Answer Key

Sub:	**Advanced Java**					Sub Code:	BIS402	Branch	ISE
Date:	17-04-2026	Duration:	90 min's	Max Marks:	50	Sem/Sec:	IV / A, B & C		OBE

	Answer any FIVE questions	MARKS	CO	RBT
1	a) What is Collection Framework? Explain the methods defined by the collection interface. **Answer:** The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects. It merits close attention by all programmers.	4 6	CO1	L1

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.

b) Mention and Explain different interfaces supported by Map.
Answer:

- A map is an object that stores associations between keys and values, or key/value pairs. Given a key, you can find its value. Both keys and values are objects.
- The keys must be unique, but the values may be duplicated. Some maps can accept a null key and null values, others cannot.

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

- **Map:**
- The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.
- Given a key and a value, you can store the value in a Map object.
- After the value is stored, you can retrieve it by using its key. Map is generic and is declared as shown here:

interface Map<K, V>

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

Methods defined by Map Interface:

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
default V compute(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	Calls *func* to construct a new value. If func returns non-**null**, the new key/value pair is added to the map, any preexisting pairing is removed, and the new value is returned. If *func* returns **null**, any preexisting pairing is removed, and **null** is returned. (Added by JDK 8.)
default V computeIfAbsent(K *k*, Function<? super K, ? extends V> *func*)	Returns the value associated with the key *k*. Otherwise, the value is constructed through a call to *func* and the pairing is entered into the map and the constructed value is returned. If no value can be constructed, **null** is returned. (Added by JDK 8.)
default V computeIfPresent(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	If *k* is in the map, a new value is constructed through a call to *func* and the new value replaces the old value in the map. In this case, the new value is returned. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned. (Added by JDK 8.)
boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.
Set<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map.

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
default V compute(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	Calls *func* to construct a new value. If func returns non-**null**, the new key/value pair is added to the map, any preexisting pairing is removed, and the new value is returned. If *func* returns **null**, any preexisting pairing is removed, and **null** is returned. (Added by JDK 8.)
default V computeIfAbsent(K *k*, Function<? super K, ? extends V> *func*)	Returns the value associated with the key *k*. Otherwise, the value is constructed through a call to *func* and the pairing is entered into the map and the constructed value is returned. If no value can be constructed, **null** is returned. (Added by JDK 8.)
default V computeIfPresent(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	If *k* is in the map, a new value is constructed through a call to *func* and the new value replaces the old value in the map. In this case, the new value is returned. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned. (Added by JDK 8.)
boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.
Set<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map.

Method	Description
default V merge(K *k*, V *v*, BiFunction<? super V, ? super V, ? extends V> *func*)	If *k* is not in the map, the pairing *k*,*v* is added to the map. In this case, *v* is returned. Otherwise, *func* returns a new value based on the old value, the key is updated to use this value, and **merge( )** returns this value. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned. (Added by JDK 8.)
V put(K *k*, V *v*)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are *k* and *v*, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> *m*)	Puts all the entries from *m* into this map.
default V putIfAbsent(K *k*, V *v*)	Inserts the key/value pair into the invoking map if this pairing is not already present or if the existing value is **null**. Returns the old value. The **null** value is returned when no previous mapping exists, or the value is **null**. (Added by JDK 8.)
V remove(Object *k*)	Removes the entry whose key equals *k*.
default boolean remove(Object *k*, Object *v*)	If the key/value pair specified by *k* and *v* is in the invoking map, it is removed and **true** is returned. Otherwise, **false** is returned. (Added by JDK 8.)
default boolean replace(K *k*, V *oldV*, V *newV*)	If the key/value pair specified by *k* and *oldV* is in the invoking map, the value is replaced by *newV* and **true** is returned. Otherwise **false** is returned. (Added by JDK 8.)

The SortedMap Interface:

- The SortedMap interface extends Map. It ensures that the entries are maintained in ascending order based on the keys.
- SortedMap is generic and is declared as shown here:

  interface SortedMap<K, V>
- Here, K specifies the type of keys, and V specifies the type of values

Methods:

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

- **NavigableMap**
- The NavigableMap interface extends SortedMap and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys.
- NavigableMap is a generic interface that has this declaration:**interface NavigableMap <K,V>**
- Here, K specifies the type of the keys, and V specifies the type of the values associated with the keys.

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

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

2	a) Explain Legacy classes in collection framework with sample code. **Solution:** • Early versions of java.util did not include the Collections Framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing objects. When collections were added (by J2SE 1.2), several of the original classes were reengineered to Framework. However, where a modern collection duplicates the functionality of a legacy class, you will usually want to use the newer collection class. In general, the legacy classes are supported because there is still code that uses them. • none of the modern collection classes described in this chapter are synchronized, but all the legacy classes are synchronized. • Legacy Classes: Dictionary Hashtable Properties Stack Vector Example: **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 duplicate the functionality of methods defined by the Collections Framework. Vector is declared like this: *class Vector* Here are the Vector constructors: Vector( ) Vector(int size) Vector(int size, int incr) Vector(Collection c)	4 6	C O 1	L1

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.
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.

b) Explain Set interface with constructors and methods.

Answer:

- The **Set** interface defines a set. It extends **Collection** and specifies the behavior of an collection that **does not allow duplicate elements.** Therefore, the **add( )** method returns **false** if an attempt is made to add duplicate elements to a set. It does not specify any additional methods of its own. **Set** is a generic interface that has this declaration:

interface Set<E>

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

(1)

- 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.
-

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 18-3 The Methods Declared by **SortedSet**

- (2) The NavigableSet interface extends SortedSet and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.
- NavigableSet is a generic interface that has this declaration:

interface NavigableSet<E>

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

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
E higher(E *obj*)	Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.

3	a) Explain how collectors can be accessed using an iterator with an example. **Answers:** • **To cycle through the elements in a collection** • **you might want to display each element**.	4 6	C O 1	L3

- way to do this is to employ an **iterator**, which is an object that i**mplements** either the **Iterator or the ListIterator interface.**

Steps:

1.Obtain an iterator to the start of the collection by **calling the collection's iterator( ) method.**

2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.

3. Within the loop, obtain each element by calling next( ).

Example Program:

```
package CollectionHandson;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;
public class IteratorExample {
//Demonstrate iterators.
public static void main(String args[]) {
//Create an array list.ogram
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();
        }
    }
```

b) What are comparators? Write a comparator program to sort accounts by last name

Answers:

- Both TreeSet and TreeMap store elements in sorted order.
- If you want to order elements a different way, then specify a Comparator when you construct the set or map.
- Doing so gives you the ability to govern precisely how elements are stored within sorted collections and maps.
- Comparator is a generic interface that has this declaration:

interface Comparator<T>

- Here, T specifies the type of objects being compared.

The **compare( ) method**, shown here, compares two elements for order:

int compare(T obj1, T obj2)

- obj1 and obj2 are the objects to be compared.

1. this method returns zero if the objects are equal.
2. It returns a positive value if obj1 is greater than obj2.
3. Otherwise, a negative value is returned.

- The equals( ) method, shown here, tests whether an object equals the invoking comparator:

boolean equals(object obj)

- Here, obj is the object to be tested for equality.
- The method returns true if obj and the invoking object are both Comparator objects and use the same ordering.
- Otherwise, it returns false.

Example Program:

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

// Account class
class Account {
String firstName;
String lastName;
int accNo;

Account(String firstName, String lastName, int accNo) {
this.firstName = firstName;
this.lastName = lastName;
this.accNo = accNo;
}

public String toString() {
return firstName + " " + lastName + " - " + accNo;
}
}

// Comparator to sort by last name
class LastNameComparator implements Comparator<Account> {
public int compare(Account a1, Account a2) {
return a1.lastName.compareTo(a2.lastName);
}
}

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

// Create TreeSet with custom comparator
TreeSet<Account> accounts = new TreeSet<>(new
LastNameComparator());

// Add accounts
accounts.add(new Account("John", "Smith", 101));
accounts.add(new Account("Alice", "Brown", 102));
accounts.add(new Account("David", "Clark", 103));
accounts.add(new Account("Emma", "Anderson", 104));
```

	// Display sorted accounts
	for(Account acc : accounts)
	System.out.println(acc);
	}
	}			

4	a) Explain StringBuffer constructor with example.
	Answer:
	StringBuffer supports a modifiable string. As you know,String represents fixed-length, immutable charactersequences.
	• In contrast, StringBuffer represents growable and writablecharacter sequences.
	• StringBuffer may have characters and substrings inserted inthe middle or appended to the end.
	• StringBuffer will automatically grow to make room for suchadditions and often has more characters pre-allocated than are actually needed, to allow room for growth.
	StringBuffer Constructors
	StringBuffer defines these four constructors:
	1. StringBuffer( )
	2. StringBuffer(int size)
	3. StringBuffer(String str)
	4. StringBuffer(CharSequence chars)
	1. StringBuffer sb1 = new StringBuffer();
	2. StringBuffer sb2 = new StringBuffer(30);
	3. StringBuffer sb3 = new StringBuffer("Hello");
	4. CharSequence cs = "Welcome";
	StringBuffer sb4 = new StringBuffer(cs);
	b) Write a Java program to remove duplicate characters from the given string and display the result nswers
	
	Solution:
	import java.util.*;
	
	public class RemoveDuplicates {
	public static void main(String[] args) {
	
	String input = "programming";
	String result = "";
	
	// Use LinkedHashSet to preserve insertion order
	Set<Character> set = new LinkedHashSet<>();
	
	// Add characters to set
	for (char ch : input.toCharArray()) {
	set.add(ch);
	}
	
	// Convert set back to string
	for (char ch : set) {
	result += ch;
		4
		6
			C
			O
			2
				L3

```
}

System.out.println("Original String: " + input);
System.out.println("After Removing Duplicates: " + result);
}
}
```

4.b

Write a Java program to remove duplicate characters from the given string and display the result ant string [provide sample output].

Solution:

```
import java.util.LinkedHashSet;
import java.util.Scanner;

public class RemoveDuplicates {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter a string: ");
        String input = sc.nextLine();

        // Using LinkedHashSet to maintain insertion order
        LinkedHashSet<Character> set = new LinkedHashSet<>();

        for (char ch : input.toCharArray()) {
            set.add(ch);
        }

        // Build result string
        StringBuilder result = new StringBuilder();
        for (char ch : set) {
            result.append(ch);
        }

        System.out.println("String after removing duplicates: " + result);
    }
}
```

5 a) Differentiate between equals() and == with respect to String class and StringBuffer.

4 6 | C O 2 | L1

Feature	String Class	StringBuffer Class
== Operator	Compares memory addresses (references)	Compares memory addresses (references)
.equals() Method	Compares character content	Compares memory addresses

Common Use	Use `.equals()` to check if text/content is the same	Use `.toString().equals()` for content comparison

b)Explain the following with Examples:
1. valueOf() 2.toLowerCase() 3. toUpperCase()
4.indexOf() 5. toCharArray( ) 6 toString()

Answer:

1.valueOf()

Converts different data types into a **String**

Example:

```
int num = 100;
String str = String.valueOf(num);

System.out.println(str);        // "100"
System.out.println(str + 50);   // "10050"
```

Used to convert **int, float, boolean, char, etc. → String**

---

2. toLowerCase()

Converts all characters to **lowercase**

Example:

```
String s = "HELLO JAVA";
System.out.println(s.toLowerCase());
```

Output:

hello java

---

3. toUpperCase()

Converts all characters to **uppercase**

Example:

```
String s = "hello java";
System.out.println(s.toUpperCase());
```

Output:

HELLO JAVA

4.indexOf()

Returns the **position (index)** of a character or substring

Example:

```
String s = "programming";

System.out.println(s.indexOf('g'));     // first occurrence
System.out.println(s.indexOf("gram"));  // substring
```

Output:

3
3

Returns **-1** if not found

---

5. toCharArray()

Converts string into a **character array**

Example:

```
String s = "Java";
char[] arr = s.toCharArray();

for(char c : arr) {
   System.out.print(c + " ");
}
```

Output:

J a v a

6. toString()

Converts an object into its **String representation**

	Example: String s = "Hello"; System.out.println(s.toString()); **Output:** Hello Often used with objects: Integer num = 50; System.out.println(num.toString());			

6	a) Briefly String Builder with Example program explain.
	
	Solution:
	
	StringBuilder is a class in Java used to create **mutable (modifiable) strings**. Unlike String, whose content cannot be changed once created, StringBuilder allows you to **modify the same object** without creating new ones.
	
	Key Features
	
	• Mutable (can change content)
	• Faster than String for frequent modifications
	• Not synchronized (so it's faster but **not thread-safe**)
	• Located in java.lang package
	
	Example Program:
	class StringBuilderDemo {
	public static void main(String[] args) {
	
	// Create StringBuilder object
	StringBuilder sb = new StringBuilder("Hello");
	
	// Append text
	sb.append(" Java");
	
	// Insert text
	sb.insert(5, " World");
	
	// Replace text
	sb.replace(6, 11, "Java");
	
	// Reverse string
	sb.reverse();
	
	// Display result
	System.out.println("Final String: " + sb);
	}
	}
		4
		6
			C
			O
			3
				L2

b)Write short note MVC Connection and Justify how MVC is different from Model- Delegate architectures.

Answer:

The MVC Connection

is a visual component is a composite of three distinct aspects:

1. The way that the component looks when rendered on the screen

2. The way that the component reacts to the user

3. The state information associated with the component

one component architecture has proven itself to be exceptionally effective: **Model-View-Controller, or MVC for short.**

In MVC terminology, the model corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.

• The view determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.

• The controller determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked).

• By separating a component into a model, a view, and a controller, the

specific implementation of each can be changed without affecting the other two.

• For instance, different view implementations can render the same component in different ways without affecting the model or the controller.Model-Delegate architecture

• Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the UI delegate.

• Swing's approach is called either the Model-Delegate architecture or the Separable Model architecture.

Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.

• Swing's pluggable look and feel is made possible by its Model-Delegate architecture.

	Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program. • it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input. To support the Model-Delegate architecture, most Swing components contain two objects. 1. The first represents the model. 2. The second represents the UI delegate. • Models are defined by interfaces. For example, the model for a button is defined by the ButtonModel interface. • UI delegates are classes that inherit ComponentUI. For example, the UI delegate for a button is ButtonUI. • Normally, your programs will not interact directly with the UI delegate.			