CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test 3– Mar. 2024

Sub:	Advanced Java& J2EE							Sub Code:	22MCA341
Date:	13/3/2024	Duration:	90 min's	Max Marks:	50	Sem:	III	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

		MARKS	OBE	
			CO	RBT
	PART I			
1	Explain exception and methods that are provided in collection interface.	**10**	CO1	L2
	OR			
2.	Create a class STUDENT with 2 private string members : USN,NAME using LinkedList class in java. Write a program to add atleast 3 objects of above STUDENT class. Also display the data in neat format.	**10**	CO2	L4
	PART II			
3	Explain the following methods of StringBuffer class. i)append() ii)insert() iii)reverse() iv)replace()	**10**	CO1	L2
	OR			
4.	Explain the constructors of TreeSet class and write a java program to create TreeSet collection and access via iterator.	**10**	CO1,CO2	L2

CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test 3– Mar. 2024

Sub:	Advanced Java& J2EE							Sub Code:	22MCA341
Date:	13/3/2024	Duration:	90 min's	Max Marks:	50	Sem:	III	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

		MARKS	OBE	
			CO	RBT
	PART I			
1	Explain exception and methods that are provided in collection interface.	**10**	CO1	L2
	OR			
2.	Create a class STUDENT with 2 private string members : USN,NAME using LinkedList class in java. Write a program to add atleast 3 objects of above STUDENT class. Also display the data in neat format.	**10**	CO2	L4
	PART II			
3	Explain the following methods of StringBuffer class. i)append() ii)insert() iii)reverse() iv)replace()	**10**	CO1	L2
	OR			
4.	Explain the constructors of TreeSet class and write a java program to create TreeSet collection and access via iterator.	**10**	CO1,CO2	L2

5	**PART III** Explain the following legacy classes with an example. i)HashTable ii)Vector	**10**	CO1	L2
	OR			
6	Write a program to check if two accepted strings are the rotation of each other. A string is said to be a rotation of other string if they contain same characters and the sequence is rotated across any character. Example “dabc” is a rotation of “abcd” but “dbac” is not.	**10**	CO2	L4
	PART IV			
7	Define String Handling? Explain Special string operations with example.	**10**	CO1	L2
	OR			
8	Explain the following collection interfaces. i) Queue ii)SortedSet	**10**	CO1	L2
	PART V			
9	Explain ArrayList class and explain following methods: i)binarysearch ii)copyOf() iii)equals() iv)fill	**10**	CO1	L2
	OR			
10	Write a program to find the first repeated and non-repeated character in a given string.	**10**	CO2	L4

5	**PART III** Explain the following legacy classes with an example. i)HashTable ii)Vector	**10**	CO1	L2
	OR			
6	Write a program to check if two accepted strings are the rotation of each other. A string is said to be a rotation of other string if they contain same characters and the sequence is rotated across any character. Example “dabc” is a rotation of “abcd” but “dbac” is not.	**10**	CO2	L4
	PART IV			
7	Define String Handling? Explain Special string operations with example.	**10**	CO1	L2
	OR			
8	Explain the following collection interfaces. i) Queue ii)SortedSet	**10**	CO1	L2
	PART V			
9	Explain ArrayList class and explain following methods: i)binarysearch ii)copyOf() iii)equals() iv)fill	**10**	CO1	L2
	OR			
10	Write a program to find the first repeated and non-repeated character in a given string.	**10**	CO2	L4

1. **Explain exception and methods that are provided in collection interface.**

The methods defined by collection are

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

TABLE 17-1 The Methods Defined by **Collection**

In Java, the Collections interface doesn't directly raise exceptions itself since it's an interface and doesn't provide implementation. However, the classes that implement the Collections interface (like List, Set, Map, etc.) can raise various exceptions depending on the operation being performed. Some common exceptions raised during collection operations include:

1. NullPointerException: This is commonly thrown when attempting to perform an operation on a null object reference, such as adding a null element to a collection that doesn't support null elements.

2. ClassCastException: This exception is thrown when attempting to add an element of an incompatible type to a collection. For example, trying to add a String to a collection of Integers.

3. IllegalArgumentException: This exception can be thrown for various reasons, such as trying to add an element to a collection that violates its constraints or passing an illegal argument to a method.

4. UnsupportedOperationException: This exception indicates that an unsupported operation was attempted on a collection. For example, trying to modify an immutable collection or removing an element from an unmodifiable collection.

5. **ConcurrentModificationException**: This exception occurs when a collection is modified while it is being iterated over by a different thread. This can happen, for example, if elements are added or removed from a collection while it is being iterated over using an iterator.

These are some of the common exceptions raised during collection operations in Java. The specific exceptions raised may vary depending on the implementation and usage context. It's important to handle these exceptions appropriately in our code to ensure robustness and reliability.

2. Create a class STUDENT with 2 private string members : USN,NAME using LinkedList class in java. Write a program to add atleast 3 objects of aboveSTUDENT class. Also display the data in neat format.

```
import java.util.LinkedList;
import java.util.Iterator;

class Student {
    private String USN;
    private String name;

    public Student(String USN, String name) {
        this.USN = USN;
        this.name = name;
    }

    public String getUSN() {
        return USN;
    }

    public String getName() {
        return name;
    }
}

public class Main {
    public static void main(String[] args) {
        LinkedList<Student> students = new LinkedList<>();

        students.add(new Student("1", "Alice"));
        students.add(new Student("2", "Bob"));
        students.add(new Student("3", "Charlie"));

        System.out.println("Student Details:");
        System.out.println("----------------");
        Iterator<Student> iterator = students.iterator();
        while (iterator.hasNext()) {
            Student student = iterator.next();
            System.out.println("USN: " + student.getUSN() + ", Name: " + student.getName());
        }
    }
}
```

2. Explain the following methods of StringBuffer class.i)append() ii)insert() iii)reverse() iv)replace()

i)append()

The append( ) method concatenates the string representation of any other type of data to the
end of the invoking StringBuffer object. It has several overloaded versions. Here are a few

of its forms:
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
String.valueOf( ) is called for each parameter to obtain its string representation. The result is appended to the current StringBuffer object. The buffer itself is returned by each
version of append( ). This allows subsequent calls to be chained together, as shown in the
following example:

```
// Demonstrate append().
class appendDemo {
public static void main(String args[]) {
String s;
int a = 42;
StringBuffer sb = new StringBuffer(40);
s = sb.append("a = ").append(a).append("!").toString();
System.out.println(s);
}
}
```

ii)insert()

The insert( ) method inserts one string into another. It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences. Like append( ), it calls String.valueOf( ) to obtain the string representation of the value it is called with. This string is then inserted into the invoking StringBuffer object. These are a few of its forms:

StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)

Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object. The following sample program inserts “like” between “I” and “Java”:

```
// Demonstrate insert().
class insertDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "like ");
System.out.println(sb);
}
}
```

iii)reverse()

You can reverse the characters within a StringBuffer object using reverse( ), shown here:

StringBuffer reverse( )

This method returns the reversed object on which it was called.
The following program demonstrates reverse( ):

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
public static void main(String args[]) {
StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
s.reverse();
System.out.println(s);
}
}
```

iv)replace()

You can replace one set of characters with another set inside a StringBuffer object by calling
replace( ). Its signature is shown here:
StringBuffer replace(int startIndex, int endIndex, String str)
The substring being replaced is specified by the indexes startIndex and endIndex. Thus, the
substring at startIndex through endIndex1 is replaced. The replacement string is passed in – str.
The resulting StringBuffer object is returned.
The following program demonstrates replace( ):

```
// Demonstrate replace()
class replaceDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was");
System.out.println("After replace: " + sb); } }
```

4. Explain the constructors of TreeSet class and write a java program to create TreeSet collection and access via iterator.

TreeSet extends AbstractSet and implements the NavigableSet interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.
TreeSet is a generic class that has this declaration:
class TreeSet<E>
Here, E specifies the type of objects that the set will hold.
TreeSet has the following constructors:
TreeSet( )
TreeSet(Collection<? extends E> c)
TreeSet(Comparator<? super E> comp)
TreeSet(SortedSet<E> ss)
The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements. The second form builds a tree set that contains the elements of c. The third form constructs an empty tree set that will be sorted according to the comparator specified by comp. (Comparators are described later in this chapter.) The fourth form builds a tree set that contains the elements of ss

```
import java.util.TreeSet;
import java.util.Iterator;

public class Main {
   public static void main(String[] args) {
      TreeSet<String> treeSet = new TreeSet<>();

      // Adding elements to the TreeSet
      treeSet.add("Apple");
      treeSet.add("Banana");
      treeSet.add("Orange");
      treeSet.add("Grapes");

      // Accessing elements using iterator
      System.out.println("Elements in TreeSet:");
      System.out.println("--------------------");
      Iterator<String> iterator = treeSet.iterator();
      while (iterator.hasNext()) {
         String element = iterator.next();
```

```
            System.out.println(element);
        }
    }
}
```

5. Explain the following legacy classes with an example. i)HashTable ii)Vector

Hashtable stores key/value pairs in a hash table. However, neither keys nor values can be null. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Hashtable was made generic by JDK 5. It is declared like this:

class Hashtable<K, V>

Here, K specifies the type of keys, and V specifies the type of values.

A hash table can only store objects that override the hashCode( ) and equals( ) methods that are defined by Object. The hashCode( ) method must compute and return the hash code for the object. Of course, equals( ) compares two objects. Fortunately, many of Java's built-in classes already implement the hashCode( ) method. For example, the most common type of Hashtable uses a String object as the key. String implements both hashCode( ) and equals( ).

The Hashtable constructors are shown here:

Hashtable( )

Hashtable(int size)

Hashtable(int size, float fillRatio)

Hashtable(Map<? extends K, ? extends V> m)

Method	Description
void clear( )	Resets and empties the hash table.
Object clone( )	Returns a duplicate of the invoking object.
boolean contains(Object *value*)	Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found.
boolean containsKey(Object *key*)	Returns **true** if some key equal to *key* exists within the hash table. Returns **false** if the key isn't found.
boolean containsValue(Object *value*)	Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found.
Enumeration<V> elements( )	Returns an enumeration of the values contained in the hash table.
V get(Object *key*)	Returns the object that contains the value associated with *key*. If *key* is not in the hash table, a **null** object is returned.
boolean isEmpty( )	Returns **true** if the hash table is empty; returns **false** if it contains at least one key.
Enumeration<K> keys( )	Returns an enumeration of the keys contained in the hash table.
V put(K *key*, V *value*)	Inserts a key and a value into the hash table. Returns **null** if *key* isn't already in the hash table; returns the previous value associated with *key* if *key* is already in the hash table.
void rehash( )	Increases the size of the hash table and rehashes all of its keys.
V remove(Object *key*)	Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the hash table, a **null** object is returned.
int size( )	Returns the number of entries in the hash table.
String toString( )	Returns the string equivalent of a hash table.

TABLE 17-18 The Legacy Methods Defined by **Hashtable**

```
import java.util.*;
class HTDemo2 {
public static void main(String args[]) {
Hashtable<String, Double> balance = new Hashtable<String, Double>();
String str;
double bal;
balance.put("John Doe", 3434.34);
balance.put("Tom Smith", 123.22);
balance.put("Jane Baker", 1378.00);
balance.put("Tod Hall", 99.22);
balance.put("Ralph Smith", -19.08);
// Show all balances in hashtable.
// First, get a set view of the keys.
Set<String> set = balance.keySet();
// Get an iterator.
Iterator<String> itr = set.iterator();
while(itr.hasNext()) {
str = itr.next();
System.out.println(str + ": " +
balance.get(str));
}
System.out.println();
// Deposit 1,000 into John Doe's account.
bal = balance.get("John Doe");
balance.put("John Doe", bal+1000);
System.out.println("John Doe's new balance: " +
balance.get("John Doe"));
}
}
```

ii)Vector

Vector implements a dynamic array. It is similar to ArrayList, but with two differences: Vector is synchronized, and it contains many legacy methods that are not part of the Collections Framework. With the advent of collections, Vector was reengineered to extend AbstractList and to implement the List interface. With the release of JDK 5, it was retrofitted for generics and reengineered to implement Iterable. This means that Vector is fully compatible with collections,
and a Vector can have its contents iterated by the enhanced for loop.

Vector is declared like this:

```
class Vector<E>
```

Here, E specifies the type of element that will be stored.

Here are the Vector constructors:

```
Vector( )
Vector(int size)
Vector(int size, int incr)
Vector(Collection<? extends E> c)
```

Method	Description
void addElement(E *element*)	The object specified by *element* is added to the vector.
int capacity( )	Returns the capacity of the vector.
Object clone( )	Returns a duplicate of the invoking vector.
boolean contains(Object *element*)	Returns **true** if *element* is contained by the vector, and returns **false** if it is not.
void copyInto(Object *array*[ ])	The elements contained in the invoking vector are copied into the array specified by *array*.
E elementAt(int *index*)	Returns the element at the location specified by *index*.
Enumeration<E> elements( )	Returns an enumeration of the elements in the vector.
void ensureCapacity(int *size*)	Sets the minimum capacity of the vector to *size*.
E firstElement( )	Returns the first element in the vector.
int indexOf(Object *element*)	Returns the index of the first occurrence of *element*. If the object is not in the vector, –1 is returned.
int indexOf(Object *element*, int *start*)	Returns the index of the first occurrence of *element* at or after *start*. If the object is not in that portion of the vector, –1 is returned.
void insertElementAt(E *element*, int *index*)	Adds *element* to the vector at the location specified by *index*.
boolean isEmpty( )	Returns **true** if the vector is empty, and returns **false** if it contains one or more elements.
E lastElement( )	Returns the last element in the vector.
int lastIndexOf(Object *element*)	Returns the index of the last occurrence of *element*. If the object is not in the vector, –1 is returned.
int lastIndexOf(Object *element*, int *start*)	Returns the index of the last occurrence of *element* before *start*. If the object is not in that portion of the vector, –1 is returned.
void removeAllElements( )	Empties the vector. After this method executes, the size of the vector is zero.
boolean removeElement(Object *element*)	Removes *element* from the vector. If more than one instance of the specified object exists in the vector, then it is the first one that is removed. Returns **true** if successful and **false** if the object is not found.
void removeElementAt(int *index*)	Removes the element at the location specified by *index*.
void setElementAt(E *element*, int *index*)	The location specified by *index* is assigned *element*.
void setSize(int *size*)	Sets the number of elements in the vector to *size*. If the new size is less than the old size, elements are lost. If the new size is larger than the old size, **null** elements are added.
int size( )	Returns the number of elements currently in the vector.
String toString( )	Returns the string equivalent of the vector.
void trimToSize( )	Sets the vector's capacity equal to the number of elements that it currently holds.

TABLE 17-15 The Legacy Methods Defined by **Vector**

```
// Demonstrate various Vector operations.
import java.util.*;
class VectorDemo {
public static void main(String args[]) {
// initial size is 3, increment is 2
Vector<Integer> v = new Vector<Integer>(3, 2);
System.out.println("Initial size: " + v.size());
System.out.println("Initial capacity: " +
v.capacity());
v.addElement(1);
v.addElement(2);
v.addElement(3);
v.addElement(4);
System.out.println("Capacity after four additions: " +
v.capacity());
v.addElement(5);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(6);
v.addElement(7);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(9);
v.addElement(10);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(11);
v.addElement(12);
System.out.println("First element: " + v.firstElement());
System.out.println("Last element: " + v.lastElement());
```

```
if(v.contains(3))
System.out.println("Vector contains 3.");
// Enumerate the elements in the vector.
Enumeration vEnum = v.elements();
System.out.println("\nElements in vector:");
while(vEnum.hasMoreElements())
System.out.print(vEnum.nextElement() + " ");
System.out.println();
}
}
```

6. Write a program to check if two accepted strings are the rotation of each other. A string is said to be a rotation of other string if they contain same characters and thesequence is rotated across any character. Example “dabc” is a rotation of “abcd” but “dbac” is not.

```
// Java program to check if two given strings are
// rotations of each other

class StringRotation {

	/* Function checks if passed strings (str1 and str2)
	are rotations of each other */
	static boolean areRotations(String str1, String str2)
	{
		// There lengths must be same and str2 must be
		// a substring of str1 concatenated with str1.
		return (str1.length() == str2.length()) &&
			((str1 + str1).contains(str2));
	}

	// Driver method
	public static void main(String[] args)
	{
		String str1 = "AACD";
		String str2 = "ACDA";

		if (areRotations(str1, str2))
			System.out.println("Yes");
		else
			System.out.printf("No");
	}
}
```

7. Define String Handling? Explain Special string operations with example.

Method	Description
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*. Added by J2SE 5.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*. Added by J2SE 5.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1. Added by J2SE 5.
boolean contains(CharSequence *str*)	Returns **true** if the invoking object contains the string specified by *str*. Returns **false**, otherwise. Added by J2SE 5.
boolean contentEquals(CharSequence *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**. Added by J2SE 5.
boolean contentEquals(StringBuffer *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
static String format(String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. (See Chapter 18 for details on formatting.) Added by J2SE 5.
static String format(Locale *loc*, String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. Formatting is governed by the locale specified by *loc*. (See Chapter 18 for details on formatting.) Added by J2SE 5.
boolean matches(string *regExp*)	Returns **true** if the invoking string matches the regular expression passed in *regExp*. Otherwise, returns **false**.
int offsetByCodePoints(int *start*, int *num*)	Returns the index with the invoking string that is *num* code points beyond the starting index specified by *start*. Added by J2SE 5.
String replaceFirst(String *regExp*, String *newStr*)	Returns a string in which the first substring that matches the regular expression specified by *regExp* is replaced by *newStr*.
String replaceAll(String *regExp*, String *newStr*)	Returns a string in which all substrings that match the regular expression specified by *regExp* are replaced by *newStr*.

Method	Description
String[ ] split(String *regExp*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*.
String[ ] split(String *regExp*, int *max*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*. The number of pieces is specified by *max*. If *max* is negative, then the invoking string is fully decomposed. Otherwise, if *max* contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If *max* is zero, the invoking string is fully decomposed.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is now implemented by **String**.

8. Explain the following collection interfaces.
i) Queue ii)SortedSet
i)Queue

ii)The SortedSet Interface

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order. SortedSet is a generic interface that has this declaration:

interface SortedSet<E>

Here, E specifies the type of objects that the set will hold.

In addition to those methods defined by Set, the SortedSet interface declares the methods summarized in Table 17-3. Several methods throw a NoSuchElementException when no items are contained in the invoking set. A ClassCastException is thrown when an object is incompatible with the elements in a set. A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set. An IllegalArgumentException is thrown if an invalid argument is used.

SortedSet defines several methods that make set processing more convenient. To obtain

the first object in the set, call first( ). To get the last element, use last( ). You can obtain a subset of a sorted set by calling subSet( ), specifying the first and last object in the set. If you need the subset that starts with the first element in the set, use headSet( ). If you want the subset that ends the set, use tailSet( ).

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

TABLE 17-3 The Methods Defined by **SortedSet**

ii)The Queue Interface

The Queue interface extends Collection and declares the behavior of a queue, which is often a first-in, first-out list. However, there are types of queues in which the ordering is based upon other criteria. Queue is a generic interface that has this declaration: interface Queue<E>

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

TABLE 17-5 The Methods Defined by **Queue**

Several methods throw a ClassCastException when an object is incompatible with the elements in the queue. A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the queue. An IllegalArgumentException is thrown if an invalid argument is used. An IllegalStateException is thrown if an attempt is made to add an element to a fixed-length queue that is full. A NoSuchElementException is thrown if an attempt is made to remove an element from an empty queue.

9. Explain ArrayList class and explain following methods:

i)binarysearch ii)copyOf() iii)equals() iv)fill

i)binarysearch()

Searches for value in list ordered according to c. Returns the position of value in list, or a negative value if value is not found.

Syntax: static int binarySearch(List list, T value, Comparator <? Super T> c)

ii)copyOf()

The copyOf( ) method was added by Java SE 6. It returns a copy of an array and has the following forms:

static boolean[ ] copyOf(boolean[ ] source, int len)
static byte[ ] copyOf(byte[ ] source, int len)
static char[ ] copyOf(char[ ] source, int len)
static double[ ] copyOf(double[ ] source, int len)
static float[ ] copyOf(float[ ] source, int len)
static int[ ] copyOf(int[ ] source, int len)
static long[ ] copyOf(long[ ] source, int len)
static short[ ] copyOf(short[ ] source, int len)
static <T> T[ ] copyOf(T[ ] source, int len)
static <T,U> T[ ] copyOf(U[ ] source, int len, Class<? extends T[ ]> result T)

iii)equals()

The equals( ) method returns true if two arrays are equivalent. Otherwise, it returns false.
The equals( ) method has the following forms:

static boolean equals(boolean array1[ ], boolean array2[ ])
static boolean equals(byte array1[ ], byte array2[ ])
static boolean equals(char array1[ ], char array2[ ])
static boolean equals(double array1[ ], double array2[ ])
static boolean equals(float array1[ ], float array2[ ])
static boolean equals(int array1[ ], int array2[ ])
static boolean equals(long array1[ ], long array2[ ])
static boolean equals(short array1[ ], short array2[ ])
static boolean equals(Object array1[ ], Object array2[ ])

iv)fill()

The fill( ) method assigns a value to all elements in an array. In other words, it fills an array with a specified value. The fill( ) method has two versions. The first version, which has the following forms, fills an entire array:

static void fill(boolean array[ ], boolean value)
static void fill(byte array[ ], byte value)
static void fill(char array[ ], char value)
static void fill(double array[ ], double value)
static void fill(float array[ ], float value)
static void fill(int array[ ], int value)
static void fill(long array[ ], long value)
static void fill(short array[ ], short value)
static void fill(Object array[ ], Object value)

10.Write a program to find the first repeated and non-repeated character in a givenstring.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class Main {

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter a string: ");
      String input = scanner.nextLine();
```

```
        char firstRepeatedChar = findFirstRepeatedChar(input);
        char firstNonRepeatedChar = findFirstNonRepeatedChar(input);

        System.out.println("First repeated character: " + (firstRepeatedChar != '\0' ? firstRepeatedChar : "None"));
        System.out.println("First non-repeated character: " + (firstNonRepeatedChar != '\0' ? firstNonRepeatedChar : "None"));
    }

    public static char findFirstRepeatedChar(String str) {
        Map<Character, Integer> charCountMap = new HashMap<>();
        for (char c : str.toCharArray()) {
            charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);
        }
        for (char c : str.toCharArray()) {
            if (charCountMap.get(c) > 1) {
                return c;
            }
        }
        return '\0'; // No repeated character found
    }

    public static char findFirstNonRepeatedChar(String str) {
        Map<Character, Integer> charCountMap = new HashMap<>();
        for (char c : str.toCharArray()) {
            charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);
        }
        for (char c : str.toCharArray()) {
            if (charCountMap.get(c) == 1) {
                return c;
            }
        }
        return '\0'; // No non-repeated character found
    }
}
```