Internal Assessment Test 1 – March 2025											
Sub:		**Advanced Java -SET 1**					**Sub Code:**	**BIS402**	**Branch**	**ISE**	
Date:		**24-03-2024**	**Duration:**	**90 min's**	**Max Marks:**	**50**	**Sem/Sec:**	**IV/ A, B, & C**		**OBE**	
	Answer any FIVE questions								MARKS	CO	RBT
1		Define Collection. Explain with an Example different form of Lists supported by Java. **Solution:** The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects. **(i)The ArrayList Class** The ArrayList class extends AbstractList and implements the List interface. ArrayList is a generic class that has this declaration: **class ArrayList<E>** Here, E specifies the type of objects that the list will hold.ArrayList supports dynamic arrays that can grow as needed.An ArrayList is a variable-length array of object references. That is, an ArrayList can dynamically increase or decrease in size.Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk. ArrayList has the constructors shown here: 1. **ArrayList( )** 2. **ArrayList(Collection<? extends E> c)** 3. **ArrayList(int capacity)** The first constructor builds an empty array list. The second constructor builds an array list that is initialized with the elements of the collection c.The third constructor builds an array list that has the specified initial capacity. The contents of a collection are displayed using the default conversion provided by **toString( ),** which was inherited from AbstractCollection. you can increase the capacity of an ArrayList object manually by calling **ensureCapacity(**).if you want to reduce the size of the array that underlies an ArrayList object, call trimToSize( ). The signature for ensureCapacity( ) is shown here:void ensureCapacity(int cap).Here, cap specifies the new minimum capacity of the collection. **Example program:**		10	CO1	L2

```
package Collections;
import java.util.*;
public class ArrayListExample {
public static void main(String[] args) {
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " + al.size());
// Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " +
al.size());
// Display the array list.
System.out.println("Contents of al: " + al);
// Remove elements from the array list.
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " +
al.size());
System.out.println("Contents of al: " + al);
}
}
```

(ii) **The LinkedList Class**

The LinkedList class extends AbstractSequentialList and implements the List, Deque, and Queue interfaces.It provides a linked-list data structure. LinkedList is a generic class that has this declaration:

class LinkedList<E>

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

1. LinkedList( )
2. LinkedList(Collection<? extends E> c)

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

```
package Collections;

import java.util.*;

public class LinkedListExample {

        public static void main(String args[]) {

        // Create a linked list.

        LinkedList<String> ll = new LinkedList<String>();

        // Add elements to the linked list.

        ll.add("F");

        ll.add("B");

        ll.add("D");

        ll.add("E");

        ll.add("C");

        ll.addLast("Z");

        ll.addFirst("A");

        ll.add(1, "A2");

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

        // Remove elements from the linked list.

        ll.remove("F");

        ll.remove(2);

        System.out.println("Contents of ll after deletion: "

        + ll);
```

		// Remove first and last elements.
		ll.removeFirst();
		ll.removeLast();
		System.out.println("ll after deleting first and last: "
		+ ll);
		// Get and set a value.
		String val = ll.get(2);
		ll.set(2, val+ " changed");
		System.out.println("ll after change: " + ll);
		}
		}
		(iii) vector iv)stack→. To explain as other collecction			

2		**Define Iterator. Explain the concept of Iterator with a Programming Example.** **Solution:** To cycle through the elements in a collection you might want to display each element. way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface. 1. Iterator enables you to cycle through a collection, obtaining or removing elements. 2. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements. Iterator and ListIterator are generic interfaces which are declared as shown here: **interface Iterator<E>** **interface ListIterator<E>** Here, E specifies the type of objects being iterated. **The Methods Provided by Iterator**	10	CO1	L2

Method	Description
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
E next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
default void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**. The default version throws an **UnsupportedOperationException**.

The Methods Provided by ListIterator

Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.)
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

Using an Iterator

Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

Program:

```
package Collections;
import java.util.*;
public class IteratorExample {
//Demonstrate iterators.
public static void main(String args[]) {
//Create an array list.
ArrayList<String> al = new ArrayList<String>();
//Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
//Use iterator to display contents of al.
System.out.print("Original contents of al: ");
Iterator<String> itr = al.iterator();
while(itr.hasNext())
{
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
//Modify objects being iterated.
ListIterator<String> litr = al.listIterator();
while (litr.hasNext()) {
   String element = litr.next();
   // // Modify objects being iterated
```

```
   if(element.equals("A"))
            litr.set("Apple");
   else if(element.equals("B"))
            litr.set("Banana");
   else if(element.equals("C"))
            litr.set("cherry");
   else
            litr.set("Fruits");
       // litr.set(element + "+");
}
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
//Now, display the list backwards.
System.out.print("Modified list backwards: ");
while(litr.hasPrevious())
{
            String element = litr.previous();
            System.out.print(element + " ");
            }
            System.out.println();
            }
}
```

10

3		**Implement a java program to illustrate storing user defined classes in collection.**
		Solution:
		Storing User-Defined Classes in Collections
		Collections are not limited to the storage of built-in objects. The power of collections is that they can store any type of object, including objects of classes that you create. Collections offer off-the-shelf solutions to a wide variety of programming problems.
		Program:
		package Collections;
		import java.util.*;
		class Address
		{
		//A simple mailing list example.
		private String name;
		private String street;
		private String city;
		private String state;
		private String code;
		Address(String n, String s, String c,String st, String cd) {
		this.name = n;
		this.street = s;
		this.city = c;
		this.state = st;
		this.code = cd;
		}
		@Override
		public String toString() {
		return "Address [name=" + name + ", street=" + street + ", city=" + city + ", state=" + state + ", code=" + code + "]";
				CO1	L3

		}
		}
		public class MailList {
		public static void main(String args[]) {
		LinkedList<Address> ml = new LinkedList<>();
		//Add elements to the linked list.
		ml.add(new Address("J.W. West", "11 Oak Ave","Urbana", "IL", "61801"));
		ml.add(new Address("Ralph Baker", "1142 Maple Lane","Mahomet", "IL", "61853"));
		ml.add(new Address("Tom Carlton", "867 Elm St","Champaign", "IL", "61820"));
		//Display the mailing list.
		for(Address e : ml)
		{
		System.out.println(e + "\n");
		System.out.println();
		}}}			

4		**Explain different forms of String Search Methods supported in Java.** **Solution:** **Searching Strings:** The String class provides two methods that allow you to search a string for a specified character or substring: 1. **indexOf( )** Searches for the first occurrence of a character or substring. 2. **lastIndexOf( )** Searches for the last occurrence of a character or substring. These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure. To search for the first occurrence of a character, use **int indexOf(int ch)** To search for the last occurrence of a character, use **int lastIndexOf(int ch)** Here, ch is the character being sought.	10	CO2	L2

To search for the first or last occurrence of a substring, use

3. int indexOf(String str)

4. int lastIndexOf(String str)

Here, str specifies the substring.

You can specify a starting point for the search using these forms:

1. int indexOf(int ch, int startIndex)
2. int lastIndexOf(int ch, int startIndex)
3. int indexOf(String str, int startIndex)
4. int lastIndexOf(String str, int startIndex)

Here, startIndex specifies the index at which point the search begins. For indexOf(), the search runs from startIndex to the end of the string. For lastIndexOf( ), the search runs from startIndex to zero.

Program:

```
class indexOfDemo {

public static void main(String args[]) {

String s = "Now is the time for all good men " +

"to come to the aid of their country.";

System.out.println(s);

System.out.println("indexOf(t) = " +

s.indexOf('t'));

System.out.println("lastIndexOf(t) = " +

s.lastIndexOf('t'));

System.out.println("indexOf(the) = " +

s.indexOf("the"));

System.out.println("lastIndexOf(the) = " +

s.lastIndexOf("the"));

System.out.println("indexOf(t, 10) = " +

s.indexOf('t', 10));

System.out.println("lastIndexOf(t, 60) = " +

s.lastIndexOf('t', 60));
```

		System.out.println("indexOf(the, 10) = " +
		s.indexOf("the", 10));
		System.out.println("lastIndexOf(the, 60) = " +
		s.lastIndexOf("the", 60));
		}
		}			

5		**Implement a java program to illustrate the use of different types of StringBuffer methods.** **Solution:** **StringBuffer** • StringBuffer supports a modifiable string. As you know, String represents fixed-length, immutable character sequences. • In contrast, StringBuffer represents growable and writable character sequences. StringBuffer may have characters and substrings inserted in the middle or appended to the end. • StringBuffer will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth. **StringBuffer Constructors** 1. StringBuffer defines these four constructors: 2. StringBuffer( ) 3. StringBuffer(int size) 4. StringBuffer(String str) 5. StringBuffer(CharSequence chars) **Methods:** 1. StringBuffer append(String str) 2. StringBuffer append(int num) 3. StringBuffer append(Object obj) 4. StringBuffer insert(int index, String str) 5. StringBuffer insert(int index, char ch) 6. StringBuffer insert(int index, Object obj) 7. StringBuffer reverse( ) 8. StringBuffer delete(int startIndex, int endIndex) 9. StringBuffer deleteCharAt(int loc) 10. StringBuffer replace(int startIndex, int endIndex, String str) **Program:** package Strings;	10	CO2	L3

```
public class StringBufferDemo {
        public static void main(String[] args) {
                String word = "Hello";
                StringBuffer stringBuffer = new StringBuffer();
                stringBuffer.append(word);
                stringBuffer.insert(0, "Howdy ");
                stringBuffer.append(" Hi");
                System.out.println(stringBuffer.toString());
                System.out.println(stringBuffer.capacity());
                System.out.println(stringBuffer.length());
                System.out.println();//Division
                StringBuilder stringBuilder = new StringBuilder();
                stringBuilder.append(word);
                stringBuilder.insert(0, "Howdy ");
                stringBuilder.append(" Hi");
                System.out.println(stringBuilder.toString());
                System.out.println(stringBuilder.capacity());
                System.out.println(stringBuilder.length());
            }
        }
```

6		**Define a) toUppercase() b) contains() c) isEmpty() d) toString() e) valueOf()** **Solution:** **1. String toUpperCase( )** return a String object that contains the uppercase **2. boolean contains(CharSequence str)** Returns true if the invoking object contains the string specified by str. Returns false otherwise.	10	CO2	L2

		3. boolean isEmpty( ) Returns true if the invoking string contains no characters and has a length of zero. **4. String toString( )** simply return a String object that contains the human-readable string that appropriately describes an object of your class. **5. valueOf()** is overloaded for all the primitive types and for type Object. converts data into its string representation during **concatenation.** **Any sample java program using above methods.**			