2.	Array list provides random access for the elements present in the list.	2.	Linked list does not provide random access for the elements present in the list.
3.	Array list is extended by the Abstract list class.	3.	Linked list is extended by the Abstract sequential list.
4.	Array list is implemented by the list interface.	4.	Linked list is implemented by the list, deque, queue.
5.	The elements are accessed very fastly from the array list.	5.	The elements are accessed very slowly from the linked list.
6.	The manipulation for the elements of array list is slow.	6.	The manipulation for the elements of the linked list is fast.
7.	The behaviour of array list is as a list.	7.	The behaviour of linked list as a list and queue.

4.3 ACCESSING A COLLECTION VIA AN ITERATOR, USING AN ITERATOR

Q19. What is Iterator and ListIterator?

OR

What is different between Iterator and ListIterator? Explain different ways to iterate over a list.

Answer :

Nov./Dec.-18(R16), Q8(b)

Iterator

In computer vocabulary, iterator means an object with which user can travel all through a range of elements. In Java, Iterator is an interface that allows programmers to access a collection.

Following is the signature of iterator interface.

Public Interface Iterator

Enumeration interface is replaced by Iterator in collections framework. The advantages of iterator over Enumeration are,

- It makes the caller to delete the elements from collection while performing iteration by using well-defined semantics.
- Method names are improved.
- Any index number can be printed.

It includes three abstract methods. They are as follows,

- **boolean hasNext( ):** It returns true if the collection has more elements.
- **Object next( ):** It returns the next element in the collection.
- **void remove( ):** It removes the current element in the collection.

In order to use iterator the following steps should be performed.

Step 1: Obtain an object of Iterator on the collection class by calling iterator( ) method on the collection. (by default, the pointer is placed on the first element so that all the elements are printed. But, user can choose the number of elements to print).

Step 2: Create a control loop to iterate through all the elements by using **hasNext( )** method.

Step 3: Extract each element by calling **next( )** method.

The following code illustrates these steps.

```
Iterator it = list.Iterator( ) ;  // 1st step
while(it.hasNext( ) )             // 2nd step
{
    System.out.println(it.next( ) ) ; // 3rd step
}
```

Here, list is an object of a collection (e.g : ArrayList).

ListIterator

ListIterator is an interface that allows programmers to loop through the collection in both forward and backward directions. It also allows to modify the element in collection. It is used only for those collections that implement List. It has the following methods.

void add(Object O): This method inserts the given element before the element returned by next( ).

boolean hasNext( ): This method returns true if there is an element after the current element in the collection. Otherwise, it returns false.

boolean hasPrevious( ): This method returns true if there is an element before the current element. Otherwise, it returns false.

Object next( ): This method returns the element after the current element, if there is one. Otherwise, it throws NoSuchElementException.

int nextIndex( ): This method returns the index of the next element. However, if there is no next element, it returns the size of the list.

E Previous( ): This method returns the element before the current element. If there is no previous element, it throws NoSuchElement Exception.

int PreviousIndex( ): This method returns the index of the previous element, if there is one. However if there is no previous element, it returns –1.

void remove( ): This method deletes the current element from the list. It throws IllegalStaticException if there is no current element.

void set(Object O): This method assigns the given object to the current element in the list.

Program

Java program to illustrate iterators.

```
import java.util.*;
class IteratorEx

    public static void main(String args[])
    {
        ArrayList<String> array_list = new ArrayList<String>();
        array_list.add("AB");
        array_list.add("XY");
        array_list.add("CD");
        array_list.add("PQ");
        array_list.add("EF");
        array_list.add("RS");
        System.out.print("The contents of Array List are:");
        Iterator<String> itrtr = array_list.iterator();
        while(itrtr.hasNext())
        {
            String ele = itrtr.next();
            System.out.print(ele + " ");
        }
        System.out.println();
        ListIterator<String> litr = array_list.listIterator();
        while(litr.hasNext())
        {
            String ele = litr.next();
            litr.set(ele + "+");
        }
```

```
        System.out.print("After modifications, the contents of Array List are:");
        itrtr = array_list.iterator();
        while(itrtr.hasNext())
        {
            String ele = itrtr.next();
            System.out.print(ele + " ");
        }
        System.out.println();
        System.out.print("After modifications, the Array List in backwards is:");
        while(litr.hasPrevious())
        {
            String ele = litr.previous();
            System.out.print(ele + " ");
        }
        System.out.println();
    }
}
```

Output

The contents of Array List are : AB XY CD PQ EF RS

After modifications, the contents of Array List are : AB+ XY+ CD+ PQ+ EF+ RS+

After modifications, the Array List in backwards is : RS+ EF+ PQ+ CD+ XY+ AB+

Different ways to Iterate Over a List

List can be iterated by using two methods which are as follows,

(i) iterator

(ii) for each loop.

(i) By Using iterator

```
Iterator it = list.Iterator( )
while(it.hasNext( ))
{
    System.out.println(it.next( ));
}
```

(ii) By Using for-each Loop

```
for(int a : val)
{
    System.out.println(a);
}
```

Q20. Discuss the differences between HashList and HashMap, Set and List.

Answer :

Nov./Dec.-17(R16), Q9(b)

Differences Between HashList and HashMap

	HashList		HashMap
1.	HashList is a data structure used to store the objects in hash table and list.	1.	HashMap is the implementation of map interface as hash table.
2.	It is the combination of hashmap and doubly linked list.	2.	It is same as that of hash table, but its not synchronized.
3.	It has key value pairs.	3.	It also has key value pairs contains values based on the key.
4.	It stores the data in array and dictionary/hash-map.	4.	It stores data into multiple singly linked list of entries. These lists are registered in array of entry.

Differences Between Set and List

	List		Set
1.	List is the ordered collection of data.	1.	Set is the unordered collection of data.
2.	They can duplicate values.	2.	They does not contain duplicate values.
3.	It allows null values.	3.	It allows atmost one null value.
4.	It maintains the insertion order of elements.	4.	Only few implementations of it maintains insertion order.
5.	It has one legacy class.	5.	It does not have any legacy class.
6.	Various implementations of it are ArrayList, LinkedList etc.	6.	Various implementations of it are HashSet, Linked-HashSet, TreeSet etc.

4.4 THE FOR-EACH ALTERNATIVE

Q21. Write a short notes on for-each alternative.

Answer :

For-each Alternative to Iterators

The for-each version of the for loop is very simple to iterate the set of objects instead of using an iterator. It is mostly used to modify the contents of a set of objects or to access the elements in reverse order can not modified. '*For*' can be used on the collection that implements iterable interface.

Example

The following is the example to calculate the sum of the set of elements by using for loop,

```
import java.util.*;
class ForEachExmp
{
    public static void main(String args[ ])
    {
        ArrayList<Integer> val = new ArrayList<Integer>( );
        val.add(2);
        val.add(4);
        val.add(6);
        val.add(8);
        System.out.println("Values are :");
        for(int a : val)
        System.out.print(a + " ");
        System.out.println( );
        int total = 0;
        for(int a : val)
        total += a;
        System.out.println("sum of all values :" + total);
    }
}
```

Output

```
Values are :
2 4 6 8
sum of all values : 20
```

In the above example, the for loop used is simple and short than the iterator. By using it, the set of objects are iterated in forward direction and the contents of it are not modified.

4.5 MAP INTERFACES AND CLASSES, COMPARATORS, COLLECTION ALGORITHMS, ARRAYS

Q22. Explain map interfaces with suitable examples.

Answer :

A map is an object that stores key/value pairs. Each key is unique and has its associated value, which may be repeated. Some maps allow keys and values to be null. Each key and value are object. Maps allow to find a value for a given key. An important point about maps is that, they can not obtain an iterator to cycle through a map. Thus, for-each style for loop cannot be used with maps.

There are four map interfaces,

1. Map
2. SortedMap
3. NavigableMap
4. Map.Entry.

1. Map Interface

A Map stores key/value pairs. It will not allow duplicate keys. Each key maps to one value. When the key is supplied, Map returns the value associated with it. A few maps allow null values and null keys also.

Following is the class signature,

public interface Map.

Map and its subclasses cannot use Iterator to retrieve the elements because Map does not implement Iterator interface. Keys must be unique i.e., map does not allow duplicate keys. Keys and values must be objects of any Java class.

Methods

The methods defined by the Map interface are as given in the below table,

	Method	Description
1.	void clear( )	Deletes all key/value pairs of the invoking Map object.
2.	boolean containsKey(Object key)	Returns true if the given key is found in the invoking Map object. Otherwise, returns false.
3.	boolean containsValue(Object value)	Returns true if the given value is found in the invoking Map object. Otherwise, returns false.
4.	Set<Map.Entry<K, V>> entrySet( )	Returns a Set of (key, value) pairs in the invoking Map object in the form of Map.Entry objects.
5.	boolean equals (Object O)	Returns true if the given object is a Map object and has same (key, value) pairs as those in the invoking Map object. Otherwise, returns false.
6.	V get(Object key)	Returns the value corresponding to the given key. Return null if the key is not found.
7.	int hashCode( )	Returns the hash code for the invoking Map object.
8.	boolean isEmpty( )	Returns true if the invoking Map object is empty. Otherwise, returns false.
9.	Set<K> keySet( )	Returns a Set of keys that are present in the invoking Map object.
10.	V put(K key, V value)	Inserts the given key/value pair in the invoking Map object. If the key already exist then the put( ) overwrites the value with the given value and returns the overwritten value. However, if the key is not present then it returns a null.
11.	void putAll (Map<?extends K, ?extends V map)	Inserts the key/value pairs of the given Map object into the invoking Map object.
12.	V remove (Object key)	Removes that key, value pair whose key matches with the given key.
13.	int size( )	Returns the number of elements in the invoking Map object.
14.	Collection <V> values( )	Returns a collection of values present in the invoking Map object.

Table: Methods in Map Interface

Program

```
import java.util.*;
class MapExample
{
    public static void main (String args[ ])
    {
        Map<Object, String> map = new HashMap<Object, String>( );
        map.put(new Integer (3), "Three");
        map.put(new Integer (4), "Four");
        map.put(new Integer (5), "Five");
        map.put(new Integer (1), "One");
        map.put(new Integer (2), "Two");
        System.out.println("Number of elements in map:" + map.size( ));
        System.out.println("Key/value pairs are:");
        System.out.println(map);
    }
}
```

Output

Number of elements in map:5
Key/value pairs are:
<1=One, 2=Two, 3=Three, 4=Four, 5=Five>

1. **SortedMap Interface**

It is just a sorted version of Map where elements can be sorted. It extends Map interface. A SortedMap is a Map contains the elements in ascending order. They are sorted based on keys natural ordering. User can specify the ordering also by supplying a Comparator.

Following is the class signature,

public interface SortedMap extends Map

Methods

The methods defined by the SortedMap interface are as shown in the below table,

	Method	Description
1.	Comparator<?super K> comparator( )	It returns the comparator of the invoking SortedMap object. It returns null if map objects is invoked using natural ordering.
2.	K firstKey( )	It returns first key of invoking Map object.
3.	K lastKey( )	It returns last key of invoking Map object.
4.	SortedMap<K, V> subMap(k start, k end)	It returns a SortedMap object that contains a set of key/value pairs whose keys are greater than or equal to 'start' and less than 'end'.
5.	SortedMap<K, V> tailMap(K start)	It returns a SortedMap object that contains a set of key/value pairs whose keys greater than or equal to 'start'.
6.	SortedMap<K, V> headMap(K end)	It returns a SortedMap object that contains a set of key/value pairs whose keys are less than 'end'.

Table: Methods in SortedMap Interface

Program

```
import java.util.*;
public class SortedMapDemo
{
    public static void main(String args[ ])
    {
        SortedMap smap = new TreeMap( );
        //Insert some (key, value) pairs in smap
```

```
        smap.put("7", "Seven");
        smap.put("25", "Twenty Five");
        smap.put("9", "Nine");
        smap.put("1", "One");
        System.out.println("the lowest key in the sortedmap is :" + smap.firstKey( ));
        System.out.println("the highest key value in the sorted map is :" + smap.lastKey( ));
        System.out.println("the lowest key values in the sorted map in increasing order are :\n" + smap);
    }
}
```

Output

```
the lowest key in the sortemap is : 1
the highest key value in the sorted map is : 9
the lowest key values in the sorted map in increasing order are :
<1=One, 25=Twenty Five, 7=Seven, 9=Nine>
```

3. NavigableMap Interface

The NavigableMap interface is derived from SortedMap. It supports the retrieval of (key, value) pairs that closely match to the given key or keys.

The methods defined by NavigableMap interface are as shown in the below table,

	Method	Description
1.	Map.Entry<K, V> ceilingEntry (K object)	Searches the map object for the smallest key, which is greater than or equal to the given key. Returns the entry for the smallest key, if it is found. Otherwise, returns null.
2.	K ceiling key(K object)	Searches the map object for the smallest key, which is greater than or equal to the object. Then, it returns the entry for the smallest key, if it is found. Otherwise, returns null.
3.	NavigableSet<K>descending keySet( )	Returns a NavigableSet whose descending keySet( ) keys are in decreasing order of the keys in invoking map object.
4.	NavigableMap<k, V> descendingMap( )	Returns a NavigableMap whose keys and values are in decreasing order of the keys and values in invoking map object.
5.	Map.Entry<K, V> firstEntry( )	Returns the first (key, value) pair whose key is the smallest one in the map.
6.	K higherkey (K object)	Searches the set for the largest key that is greater than the given object. Returns the largest key if it is found. Otherwise, returns null.
7.	Map.Entry<K, V> lastEntry( )	Returns the entry that has the largest key in the map.
8.	NavigableSet<K> navigableSetkey( )	Returns a NavigableSet that has the keys of the invoking map.
9.	Map.Entry<K, V> pollFirstEntry( )	Returns the first entry that has the smallest key and removes it from the map. Returns null if the map is empty.
10.	Map.Entry<K, V> pollLastEntry( )	Returns the last entry that has the largest key and removes it from the map. Returns null if the map is empty
11.	Map.Entry<K, V> floorEntry(K object)	It searches the map to find the largest key, K i.e., lessthan or equal to an object (K ≤ object). It returns the entry for the largest key when it is obtained. Else, it returns null.
12.	K floorKey(K object)	It searches the map to find a largest key K i.e., less than or equal to an object (K ≤ object). It returns largest key, when it is found. Else, it returns null value.
13.	NavigableMap <K, V> headMap(K upperBound, boolean incl)	It returns a NavigableMap object which involves each entry at the invoking map having keys which can be less than an uppderBound. When 'incl' is set true, it involves the element that is equivalent to an upperBound. Therefore, the map which is found can be backed using an invoking map.

14.	Map.Entry<K, V>higherEntry(K object)	It searches the set object to find a largest key K i.e., greater than an object (K ≥ object). It returns the entry for a largest key when it is found. Else, it returns null.
15.	Map.Entry<K, V>lowerEntry(K object)	It searches the set object to find a largest key K i.e., less than an object (K ≤ object). It returns the entry for a largest key when it is found. Else, it returns 'null'.
16.	K lowerKey(K object)	It searches the set object to find a largest key K i.e., less than an object (K < object). Then, it returns a particular key when it is found. Else, it returns 'null'.
17.	NavigableMap<K, V> subMap(K lowerBound, boolean lowIncl, K upperBound, boolean highIncl)	It returns a NavigableMap object which involve each entry at the invoking map which consists of keys that can be greater than lowerBound as well as less than the upperBound. In certain cases, when lowIncl is set to true, then it adds an element that is equal to lowerBound. Where as, it involves an element that is equal to highIncl is added when highIncl is set true. The resultant map can be backed through an invoking map.
18.	NavigableMap<K, V>tailMap(K lowerBound, boolean incl)	It returns a NavigableMap object which consists of every at the invoking map having keys that are greater than lowerBound. Whenever incl is set true, it equates the elements with the lowerBound. The resultant map can be backed through an invoking map.

Table: Methods in NavigableMap Interface

Program

```
import java.util.*;
import java.util.concurrent.*;
public class NavigableMapDemo

    public static void main (String args[ ])
    {
        NavigableMap<Integer, String> nmap = new TreeMap<Integer, String>( );
        //Insert some elements
        nmap.put(1, "Summer");
        nmap.put(2, "Winter");
        nmap.put(3, "Rainy");
        nmap.put(4, "Autumn");
        nmap.put(5, "Fall");
        System.out.println("The Navigable map in descending order is :\n" + nmap.descendingMap( ));
        System.out.println("The first entry in the Navigable map is :\n" + nmap.firstEntry( ));
        System.out.println("The last entry in the Navigable map is :\n" + nmap.lastEntry( ));
        System.out.println("Removing the first entry:\n" + nmap.pollFirstEntry( ));
        System.out.println("Removing the last entry:\n" + nmap.pollLastEntry( ));
        System.out.println("The Navigable map in descending order is:\n" + nmap.descendingMap( ));
    }
}
```

Output

```
The Navigable map in descending order is :
{5 = Fall, 4 = Autumn, 3 = Rainy, 2 = Winter, 1 = Summer}
The first entry in the Navigable map is : 1 = Summer
The last entry in the Navigable map is : 5 = Fall
Removing the first entry : 1 = Summer
Removing the last entry : 5 = Fall
The Navigable map in descending order is,
{4 = Autumn, 3 = Rainy, 2 = Winter}.
```

4. Map.Entry Interface

The Map.Entry interface is an inner class of Map that allows programmers to work with each entry in a map.

The methods defined by the Map.Entry interface are as shown in the below table,

	Method	Description
1.	boolean equals(Object object)	Returns true if the given object is a Map.Entry object whose key and value matches with the key and value of the invoking object.
2.	k getKey( )	Returns the key of the invoking Map.Entry Object.
3.	V getValue( )	Returns the value of the invoking Map.Entry Object.
4.	int hashCode( )	It returns the hash code to invoke Map.Entry object.
5.	V setValue (V value)	Sets the value of the invoking Map.Entry object to the given value. It throws classCastException if the given value is not the correct type for the invoking Map object and IllegalArgumentException if there is any problem with the given value. A NullPointerException is thrown if the given value is null and the invoking Map object restricts null values. It throws Unsupported OperationException if the invoking Map object cannot be changed.

Table: Methods in Map.Entry Interface

Program

```
import java.util.HashMap;
import java.util.Map;
import java.util.Iterator;
import java.util.Set;
import java.util.*;
public class MapEntryDemo
{
    public static void main (String args[ ])
    {
        Map map = new HashMap( );
        //Insert some elements in the map
        map.put("7", "Sunday");
        map.put("3", "Wednesday");
        map.put("1", "Monday");
        map.put("2", "Tuesday");
        map.put("4", "Thursday");
        map.put("6", "Saturday");
        map.put("5", "Friday");
        Set s = map.entrySet( );
        Iterator i = s.iterator( );
        System.out.println("The key/value pairs are");
        while(i.hasNext( ))
        {
            Map.Entry e = (Map.Entry)i.next( );
            System.out.println(e.getKey( ) + " " + e.getValue( ));
        }
}
```

Output

The key/value pairs are,

1. Monday
2. Tuesday
3. Wednesday
4. Thursday
5. Friday
6. Saturday
7. Sunday.

Q23. Discuss any two Map classes.

Answer :

There are many Map classes pre-defined in the Java core classes. That is, many subclasses exist for Map interface. Some of the important classes are as follow,

Class Name	Functionality
AbstractMap	It implements Map interface and a super class for many classes.
HashMap	It works like Hashtable, storing key/value pairs.
TreeMap	It gives the elements as nodes comprising a tree.
WeakHashMap	It is like HashMap where weak keys are used.

(i) **HashMap**

HashMap is like Hashtable which stores key/value pairs. It implements Map interface and user can use all the operations (methods) of Map interface. **It allows null values as well as the null keys.** The HashMap class is almost similar to Hashtable, except that it is unsynchronized and allows null values. The order of elements is not guaranteed.

The class signature of HashMap class is,

public class HashMap extends AbstractMap implements Map, Cloneable, Serializable

HashMap constructor can be specified with – **initial capacity** and **load factor**. The **capacity( )** of HashMap gives the storage capacity and the size( ) gives the actual elements present in the HashMap. The initial capacity gives the starting capacity of HashMap when it is created. The **load factor** gives at what rate the capacity must be increased when the existing capacity gets exhausted. The default load factor is 0.75.

Higher load factor values decrease the space overhead but increase the lookup cost (searching takes long time). The HashMap methods are not synchronized.

Constructors

1. **HashMap( ):** It creates an empty HashMap object with an initial capacity of 10 and a default load capacity of 16.
2. **HashMap(int cap):** It creates an empty HashMap object with capacity "cap" and default load capacity 16.
3. **HashMap(int cap, float loadfac):** It creates an empty HashMap object with an initial capacity specified as "cap" and a load capacity specified as "loadcap".
4. **HashMap(Map m1):** It creates a HashMap object with the elements of another Map class specified by "m1".

Methods

1. **boolean containsKey(Object key):** It returns true if the key is found.
2. **boolean containsValue(Object value):** It returns true if the value is found.
3. **Object get(Object key):** It returns the value associated with the key.
4. **boolean isEmpty( ):** It returns true if this map does not contains no key-value mappings.
5. **Object remove(Object key):** It removes the entry that matches key.
6. **int size( ):** It returns the number of key-value pairs.
7. **void clear( ):** It removes the entries.

Program

```
import java.util.*;
public class HashMapDemo
{
    public static void main(String args[ ])
    {
        HashMap hm = new HashMap(10, 0.5f );
        hm.put("John", "Serena");
        hm.put("salary", new Double(9999.99));
        hm.put("distance", new Integer(350));
        hm.put("expenditure", "8888.88");
        System.out.println("Salary key exists:" + hm.containsKey ("salary"));
        System.out.println("Sita value exists:" + hm.containsValue ("Sita"));
        System.out.println("Value of Rama is" + hm.get("Rama"));
        System.out.println("No. of elements:" +hm.size( ));
        System.out.println("\nPrinting the values using Iterator:");
        Set allKeys = hm.keySet( ); //Returns a set of keys contained in HashMap
        Iterator It = allKeys.iterator( ); //Returns an object of iterator
        while (It.hasNext( ))
        {
            Object obj = It.next( );
            System.out.println(obj + " : " + hm.get(obj));
        }
    }
}
```

Output

```
Salary key exists : true
Sita value exists : false
Value of Rama is null
No. of elements : 4
Printing the values using Iterators :
expenditure : 8888.88
distance : 350
John : Serena
Salary : 9999.99
```

(ii) TreeMap

TreeMap class extends AbstractMap and implements SortedMap. This class stores the elements in ascending key order, sorted according to the natural order based on the keys. The order can be changed by supplying a comparator at creation time. The methods are not synchronized.

Constructors

1. **TreeMap( ):** It creates an empty TreeMap sorted as per the key's natural order(ascending order).
2. **TreeMap(Map m1):** It creates a TreeMap object with the elements of another Map class as per key's natural order.
3. **TreeMap(Comparator comp):** It creates an empty TreeMap object with the sorting order of the Comparator specified by "comp".
4. **TreeMap(SortedMap m1):** It creates a TreeMap object with the same elements of the SortedMap specified by "m1" with the same sorting order of "m1".

This class does not define its own methods.

The signature of TreeMap class is,

public class TreeMap extends AbstractMap implements SortedMap, Cloneable, Serializable

```
ogram
port java.util.*;
ss marksMap

blic static void main(String args[])

eeMap<Integer,String> map=new TreeMap<Integer,String>();
map.put(100, "Maths");
map.put(96, "Science");
map.put(95, "Social");
map.put(93, "English");
for(Map.Entry m:map.entrySet()){
System.out.println(m.getKey()+ ":" +m.getValue());
}

}
```

Output

93 : English
95 : Social
96 : Science
100 : Maths

Q24. What is Comparator interface and why is it used?

Answer :

Comparator Interface

In general, a comparator compares two items of data. Java also means the same. In Java, Comparator is used to compare two stored objects in a Set or a Map etc.

The java.util.Comparator interface is used to dictate in which order we want the sorting of the objects of a data structure. Generally, Comparator is not needed if there's a natural sorting order. Natural sorting order means alphabetical order-A before B or 1 before 2 etc. A TreeSet or a TreeMap, by default, gives a natural sorting order, but if we want a different order, then we require comparator.

User can pass a Comparator object, which defines the sorting order, as parameter to a TreeMap or TreeSet constructor. It defines two methods - **compare( )** and **equals( )**.

public int compare(Object obj1, Object obj2)

public boolean equals(Object obj).

1. **compare( )** method takes two Objects and returns 0, 1, –1 depending upon the parameters passed. 0 is returned when both objects are same. If first object is greater than second, returns 1 else –1.
2. **equals( )** method returns boolean value true, if both objects are same. Otherwise, it returns false. Generally, we do not override this method because an implemented concrete method is inherited from Object class.

Q25. What is Comparable and Comparator Interface? Differentiate between them.

Nov./Dec.-18(R16), Q9(b)

Answer :

Comparable Interface

Comparable interface allows the comparison of objects of a class. The class that implements *Comparable* interface contains objects can be compared. *Comparable* interface can be declared as,

interface *Comparable* < T >

Where, *T* represents the object type being compared.

Comparable interface declares a method called compareTo( ), which can be used for ordering the objects of the classes. The method is declared as follows,

int compareTo(T object)

This method compares the invoking object with object and returns zero, a negative value or a positive value.

zero – If the values are equal.

Negative value – If the invoking object has a lower value.

Positive value – If the invoking object has a larger value.

In Java, several classes implement the *Comparable* interface. They are Byte, Character, Double, Float, Short, Long, String and Integer. All these classes defines a *compareTo*( ) method.

Comparator Interface

For answer refer Unit-IV, Q24.

Differences between Comparable and Comparator Interface

	Comparable Interface		Comparator Interface
1.	It is available in java.lang package.	1.	It is available in java.util package.
2.	It contains compare To( ) method to sort the elements/objects.	2.	It contains compare( ) method to sort the elements/objects.
3.	It provides a natural or single sorting order.	3.	It provides unnatural or multiple sorting order.
4.	Comparable type elements can be sorted by using collections. Sort(List) method.	4.	Comparator type elements can be sorted by using collections. Sort (List, Comparator) method.
5.	It makes changes in the original class.	5.	It does not make any changes in the original class.
6.	It is declared as, interface comparable <T>	6.	It is declared as, interface comparator <T>

Q26. Using comparator write a Java program to sort a tree set in a reverse order.

Answer :

```
import java.util.*;
    // Implement Comparator interface
class MyComparator implements Comparator<Strings>
{
    //Override compare( ) method to reverse the order
    public int compare(String str1, String str2)
    {
        String s1, s2;
        s1 = str1;
        s2 = str2;
    // compareTo( ) invoked by s2 reverses the tree set
      return s2.compareTo(s1);
    }
}
class RevTreeSet
{
    public static void main(String args[ ])
    {
        // Create a tree set of strings
        TreeSet<String> tree = new TreeSet<String>(new MyComparator( ));
        // Add elements
        tree.add("Ashraf");
        tree.add("Shazia");
        tree.add("Ameer");
        tree.add("Rubin");
```

```
        tree.add(Haseena");
        // Display the treeSet
        System.out.println("TreeSet in a reverse order is");
        for(String element : tree)          //Display the elements
        System.out.println(element);
    }
}
```

Output

```
Shazia
Rubin
Haseena
Ashraf
Ameer
```

What are collection algorithms?

OR

What are the common algorithms implemented in Collections Framework? Discuss.

Answer : **April/May-18(R16), Q9(a)**

One of the components of the collections framework is algorithms (the others are interfaces and implementation class). The algorithms are pieces of reusable functionality provided by the Java collections. Algorithms can be used with collection classes and map classes. The Collection class contains many static methods, which are commonly known as algorithms. These static methods can be applied to collections classes for data manipulations.

JDK 1.5 (J2SE 5) added many methods (like checked methods) to the already existing algorithms, especially generics are added. Generics give type-safe elements in a data structure. That is, if we set the generic to strings, we can add only strings to a data structure.

The static methods take first argument as the collection class on which the operation is to be performed. The great majority of the algorithms provided by the Java platform operate on List instances, but a few of them operate on other collection classes.

The algorithms are helpful for the following operations on collections:

(a) Sorting
(b) Shuffling
(c) Routine Data Manipulation
(d) Searching
(e) Composition
(f) Finding Extreme Values

Some of the operations performed on collection are as follows,

a) Sorting

The sort algorithm arranges a list of elements in ascending order.

Example

The following algorithm sorts the list passed as command-line argument to the program.

```
import java.util.*;
public class SortDemo
{
    public static void main(String[] args)
    {
        List list = Arrays.asList(args);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

Output

```
C:\snr>java SortDemo costly red apple
[apple, costly, red]
```

(b) Shuffling

The shuffle algorithm does the reverse of what sort does. That is, the elements are never sorted but instead shuffled. If executed at different times, different outputs are obtained; that is shuffling will change. It orders the elements randomly at different times. It takes all possible permutations. This algorithm is useful in implementing games of chance. For example, it could be used to shuffle a list of playing cards representing a deck. Also, it's useful for generating test cases while testing software.

Example

File Name: ShuffleDemo.java

```
import java.util.*;
public class ShuffleDemo
{
    public static void main(String[ ] args)
    {
        List list = new ArrayList( );
        list.add("red");
        list.add("purple");
        list.add("yellow");
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

Output

```
C:\snr>java ShuffleDemo
[purple, red, yellow]

C:\snr>java ShuffleDemo
[purple, red, yellow]

C:\snr>java ShuffleDemo
[yellow, red, purple]

C:\snr>java ShuffleDemo
[purple, yellow, red]

C:\snr>java ShuffleDemo
[purple, red, yellow]

C:\snr>java ShuffleDemo
[yellow, red, purple]
```

In the above program, three String objects are taken – red, purple and yellow. When executed at different times, as in the above output, it gives different orders.

(c) Routine Data Manipulation

The Collections class provides five algorithms for performing routine data manipulation on List objects. They are,

(i) **reverse** : reverses the order of the elements in a List.

(ii) **fill** : It overwrites all the elements of List with given value. It is used to re-initialize a List.

(iii) **copy** : It accepts two arguments, namely a destination List and a source List. It copies the source elements into destination by overwriting its contents.

(iv) **swap** : It swaps the specified elements of a List.

(v) **addAll** : adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.

(vi) **min** : It finds the minimum element of a list.

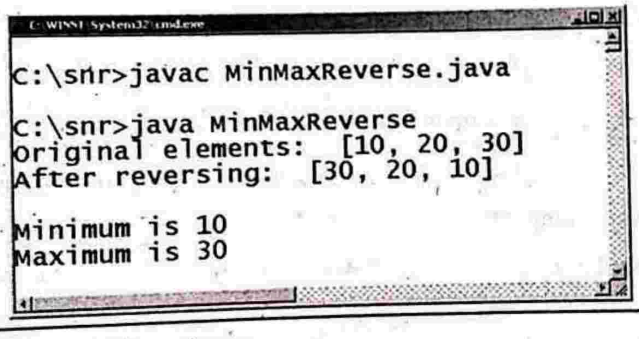
(vii) **max** : It finds the maximum element of a list.

Example

In the following program, reverse of the elements and minimum and maximum operations are done.

```
// Name: MinMaxReverse.java
import java.util.*;
public class MinMaxReverse
{
  public static void main(String[] args)
  {
        List list = new ArrayList( );
        list.add(new Integer(10));
        list.add(new Integer(20));
        list.add(new Integer(30));  // reverse operation
        System.out.println("Original elements:" + list);
        Collections.reverse(list);
        System.out.println("After reversing:" + list);  // retrieving minimum and maximum of elements
        System.out.println("\nMinimum is" + Collections.min(list));
        System.out.println("Maximum is" + Collections.max(list));
  }
}
```

Output



Q28. Describe Arrays class in java.util package.

Answer :

Arrays is a class in java.util package introduced from JDK 1.5 version, which provides many static methods to do operations on arrays like filling, comparing, sorting and searching. Earlier to the introduction of Arrays, these operations were done from scratch. Now, the coding has become easier with Arrays. Many static methods are overloaded.

Following is the class signature,

public class Arrays extends Object

Methods

Following are some static methods with their description.

1. **static int binarySearch(double myarray[], double num)**

   It is used to find a specified number (num) in the array (myarray) using binary search. This method should be used with sorted arrays. This method is overloaded that can take a byte array, char array, int array etc.

2. **static void equals(double myarray1[], double myarray2[])**

   It is used to check whether two arrays have the same elements. This method is overloaded that can take a byte array, char array, int array etc.

3. **static void fill(double myarray1[], double num)**

   It is used to fill all the array elements of the array (myarray1) with the specifeid value (num). This method is overloaded that can take a byte array, char array, int array etc.

4. **static void sort(double myarray1[])**

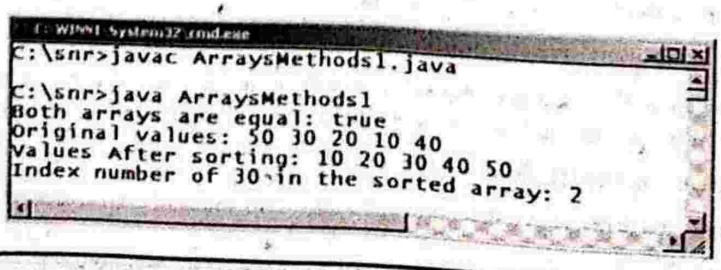
   It is used to sort the elements in the ascending order. This method is overloaded that can take a byte array, char array, int array etc.

Example

The following program illustrates the use of Arrays class.

```
import java.util.*;
public class ArraysMethods
    {
    public static void main(String[] args)
    {
    int marks[ ] = { 50,30, 20, 10, 40 };   //integer array
    int price[ ] = { 50,30, 20, 10, 40 };   // to compare two arrays whether their elements are same
    System.out.println("Both  arrays are equal:" + Arrays.equals(marks, price));
    System.out.print("Original values:");
    for(int i = 0; i < marks.length; i++)
    {
        System.out.print(marks[i]+" ");
    }
    System.out.println( );                  // sorting the elements of the array in ascending order
    Arrays.sort(marks);                     // elements in the original array itself are changed
   System.out.print("Values After sorting: ");
    for(int i = 0; i < marks.length; i++)
    {
        System.out.print(marks[i]+ " ");
    }
    int k = Arrays.binarySearch(marks, 30);         // to find the index number of an element
    System.out.println("\nIndex number of 30 in the sorted array:"+k);
  }
}
```

Output



4.6 THE LEGACY CLASSES AND INTERFACES – DICTIONARY, HASH TABLE, PROPERTIES, STACK, VECTOR

Q29. Explain about legacy classes and interfaces.

Answer : Model Paper-II, Q8(b)

Legacy Classes and Interfaces

The data structures introduced with JDK 1.0 version are called legacy classes and interfaces. Even though, the Collections framework is far more superior with its rich set of features, the legacy classes are not deprecated. Instead, some more methods are added and re-engineered to fit into collection framework.

Following table gives a list of legacy classes and interfaces.

Legacy Structure	Type of the Class	Functionality
Enumeration	interface	Used to iterate and print the elements
Vector	class(non-abstract)	Stores all types of objects
Stack	class(non-abstract)	Stores all types of objects(LIFO order)
Dictionary	abstract class	Stores key/value pairs
Hashtable	class(non-abstract)	Stores key/value pairs
Properties	class(non-abstract)	Stores key/value pairs

Enumeration Interface

The Enumeration interface allows to iterate through a list of sequentially stored elements in a data structure(this interface must be implemented by the data structure). It is replaced with Iterator interface in Collections framework.

Methods

It provides two abstract methods using which iteration can be done.

- **hasMoreElements( ):** It returns false when no elements exists
- **nextElement( ):** It retrieves the next element in the enumeration.

Method signatures of the above methods defined in the java.util package are as follows,

public abstract boolean hasMoreElements( );

public abstract java.lang.Object nextElement( );

```
//simple snippet of code illustrates how iteration can be done.
Enumeration e = vect.elements( );
while( e.hasMoreElements( ))
{
    System.out.println(e.nextElement( ));
}
```

elements( ) method of Vector class returns an object of Enumeration interface. The hasMoreElements( ) returns true as long as elements are in the Enumeration object. If all elements are exhausted it returns false and the loop terminates. nextElement( ) returns each object stored in the Enumeration interface object.

Q30. What is a Dictionary?

Answer :

Dictionary is an abstract class in the java.util package and operates much like Map interface. Like Map, it stores the elements as a list of key value pairs. It has the following methods,

1. **elements( ):** The element( ) method returns an enumeration of values in the dictionary.

Syntax

```
Enumeration < v > elements( )
```

Where, *v* is the type of the values.

2. **get( ):** The get( ) method returns the object that has the value of the given key. It returns null object if the key is not found in the dictionary.

Syntax

```
Vget(Object Key)
```

3. **isEmpty( ):** The isEmpty( ) method returns true if the dictionary is empty. Otherwise, it returns false.

Syntax

```
boolean isEmpty( )
```

4. **Keys( ):** The keys( ) method returns an enumeration of keys in the dictionary.

Syntax

```
Enumeration < k > Keys( )
```

Where, k is the type of the keys.

5. **put( ):** The put( ) method inserts a key and its corresponding value in the dictionary. It returns the previous value of the key if the key is already in the dictionary. It returns null, if the key is not in the dictionary.

Syntax

```
Vput(kkey, Vvalue)
```

6. **remove( ):** The remove( ) method deletes a key and its corresponding value from the dictionary, and returns the deleted value. It returns null if the key is not in the dictionary.

Syntax

```
Vremove(Object key)
```

7. **size( ):** The size( ) method will return the number of (key, value) pairs of dictionary.

Syntax

```
int size( )
```

From the above methods it is obvious that, a dictionary object is much like a Map or a List object.

Q31. Describe Hashtable with an example.

Answer :

Model Paper-I, Q8(a)

Hashtable

Hashtable is a concrete implementation of the dictionary class. It also stores information in the form of key value pairs but in a hash table. To store a pair, first a hash code is determined. After which, the hash code is used as an index and the values are stored in the hash table at that index.

Constructors

- **Hashtable( ):** It creates a hash table with default size of 11.
- **Hashtable(int size):** It creates a Hash table with the specified size.
- **Hashtable(int size, float fillRatio):** It creates a hash table with the specified size and fill ratio between 0.0 and 1.0
- **Hashtable(Map m):** It creates a Hash table with the elements in map m.

Hashtable was a part of original java.util package. However, with the introduction of collections, Hashtable was made to implement Dictionary as well as map interfaces. Thus, Hashtable has the methods of both of these interfaces. Following are the legacy methods defined by Hashtable.

1. **Methods void clear( ):** This method first resets the hash table and then empties it.
2. **object clone( ):** This method duplicates the invoking object and returns the same.
3. **boolean Contains(Object value), boolean containsValue (Object value):** These methods return true if there is some value in the hashtable that matches the given value. Otherwise, they return false.
4. **boolean containsKey(Object key):** This method returns true if there is some key in the hash table that matches the given key. Otherwise, it returns false.

Enumeration <V> elements( ): This method returns an enumeration of values in the hash table.

V get(Object Key): This method returns the object that has the value corresponding to the given key. It returns a null object if the key is not found in the hash table.

boolean isEmtpy( ): This method returns true if the hash table is empty. Otherwise, it returns false.

enumeration <K> keys( ): This method returns an enumeration of the keys in the hash table.

V put(K key, V value): This method inserts the given key and value in the hash table. Then, it returns the previous value of the key if the key is already in the hash table. It returns a null object, if the key is not in the hash table.

void rehash( ): This method first increases the size of the hash table and then rehashes all the keys in the hash table.

v remove(Object key): This method deletes the given key and its corresponding value from the hash table, and returns the deleted value. It returns a null object, if the key is not in the hash table.

int Size( ): This method will return number of key/value pairs of hash table.

String toString( ): This method will return string representation of hash table.

gram

```
rt java.util.*;
ic class HashDemo

public static void main(String args[ ] )
{
    Hashtable ht = new Hashtable( );
    System.out.println("Before adding elements:" + ht.isEmpty( ));
    Integer i1 = new Integer(5);
    ht.put("hello", i1); //insert a key/value pair
    System.out.println("After adding elements:" + ht.isEmpty( ));
    ht.put("world", new Integer(10));
    ht.put("apple", new Double(20.5));
    ht.put("banana", new Boolean(true));
    ht.put("world", new Integer(10));
    HashDemo hd1= new HashDemo( );
    ht.put(hd1, i1);
    ht.put("sure", "abids");
    ht.put("date", "12th March, 20001");
    StringBuffer sb = new StringBuffer( );
    ht.put(sb, "Config Software Solutions");
    System.out.println("Before removing size:" + ht.size( ));
    ht.remove("apple");
    System.out.println("After removing size:" + ht.size( ));
    System.out.println("hello object in ht:" + ht.containsKey("hello"));
    System.out.println("i1 object in ht:" + ht.contains(i1));
    System.out.println("sb is associated with:" + ht.get(sb));
    Enumeration e = ht.keys( );
    while( e.hasMoreElements( ))
    {
        Object k = e.nextElement( );
        Object v = ht.get(k);
        System.out.println("Key:" + k + " Value:" + v);
    }
}
```

Output

```
Before adding elements : true
After adding elements : false
Before removing size : 8
After removing size : 7
hello object in ht : true
i1 object in ht : true
sb is associated with : Config Software Solutions
Key : Value : Config Software Solutions
Key : banana Value : True
Key : HashDemo@1db9742 Value : 5
Key : date Value : 12th March, 20001
Key : hello Value : 5
Key : sure Value : abids
Key : world Value : 10
```

Q32. What is the importance of hashCode( ) and equals( ) methods?

Answer : April/May-18(R16), Q8(b)

hashCode( )

The hashCode( ) method is used to generate hashcode for a specific object. It might either be a signature or checksum for the object. If the objects data is similar, according to the equals( ) method then objects should posses same hashcode. If the objects contain distinct data, then the hashcode values can be different. The hashcode values are used in storing the object in hash table. In hashtable, each object is assigned with an identifier (hashcode values) that helps in storing and determining the object location easily. By default, hashcode( ) method assigns each object with a unique number known as hash code. If a class does not override the hashCode( ) method then each object contains a unique hashcode.

equals( )

equals( ) method is used to compare the two objects. It determines the equality of two objects. By default, the equals( ) method compares the references of the given object. The two objects are said to be equal if they have same memory address.

Some of the conditions to be satisfied for implementing equals( ) and hashCode( ) methods are as follows,

(i) If two objects are equal by using equals( ) method then the both the objects should have same hashcode.

(ii) If two objects consists of same hashcode, then it is not mandatory that the objects to be equal.

Q33. Write about Properties of data structure.

Answer :

Properties

Properties is a subclass of Hashtable class. It has a list of values that represent keys and values as String objects. It has an instance variable **-defaults**, which holds a default list of each property object.

Constructors

Properties( ) – It creates a properties object without any default values.

Properties(Properties defaultValues) – It creates a properties object with the given default values.

Methods

Properties class inherits all methods of Hashtable. In addition to Hashtable methods, it defines the following methods.

1. **String getProperty(String key) String getProperty(String key, String defaultValue):** These methods return the value of the given key. However, the first method returns null if the given key is not present either in the default property list or in the list. Whereas, the second method returns the given default value if the key is not available in both the lists.
2. **void List (PrintStream outStream), void List(PrintWriter outStream):** These methods send the property list to the output stream that is connected to the outStream.
3. **void load(InputStream inStream), void load(Reader inStream):** These methods take a property list as input from the input stream that is connected to the inStream. These methods throw an IOException on error.
4. **void loadFromXML(InputStream inputStream):** This method takes a property list as input from an XML document that is connected to the inStream. This method throws IOException, and InvalidPropertiesFormatException on errors.
5. ***Enumeration <?> PropertyNames( ):*** This method returns an enumeration of the keys present in list and default property list.

Object setProeprty(String key, String value): This method associates the given value with the given key, and returns the previous value of the key, if the key is already present. However, it returns null if the key is not present.

void store(OutputStream outStream, String descp), void store(OutputStream outStream, String descp): These methods first write the given string description and then write the property list to the output stream that is connected to outStream. These methods throw an IOException on error.

void storeToXML(OutputStream OutputStream, String descp): This method first writes the given string description and then writes the property list to the XML document that is connected to the outStream. This method throws an IOException on error.

set<String> string PropertyNames( ): This method returns a set of keys.

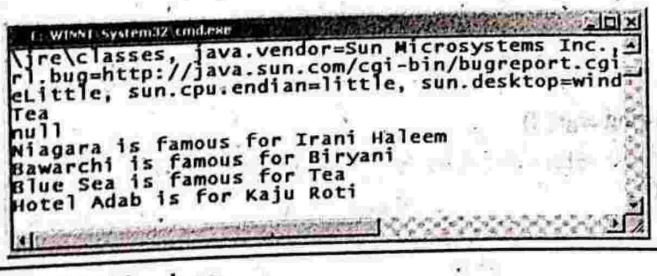
ample

Name: PropertiesDemo.java

```
xt java.util.*;
lic class PropertiesDemo

public static void main(String args[ ])
{
    Properties p = new Properties( );
    p.put("Bawarchi", "Biryani");
    p.put("Blue Sea", "Tea");
    p.put("Niagara", "Irani Haleem");
    System.out.println(p.getProperty("Blue Sea"));
    System.out.println(p.getProperty("Hotel Adab") );
    String str;
    Enumeration e = p.keys( );
    while(e.hasMoreElements( ))
    {
        str = (String) e.nextElement( );
        System.out.println( str + " is famous for" + p.getProperty(str));
    }
    String str1 = p.getProperty("Hotel Adab", "Kaju Roti");
    System.out.println("Hotel Adab is famous for" + str1);
}
```

tput



34. Explain in detail about Hashtable class.

nswer :

ashtable Class

Hashtables are very much similar to HashMap in which the pair(key,value) are stored in a Hash table. However, the only fference between these two collection classes is that, in HashMap, key and values can take a null value whereas in hashtable the ey and values can not be null. Whenever, a hashtable is being used, it is necessary for a programmer to specify an object which used as a key, and the value that must be linked with that key.

The declaration of generic hashtable is,

class hashtable<key,value>

A hashtable can store objects that have the capability of overriding hashcode( ) and equals( ). Methods defined by object class.

Constructors in Hashtable Class

1. **Hashtable( ):** It is a default constructor.
2. **Hashtable(int s):** This constructor creates a hashtable of size specified by the parameters. Basically, the default size of a hashtable is 11.
3. **Hahtable(int s, float fr):** This constructor creates a hashtable of size s and a fill ratio specified by fr. The range of fr is 0.0-1.1. The default fill ratio is 0.75.
4. **Hashtable(map<:extends key,? extends value>k):** This constructor creates a hashtable that is initialized with the elements specified in k.

Legacy Methods Defined by Hashtable

In addition to the methods defined by Map interface, hashtable defines the following legacy methods.

1. **void clear( ):** This method is used for resetting and deleting the hashtable values.
2. **object clone( ):** This method is used for returning duplicate value of the invoked object.
3. **boolean contains(Object value):** This method is used for returning boolean value. It returns true if the given value matches with the existing value in the hashtable, otherwise it returns false.
4. **enumeration<value>elements( ):** This method is used for returning numerical values present within the hashtable.
5. **void rehash( ):** This method is used for maximizing the hashtable size, and also for rehashing the corresponding keys.

Q35. Describe the concept of stack class. Also write an example program.

Answer : Model Paper-I, Q8(b)

Stack Class

Stack class is a derived class of Vector class. This class inherits all the methods defined by Vector class.

Syntax: class Stack<e1>

In the above statement, el is an object that type of element contained in a stack. The various methods used by stack are push( ), pop( ), peek( ) and search( ). This class implements the use of Last-in-first-out stack mechanism. The constructor of stack class.

Stack( )

The above constructor is a default constructor that creates an empty stack.

Methods of Stack Class

1. **empty( ):** This method returns true if the stack is empty.
2. **peek( ):** This method returns the element which is present at top of stack.
3. **push( ):** This method adds an item at the top of the stack.
4. **pop( ):** This method removes and returns the topmost element of stack.
5. **search( ):** This method searches the desired element from the stack and returns the number of pops operations to be made.

Program

```
import java.util.*;
public class StackEx
{
    public static void main(String args[ ])
    {
        Stack s=new Stack( );
        s.push(1);
        s.push(2);
        s.push(3);
        System.out.println("element of stack:"+s);
        //System.out.println("elements at the top of the stack"+s.push( ));
        System.out.println("elements removed from stack:"+s.pop( ));
    }
}
```

Output

```
element of stack : [1, 2, 3]
elements removed from stack : 3
```

Q36. How an Hashtable can change the iterator? Explain.

Answer : Nov./Dec.-17(R16), Q8(b)

Hashtable depends upon the implementation of map interface. It provides the optional map operations and even allows null values and null key. It will not assure whether the order remains constant. The iterators that are returned by this classes collection view methods" are fail fast. The iterator will throw a ConcurrentModificationException when the map is modified structurally after the iterator is created using the method other than its own remove or add methods. The iterator will fail quickly instead of going for arbitrary non deterministic behavior in future. This happens while it is modified concurrently.

The fail and fast behavior of the iterator cannot be assured in the presence of unsynchronized concurrent modification. The fail fast iterators will throw the concurrentModificationException. Such behavior of the iterators must be used only for detecting the bugs.

Q37. Explain about Vector class.

Answer :

Vector Class

Vector class is also a derived class of AbstractList class. This class has similar feature as that of ArrayList class. The two major differences between the vector class and ArrayList class is that the vector class are synchronized and it defines some specific type methods that are not present in Collection framework. The class declaration of Vector class is as follows,

class vector<E1>

In the above statement, the object E1 defines the type of element stored.

1. **Vector( ):** The constructor creates a default vector of size 10.
2. **Vector(int size):** This constructor creates a vector with the given size.
3. **Vector(collection co):** This constructor creates a vector that contains the elements of the specified by the collection co.
4. **Vector(int size,int increment):** This constructor creates a vector of given size. It takes two parameters namely size and increment. The parameter increment specifies the total number of elements that can be allocated to vector and size specifies the initial capacity of vector.

Methods of Vectors Class

1. **addElement(object EL):** This method is used to append the given element at the end of the vector. It also increments the size of the vector.
2. **Capacity( ):** This method is used to return the vector capacity.
3. **Contains(object EL):** This method returns true if the vector contains the specified object.
4. **ContainsAll(collections co):** This method returns true if the elements of the given collections are contained in the vector.
5. **elementsAt(int index):** This method returns the elements that are present at the specified index.
6. **Ensure capacity(int minimumcapacity):** This method is used to set the minimum capacity to the vector size.
7. **Get(int index):** This method is used to return the object present at the specified position in the vector.
8. **setElementsAt(object e, int index):** This method is used to replace the elements at the given index with the specified element.

Q38. What are similarities and differences between ArrayList and Vector? Explain. Nov./Dec.-18(R16), Q8(a)

OR

Differentiate between ArrayList and a Vector. Why ArrayList is faster than Vector? Explain.

(Refer Only Topic: Differences between ArrayList and Vector)

Nov./Dec.-17(R16), Q8(a)

Answer :

Similarities between ArrayList and Vector

Similarities between ArrayList and vector are as follows,

1. ArrayList and vector classes are considered as index based structures.
2. Both ArrayList and classes follows insertion order. So the elements can be displayed in the same order.
3. In both the classes, the iterator implementations are considered as fail-fast by design in java.
4. These classes can store null and duplicate values.
5. In arraylist and vector classes an element can be accessed randomly with the help of index number.

Differences between ArryList and Vector

	ArrayList		Vector
1.	ArrayList class will inherit the features of Abstract List class and List interface.	1.	Vector class will inherit the features of abstract list class.
2.	It is not synchronized.	2.	It is synchronized.
3.	It can increase the size of array upto 50% when number of elements are more than its capacity.	3.	It can increase the size of array upto 100% when number of elements are more than its capacity.
4.	It is not a legacy class.	4.	It is a legacy class.
5.	It is fast because it is not synchronized.	5.	It is slow because it is synchronized.
6.	It makes use of iterator interface for traversing the elements.	6.	It makes use of enumeration interface for traversing the elements. It event makes use of iterator.

The ArrayList is faster than vector because it is not synchronized and is added as a choice in single threaded access environment. It can be used in multithreaded environment when multiple threads are reading values from ArrayList. Vector is slow compared to ArrayList because it is synchronized. Vector contains other threads in runnable or non runnable state in multithreading environment until the current thread releases the lock of the object.

4.7 MORE UTILITY CLASSES – STRINGTOKENIZER, BITSET, DATE, CALENDAR, RANDOM, FORMATTER, SCANNER

Q39. Discuss about StringTokenizer class.

OR

Describe the important methods of StringTokenizer class.

(Refer Only Topic: Methods)

Answer :

May-19(R16), Q9(b)

StringTokenizer Class

StringTokenizer Class separates string/text into tokens delimited by \s,\t,\n,\r and \f. It takes a string as input and parses it into token. This method is known as parsing. StringTokenizer provides lexer/scanner since it is the initial step in the process of parsing. It provides the following constructors,

	Constructor	Description
1.	StringTokenizer(String s)	It takes the string to be tokenized as a parameter and considers white space as default delimiter.
2.	StringTokenizer(String s, String delimiters)	It takes the string to be tokenized as a parameter and also specifies delimiter.
3.	StringTokenizer(String s, String delimiters, boolean delimAsToken)	It takes the string to be tokenized as a parameter and also specifies a delimiter. In addition to it, it returns s tokens if the value of delimAstoken is true.

StringTokenizer class offers the following methods,

	Method	Description
1.	int countTokens( )	It is used to return the number of tokens available with StringTokenizer object.
2.	boolean hasmoreTokens( )	It is used to return true as long as more tokens exist with StringTokenizer object and false when no more tokens exist.
3.	boolean hasmoreElements( )	It is used to return true as long as more elements exist with StringTokenizer object and false when no more tokens exist.
4.	Object nextElement( )	It is used to returns the immediate token as object.
5.	String nextToken( )	It is used to returns the immediate token as string.
6.	String nextToken(string delimiters)	It is used to returns the immediate token as string and sets the delimiters string specified by the delimiters.

The class java.util.StringTokenizer is a simpler version of a class StringTokenizer(of java.io package), but it works only strings. The set of delimiters(the characters that separate tokens) may be specified at creation time.

Sometimes, it is necessary to decompose a full line of data into tokens as required by a processor and interpreter etc. Each token is a word. Delimiters are specified based on which the string is to be tokenized. The default delimiter is whitespace like " " which will be taken by default when any delimiter is omitted. Moreover, it is possible to specify group of delimiters.

gram

a program to illustrate the concept of StringTokenizer class

```
ort java.util.*;
ort java.util.StringTokenizer;
s class StringTokenizerClassDemo
{
    public static void main(String args[ ])
    {
        String str1 = "Hello World;How,Do You/Do,All Are/Okay;Thank/You";
        StringTokenizer st = new StringTokenizer(str1, ",;/");
        System.out.println("Number of tokens:" +st.countTokens( ));
        while(st.hasMoreTokens( ))
        {
            System.out.println(st.nextToken( ));
        }
    }
```

tput

```
Number of toakens : 8
Hello World
How
Do You
Do
All Are
Okay
Thank
You
```

The first parameter of the StringTokenizer constructor takes the string to be parsed and the second parameter is the list of delimiters. The list of delimiters used are whitespace, comma, semicolon and forward slash.

40. Explain about BitSet.

Model Paper-I, Q9(a)

Answer :

BitSet

A BitSet class creates an array that can store the bit values. It implements the Cloneable interface.

Constructors

BitSet( )

BitSet(int s)

The first constructor creates a default bit set. The second constructor creates a bit set of the size specified by 'S'. The size of an array can grow dynamically as needed. By default all bits are initialized to zero.

Methods

void AND(BitSet bs): This method performs the AND operation on the invoking BitSet object and the object specified by 'bs'. The result of AND operation is stored in the invoking object.

void OR(BitSet bs): This method performs the OR operation on the invoking BitSet object and the object specified by 'bs'. The result is stored in the invoking object.

3. **void XOR(BitSet bs):** This method performs the XOR operation on the invoking BitSet object and the object specified by 'bs'. The result is stored in the invoking object.
4. **void clear( ), void clear(int index):** The first form of clear( ) method assigns all bit values to zero. The second form of clear( ) method assigns the list value specified by index to zero.
5. **boolean get(int index), BitSet get(int start, int end):** The first form of get( ) method returns the bit value at the specified index. The second form returns the subset of BitSet specified by start and end index.
6. **void set(int index), void set(int index, boolean bit):** This method sets the value of a bit at the specified index. The boolean value 'bit' specifies the value to be set the bit can be either true or false.
7. **void set(int start, int end), void set(int start, int end, boolean bit):** This method sets the subset of bits from start to end–1. The boolean value 'bit' specifies the value to be set. It can be either true or false.
8. **String toString( ):** This method returns the equivalent string representation of the BitSet object.

Example

```
import java.util.BitSet;
class Bset
{
    public static void main(String args[ ])
    {
        BitSet bs1 = new BitSet(12);
        BitSet bs2 = new BitSet(12);
        for(int i=1; i <=12; i++)
        {
            if((i%6) != 0)
                bs1.set(i);
            if((i%4) == 0)
                bs2.set(i);
        }
        System.out.println("BS1 =" + bs1);
        System.out.println("BS2=" + bs2);
        bs2.and(bs1);
        System.out.println("ANDSet=" + bs2);
        bs2.or(bs1);
        System.out.println("ORSet =" + bs2);
        bs1.clear( );
        bs2.clear(2);
        bs2.clear(1);
        bs2.clear(9);
        bs2.xor(bs1);
        System.out.println("XORSet =" + bs2);
    }
}
```

Output

```
BS1 = {1, 2, 3, 4, 5, 7, 8, 9, 10, 11}
BS2 = {4, 8, 12}
ANDSet = {4, 8}
ORset = {1, 2, 3, 4, 5, 7, 8, 9, 10, 11}
XORSet = {3, 4, 5, 7, 8, 10, 11}
```

Q41. Write notes on Date class with an example.

Answer :

Date

The date class contains the current date and time.

Constructors

Date( )

Date(long mseconds)

The first form of constructor creates a default Date object and with the current date and time. Second constructor takes an argument that equals the number of milliseconds that have elapsed since midnight 1st January, 1970.

Methods

Some of the methods defined by Date class are moved into calendar and DateFormat classes and remaining methods are deprecated. The new methods added by Java to the Data class are as follows.

1. **long getTime( ):** This method is used to retrieve number of milliseconds that got elapsed from midnight 1st January, 1970.
2. **void setTime(long time):** This method is used to set the time.
3. **String toString( ):** This method returns the string equivalent to the invoking Date object.
4. **Object clone( ):** This method is used to create the duplicate of the invoking Date object.
5. **boolean equals(Object date):** This method is used to determine whether the two objects the invoking Date object and the object specified by date are equal. It returns true if they are equal, otherwise returns false.
6. **boolean before(Date date):** This method will return true if the invoking Date object contains a date less than the 'date' object. Otherwise it returns false.
7. **boolean after(Date date):** This method will return true if the invoking Date object contains a date greater than the 'date' object. Otherwise, it returns false.
8. **int hashCode( ):** This method is used to return the hash code for invoking object.
9. **int compareTo(Date date):** This method compares the invoking Date object with the specified by 'date' object. The value returned by this method are as follows.

Return value	Meaning
0	If both objects are equal.
Positive value	If the invoking objects comes after 'date'.
Negative value	If the invoking object comes before 'date'.

```
ogram
ort java.util.Date;
s DateTime

public static void main(String args[ ])
{
    long time;
    Date d = new Date( );
    System.out.println("The current date and time
                                    is:\n" + d);
    time = d.getTime( );
    System.out.println("Number of milliseconds since.
                                1/1/1970: \n" + time);
}
```

tput

The current date and time is :

Fir Nov 08 11:48:57 IST 2019

Number of milliseconds since. 1/1/1970:

1573193937349

mparison of Dates

Two date objects can be compared in three ways.

Find the time elapsed since January 1, 1970 using getTime( ) for both the objects and then compare their results.

Using the methods before( ), after( ) and equal ( ).

Using the compareTo( ) method. This method is defined by the comparable interface implemented by Date class.

ogram

```
ava program to demonstrate the Date class
port java.util.Date;
blic class DateDemo

public static void main(String args[])
{
    Date d = new Date( );
    System.out.println(d);
    //Fri Nov. 08 11.53.41 GMT+05.30
    System.out.println(d.getHours( )); //prints 11
    //to extract minutes
    System.out.println(d.getMinutes( )); //prints 53
    //to extract seconds
    System.out.println(d.getSeconds( ));
    //to extract hours
    System.out.println(d.getMonth( )); //prints 10 (for Nov)
    //to extract date
    System.out.println(d.getDate( )); //prints 8
    //to extract day
    System.out.println(d.getDay( )); //prints 5 (for Friday)
    //to extract year
    System.out.println(d.getYear());
    //to extract time
    System.out.println(d.getTime( ));
  }
}
```

Output

```
Fir Nov 08 11:53:41 IST 2019
11
53
41
10
8
5
119
1573194221974
```

Q42. Write about Calendar class.

Answer : Model Paper-I, Q9(b)

Calendar

The calendar is an abstract class. It provides a set of methods to manipulate the date and time. The subclasses of calendar class provides the specific implementation to the abstract methods defined by calendar to meet their own requirements. For example, GregorianCalendar is a subclass of calendar class.

The calendar class will not define its constructor. Thus an object of the abstract calendar class cannot created. The calendar class has several instance variables. These are declared as protected. They are,

- **fieldsSet:** It is a boolean variable that indicates whether the time component is set or not.
- **fields:** It is an array of integers that contains all the time components such as year, month, day, hour, time and second.
- **isSet:** It is a boolean array that indicates whether a specific time component is set or not.
- **time:** It is a long variable that contains the current time for this invoking object.
- **isTimeSet:** It is a boolean variable that indicates whether the current time is set or not.

The Calendar class also defines some integer constants used for setting or getting the components. These are as follows.

AM	MARCH	SUNDAY	DAY_OF_WEEK_IN_MONTH	DATE
PM	APRIL	MONDAY	WEEK_OF_MONTH	ALL_STYLES
AM_PM	MAY	TUESDAY	WEEK_OF_YEAR	LONG
HOUR	JUNE	WEDNESDAY	MONTH	SHORT
HOUR_OF_DAY	JULY	THURSDAY	YEAR	
MINUTE	AUGUST	FRIDAY	ERA	
SECOND	SEPTEMBER	SATURDAY	UNDECIMBER	
MILLISECOND	OCTOBER	DAY_OF_WEEK	DST_OFFSET	
JANUARY	NOVEMBER	DAY_OF_MONTH	FIELD_COUNT	
FEBRUARY	DECEMBER	DAY_OF_YEAR	ZONE_OFFSET	

The Calendar class defines several methods. Some commonly used methods are as follows,

1. **static Calendar getInstance( ):** This method returns the object of calendar class for the default location and time zone.
2. **int get(int const):** This method will return the value of the requested component of the invoking object. This component is specified by 'const'. It can be any one of the integer constants. For example Calendar.DATE, Calendar.HOUR etc.
3. **int get(int cal_Field):** This method is used to return the value of the given component of the invoking object. For example, Calendar.YEAR, Calendar.MINUTE, Calendar.HOUR and so on.
4. **void set(int const, int value):** This method will set the value of the date or time component as specified by const. The 'const' must be defined by calendar class. For example Calendar.HOUR.
5. **final boolean isSet(int cmpnt):** This method is used to return to when the provided component is set. Else, it returns false.
6. **final void set(int year, int month, int dayOfMonth):** This method is used to set the different components like date and time of the invoking object.
7. **final void set(int year, int month, int dayOfMonth, int hours, int minutes):** This method is used to set the different components (like year, month, dayOfMonth, hours, minutes) of the invoking object.
8. **final void setTime(Date d):** This method is used to set the different date and time components of the invoking object by using the given Date object d.
9. **void setTimeZone(TimeZone t):** This method is used to set the time zone of the invoking object provided by t.
10. **TimeZone getTimeZone( ):** This method is used to return the time zone for the invoking object.
11. **abstract void add(int cmpnt, int val):** This method is used to add or subtract the time or date provided by cmpnt.
12. **boolean equals(Object calObj):** This method will return true if both the invoking calendar object is similar to that of calObj contains. Otherwise it returns false.
13. **boolean after(Object calObj):** This method will return true if the invoking calendar object contains a date greater than 'calObj' object. Otherwise it returns false.
14. **boolean before(Object calObj):** This method will return true if the invoking calendar object contains a date less than 'calObj' object. Otherwise it returns false.
15. **final void clear( ):** This method is used to make the values of all the components in the invoking object to zero.
16. **final void clear(int cmpnt):** This method is used to make the values of the given component in the invoking object to zero.
17. **object clone( ):** This method is used to return the copy of invoking object.
18. **final Date getTime( ):** This method is used to return a Date object that is equivalent to the invoking object.
19. **static Locale( ) getAvailableLocales( ):** This method is used to return an array of Locale objects which consists of the locales which have the calendars.

Program

```
// a program to illustrate the Calendar class
import java.util.Calendar;
class DateAndTime

{
public static void main(String args[ ])
{
    Calendar cal = Calendar.getInstance( );
    System.out.print("Current date and time:");
    System.out.println(cal.getTime( ));
    cal.set(2007, 0, 25, 3, 35); //Set time to Jan.25, 2007 to 3 : 35
    System.out.println("Updated time:" + cal.getTime( ));
    cal.set(Calendar.HOUR, 10);
    cal.set(Calendar.MINUTE, 40);
    cal.set(Calendar.SECOND, 05);
    System.out.print("Updated time:");
    System.out.print(cal.get(Calendar.HOUR) + ":");
    System.out.print(cal.get(Calendar.MINUTE) + ":");
    System.out.println(cal.get(Calendar.SECOND));
    cal.set(cal.DATE, 20);
    System.out.println("Set date to 20:" + cal.getTime( ));
}
```

Output

```
Current date and time : Fri Nov 08 12:41:18 IST 2019
Updated time : Thu Jan 25 03:35:18 IST 2007
Updated time : 10:40:5
Set date to 20:Sat Jan 20 10:40:05 IST 2007
```

Q43. Explain the BitSet and Calendar classes in detail. Nov./Dec.-17(R16), Q9(a)

Answer :

BitSet Class

For answer refer Unit-IV, Q40.

Calender Class

For answer refer Unit-IV, Q42.

Q44. Write short notes on following collection framework classes.

(a) Random

(b) Scanner.

Answer :

a) **Random**

Random class generates a sequence of uniformly distributed number called pseudo random numbers.

Constructors of Random Class

- Random( ): This constructor uses current time as a starting value to create and generate number.
- Random(long s): This constructor generates a random number by taking the value specified by user as the starting value.

Methods of Random Class

1. **boolean nextBoolean( ) :** This method returns the next boolean random number.
2. **void nextBytes(byte value[ ]):** This method fills the values with the values that are randomly generated.
3. **double nextDouble( ):** This method returns the next double random number.
4. **float nextFloat( ):** This method returns the next float random number.
5. **double nextGaussian( ):** This method returns the next Gaussian random number.
6. **int nextInt( ):** This method returns the next int random number.
7. **int nextInt(int n):** This method returns the next int random number ranging from 0 to n.
8. **long nextLong( ):** This method returns the next long random numbers.
9. **void setSeed(long newseed):** This method sets the newseed specifies.

Example

```
import java.util.Random;
class Demo
{
    public static void main(String args[ ])
    {
        random ro = new random( );
        double value;
        double total=0;
        int arr [ ] = new int(10);
        for(int i=20;i<50;i+t)
        {
            value = ro.nextGaussian( );
            total+ = value;
            double x = –2;
            for(int j = 0;j<10;j++)
            x+ = 0.5;
            if(value<x)
            {
                arr[j]++;
                break;
            }
        }
        System.out.println("Average:"+(total/50));
        for(int i=0;i<10;i+1)
        {
            for(int j = arr[i]; j>0;j – –)
            System.out.println("*")
            System.out.println( );
        }
    }
```

Scanner

Scanner class is used for reading the formatted input and then converting it into the binary format. It is capable of reading types of data such as numeric values, strings, data etc. from keyboard, click file etc. It reads input from the console, or any source that implements readable interface. It is a member of java util package and hence must be imported to the class wherever required. The import statements "import java.util.*;" or "import java.util.scanner;" must be used for this purpose.

Generally, the steps to use a Scanner class are as follows,

Check whether any particular type of input data is available or not by using the hasNextX method. Here, X represents the type of data required.

In case of availability of input, call any one of the nextX methods of Scanner class and read it.

Repeat the same procedure till the complete input is read.

Scanner class provides various constructors that take data source as parameters. Following are some of them.

	Constructor		Description
1.	Scanner (File source)	1.	Constructs a new scanner that generates values scanned from a specified file.
2.	Scanner (InputStream source)	2.	Constructs a new scanner that generates values scanned from the specified input stream.
3.	Scanner (String source)	3.	Constructs a new scanner that generates values scanned from the specified string.

In addition, scanner provides various methods. Some of the methods of this class are as follows,

boolean hasNext( ): This method returns true if a token is ready to be read, otherwise it returns false.

string next( ): This method returns the next token of any type as input source.

boolean nextBoolean( ): This method returns the next token boolean value.

byte nextByte( ): This method returns the next token as byte value.

float nextFloat( ): This method returns the next token as float value.

void close( ): This method is used to close the scanner.

boolean hasNext(Pattern P): This method returns true if the next token matches with the particular or required pattern P.

boolean has Next(String P): This method returns true if the next token matches with the pattern in the particular string.

boolean hasNextBoolean( ): This method returns true if the next token present in the input can be rewritten as a boolean value.

boolean has NextByte( ): This method returns true only if the next token present in the input can be rewritten as a byte value.

boolean hasNextDouble( ): This method returns true only if the next token present in the input can be rewritten as a double value.

boolean hasNextFloat( ): This method returns true only if the next token present in the input can be rewritten as a float value.

boolean hasNextInt( ): This method returns true if the next token present in the input can be rewritten as a integer value.

boolean hasNextLine( ): This method returns true if any other line is obtained in the input.

boolean hasNextLong( ): This method returns true if the next token present in the input can be rewritten as a long value.

boolean hasNextShort( ): This method returns true if the next token present in the input can be rewritten as a short value.

string next(Pattern pattern): This method is used to return the next token if it matches with the required pattern.

double nextDouble( ): This method is used to return the next token double value.

short nextInt( ): This method is used to return the next token integer value.

short nextLine( ): This method is used to return the input in the form of string or scanner parsers the present line.

short nextShort( ): This method is used to return the next token short value.

short nextLong( ): This method is used to return the next token long value.

scanner UseDelimiter(String pattern): This method is used to set the delimiting pattern for scanning it in the required string.

string nextLine( ): This method is used to return a string as the next line of input.

long nextLong( ): This method is used to return a long value as the next token. It uses the default radix value i.e., 10.

BigDecimal nextBigDecimal( ): This method is used to return a BigDecimal object as the next token.

BigInteger nextBigInteger( ): This method is used to return a BigInteger object as the next token. It uses the default radix value i.e., 10.

Program

```
import java.util.*;
import java.util.Scanner;
class ScannerEx
{
        public static void main(String args[])
        {
              Scanner s=new Scanner(System.in);
              System.out.print("Enter ID:");
              int i=s.nextInt();
              System.out.print("Enter Name:");
              String n=s.next();
              System.out.print("Enter Department:");
              String d=s.next();
              System.out.println("The entered details are:");
              System.out.println("\nID is:" + i + "\nName is:" + n + "\nDepartment is:" + d);
              s.close();
        }
}
```

Output

```
Enter ID:100
Enter Name : Nikhil
Enter Dpartment:CSE
The entered details are:
ID is : 100
Name is : Nikhil
Department is : CSE
```

Q45. Explain in detail about formatter.

Answer :

Formatter

Formatter class provides the format conversions to display the numbers, time and date, strings in the required format. The functioning of formatter class is similar to the C/C++ printf( ) function with a small differences and some additional features.

Formatter Constructors

To format the output, user must define a formatter object. The formatter will be operated by converting the binary data used by a program to the formatted data. The formatted data is stored in the buffer, and this can be accessed by the programs in accordance with their requirement.

The formatter constructors defined by the formatter class are as follows,

Constructor	Description
Formatter( )	It constructs a new formatter object.
Formatter(Appendable buff)	It constructs a formatter with certain destination.
Formatter(Appendable buff, locale locale)	It constructs a formatter with a specified destination and locale.
Formatter(String Filename) throws FileNotFoundException	It constructs a formatter with a file. It throws an exception "FileNotFoundException" if the file is unavailable.
Formatter(String Filename, String charset) throws File-NotFoundException, unsupported encoding Exception	It constructs a formatter with a file and charset.
Formatter(File outF) throws FileNotFoundException	It constructs a formatter with a specified output file, say outF. It throws an exception "FileNotFound" if file is unavailable.
Formatter(File outstr)	It constructs a formatter with a specified output stream that will receive the output.

matter Methods

The formatter class defines various formatter methods and they are as follows,

Method	Description
id close( )	It is used to close the invoking formatter with this resources used by the object will get released. An attempt to access the closed formatter leads to the exception i.e., "FormatterClosedException".
id flush( )	It is used to flush the format buffer. When this method is invoked, the data stored in the buffer will be written to the destination.
rmatter format(string fmtstring, object.args)	It formats the arguments and returns the invoking objects.
rmatter format(local loc, string fmtstring, object...args)	It formats arguments and returns invoking object. It uses the specified loc for the format.
Exception ioException( )	It returns this Exception when the invoking object for output throws IoException otherwise, it will returns null.
cale locale( )	It returns the locale of the invoking object.
pendable out( )	It returns the reference of destination for output.
ng tostring( )	It returns a string to hold the formatted output.

Once the formatter is created, it can be used to create formatted string. It can be represented as:

Formatter format(String fmtstring, object....arg)

The format specifiers can be represented with a percent sign followed by the conversion specifier.

The various format specifiers are as follows,

Format Specifier	Conversion
%a	Floating point hexadecimal
%A	Floating point hexadecimal
%b	Boolean
%B	Boolean
%c	Character
%d	Decimal integer
%h	Hash code of argument
%H	Hash code of argument
%e	Scientific notation
%E	Scientific notation
%f	Decimal floating point
%g	Makes use of %e or %f
%G	Makes use of %e or %f
%o	Octal integer
%n	New line character
%s	String
%S	String
%t	Time and date
%T	Time and date
%x	Integer hexadecimal
%X	Integer hexadecimal
%%	will insert a % sign

A sample example to illustrate the creation of a formatted string and to display it is as follows,

```
import java.util.*;
class format
{
    public static void main(String args[ ])
    {
    Formatter F = new Formatter( );          //object creation
    F.format("Java%s basics %d %f", "Formatting", 5, 2.5);
    System.out.println(F);
    F.close( );
    }
}
```

Output

Java Formatting basic 5 2.5.

The output can also be obtained by calling out( ). It returns a reference to an Appendable object.

Formatting Strings and Characters

A character is formatted by using %c and a string is formatted by using %s.

Example

```
f.format("%c %s", "a", "specifier");
```

Formatting Numbers

An integer value/number in decimal format is formatted by using %d. A floating point value in decimal format and scientific notation is formatted by using %f and %e respectively.

The representation of number in scientific notation is given by,

x.ddde+/-y

The %g format specifier specifies the formatter when it use %f and %e.

Example

```
f.format("%f%d%e", 1.23, 12, 23.45 e + 67);
```

The specifiers %o and %x are used to display the integer value in octal and hexadecimal format respectively. And %a is used to display the floating point value in hexadecimal format,

Example

```
f.format("%x %o %a", 123, 456, 712.00);
```

Formatting Time and Date

The specifiers %t is used for formatting time and date. The suffixes are used with %t that determines the desired format of time and date.

Example

```
f.format("%tM", calender);
```

%tM is used to display the minutes in a two-character field.

The %n and %% Format Specifiers

The specifier %n is used for inserting a new line and %% is used for inserting percent sign.

Example

f.format("File is copied upto %n %d %%", 75);

Specifying a Minimum Field Width

Minimum field-width specifier is an integer that is used in between % sign and format conversion code. This is shown in the output screen where the result with spaces is displayed.

Example

f.format("%f %n %012f %n %0123f", 10.123, 10.456, 10.789);

The output of above example is shown below,

```
10.123
    10.456
        10.789
```

In the output, the first line is printed with the given value, the second line and third line prints the value with the space provided in the command.

Specifying Precision

The format specifiers %f, %e, %g and %s can be applied with specifier. It contains a period along with an integer. It is applied on the basis of the type of data. If the precision specifier is applied to floating-point values with %f or %e then the number of decimal places are determined. The default value of precision is 6. When %g is used then the precision determines the count of significant bits.

Example

f.format ("%5.2f", 9876.43215012);

The value of %5.2f represents the number that is atleast five characters wide with two decimal places.

Format Flags

The formatter identifies the collection group of format flags that are used for different types of conversion. The format flags are individual characters that are represented with the % in the format specification.

The following table represents the format flags.

Format flag	Description
–	This flag is used for left justification.
#	This flag is used for the alternate conversion format.
0	This flag is used to display the output with zeros instead of spaces.
Space	This flag allows a positive numeric output to be preceded with a space.
+	This flag allows a positive numeric output to be preceded by a + sign.
,	This flag allows a numeric values to include grouping separators.
(	This flag is used for enclosing the negative numeric values.

Justifying Output

The output of any program is right-justified by default. When the data printed is smaller than the field width then the data is moved to the right edge of field. The output can be set left- justified by using a minus sign placing it after the % symbol.

Example

```
f.format("1%10.2f", 180.01);
```

The above statement prints the output on the right edge of field i.e right-justified with the floating-point number with two decimal places in a 10-character field.

Space, +, 0, and (Flags

(a) + Flag

The + Flag is used to place + sign before any positive numeric value.

Example

```
f.format("%+d", 250);
```

(b) Space Flag

Space flag is used to display the output that has spaces in between the numbers. While creating a column with numbers, the space flag is placed before positive values such that the negative and positive are lined up differently.

Example

```
f.format("%d", –258);
f.format("%d",285);
```

(c) (Flag

(Flag is used to display negative value with the inside parenthesis that is given by a '–' sign.

Example

```
f.format("%(d", – 280);
```

The output for the above statement will be (280)

(d) 0 Flag

0 flag is used to display the output with zeros instead of spaces.

(e) Comma Flag

Comma flag (,) is used to display the larger number. The grouping separators are added to the number.

Example

```
f.format("%, .2f", 9876543210.12);
```

The output will be

9, 876, 543, 210.12

(f) # Flag

flag is used with the %f, %e, %x and %0 format specifiers. When # is used with %f and %e then it makes sure that the number is a decimal point even though it does not have decimal digits. When # is used with %x format specifier then the output will be printed with a hexadecimal number that has 0x prefix. When # is used with %0 format specifier then the output will be printed with a number that has zeros.

Uppercase Option

Certain format specifiers are required to be in uppercase. They are as follows,

Uppercase Format Specifier	Description
%A	This is used to uppercase the hexadecimal digits a to f as A to F. The prefix OX is converted to OX and p as P.
%B	This is used to uppercase the true and false values.
%E	This is used to display the exponent symbol e in uppercase.
%G	This is used to display the exponent symbol g in uppercase.
%H	This is used to uppercase the hexadecimal digits a to f as A to F.
%S	This is used to uppercase the given string.
%T	This is used to uppercase the output which is in alphabetical sequence.
%X	This is used to uppercase the hexadecimal digits a to f as A to F. The optional prefix Ox is converted to OX, if it is present.

Example

```
F.format("%X", 120);
F.format("%E", 234.456);
```

Argument Index

A formatter allows an argument to be specified with the format specifier. In this, the first format specifier is matched with the first argument and the second format specifier is matched with the second argument and so on. Therefore, the format specifiers are matched with the arguments in a left to right order. An argument index is used explicitly to specify which format specifier is matched with an argument. An argument index is given by $ symbol that is followed by the % symbol in a given format specifier.

Example

```
F.format("%4$d %2$d %6$d" 60, 40, 20);
```

An argument that is matched with the preceding format specifier can be used again and is referred to relative index. Instead of $ symbol, < is used before the format specifier.

Example

```
F.format("%d in hex is % <x", 123);
```

Closing a Formatter

A formatter is closed by explicitly calling the close( ). A formatter can be closed automatically by using an auto closeable interface. The try-with-resources statement are supported by this interface.

Syntax for Closing a Formatter

```
f.close( );
```

example for closing a formatter automatically is as follows,

```
import java.util.*;
class format
{
public static void main(String args[ ])
{
try(Formatter f = new Formatter( ))
{
f.format("Hello %s is printed in the output program %d", "world", 5);
System.out.println(f);
}
}
}
```

Printf( ) Connection

A formatted string is created automatically by using a formatter in printf( ) method. The resultant string is displayed on System.out which is the default console. The printf( ) method is used by printStream and printWriter.