
USN

VTU Question Paper - Solution

Sub:	Advanced Java				Sub Code:	BIS402	Branch:	ISE
Date:	03-07-2025	Duration:	3 hours	Max Marks:	100			OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1(a)	What is Collection Framework?Explain the methods defined by the collection interface.	[07]	CO1	L2

The **Collection Framework** in Java is a unified architecture for representing and manipulating collections, which are groups of objects. It provides a set of interfaces, implementations (classes), and algorithms that enable developers to work efficiently with different types of collections such as lists, sets, queues, and more. The framework defines standard methods for adding, removing, and accessing elements, and supports various operations like searching, sorting, and iterating.

Core Interfaces of the Collection Framework

The core collection interfaces define the fundamental behavior of collections. Some important interfaces are:

- **Collection**: The root interface for working with groups of objects.
- **List**: Extends Collection to handle ordered sequences.
- **Set**: Extends Collection to handle unique elements.
- **Queue**: Extends Collection to handle special lists where elements are processed in order.
- **Deque**: Extends Queue for double-ended queues.
- **SortedSet**: Extends Set for sorted collections.
- **NavigableSet**: Extends SortedSet to support closest-match searches.

Methods Defined by the Collection Interface

All collections implement the **Collection<E>** interface, which declares these essential methods:

Method	Description
boolean add(E obj)	Adds an object to the collection. Returns true if added successfully.
boolean addAll(Collection<? extends E> c)	Adds all elements from another collection. Returns true if the collection changed.
void clear()	Removes all elements from the collection.
boolean contains(Object obj)	Checks if the collection contains a specific object.
boolean containsAll(Collection<?> c)	Checks if the collection contains all elements of another collection.

	boolean equals(Object obj)	Checks if this collection is equal to another object.			
	int hashCode()	Returns the hash code value for the collection.			
	boolean isEmpty()	Returns true if the collection has no elements.			
	Iterator<E> iterator()	Returns an iterator to iterate over the collection's elements.			
	boolean remove(Object obj)	Removes a single instance of the specified object. Returns true if removed.			
	boolean removeAll(Collection<?> c)	Removes all elements that are also contained in the specified collection.			
	boolean retainAll(Collection<?> c)	Retains only the elements that are contained in the specified collection.			
	int size()	Returns the number of elements in the collection.			
	Object[] toArray()	Returns an array containing all elements in the collection.			
	<T> T[] toArray(T[] array)	Returns an array containing all elements, using the provided array if large enough.			

1(b)	Demonstrate ArrayList Class collection with an example.
	
	The **ArrayList** class is a part of the java.util package. It extends AbstractList and implements the List interface. Unlike arrays in Java, which have fixed size, an ArrayList supports **dynamic resizing**, allowing it to grow or shrink automatically as elements are added or removed.
	
	Declaration:
	class ArrayList<E>
	Here, E represents the type of elements stored in the list.
	
	Key Features:
	
	● Supports random access via index.
	● Allows duplicate elements.
	● Automatically resizes the internal array.
	● Provides flexibility over standard arrays.
	
	Common Constructors:
	
	● ArrayList() – creates an empty list.
	● ArrayList(Collection<? extends E> c) – initializes list with elements from another collection.
	● ArrayList(int capacity) – creates a list with specified initial capacity.
	
	Example:
	
	import java.util.*;
	public class secondexample {
	public static void main(String[] args) {
	// **TODO** Auto-generated method stub
		[07]	CO1	L2

```
                ArrayList<Integer> al=new ArrayList<Integer>();
                System.out.println("Size of al:" + al.size());
                al.add(4);
                al.add(2);
                al.add(3);
                al.add(1,8);
                al.add(1);
                al.add(6);
                System.out.println("Size of al after additions:" + al.size());
                System.out.println("Content of al:" + al);
                al.remove(3);  //Passing index
                al.remove(2);
                System.out.println("Size of al after removal:" + al.size());
                System.out.println("Content of al:" + al);
                Collections.sort(al);
                System.out.println("Content of al after sorting:" + al);

        }
}
```

Output–

```
Size of al:0
Size of al after additions:6
Content of al:[4, 8, 2, 3, 1, 6]
Size of al after removal : 4
Content of al:[4, 8, 1, 6]
Content of al after sorting:[1, 4, 6, 8]
```

1(c)	Explain how collectors can be accessed using an iterator with an example.
	
	Accessing a Collection via an Iterator – Explanation in Points
	
	● The Iterator and ListIterator interfaces in Java provide standard ways to traverse collections.
	● Iterator<E> allows unidirectional traversal (forward only) of a collection.
	● ListIterator<E> extends Iterator<E> to support bidirectional traversal (both forward and backward) and element modification.
	● All collection classes provide an iterator() method to return an Iterator.
	● Lists also provide listIterator() for more powerful traversal using ListIterator.
	
	Steps to Access a Collection Using an Iterator:
	
	1. Obtain an iterator using iterator() or listIterator() method.
	2. Use a loop that checks hasNext() to iterate forward.
	3. Call next() to retrieve each element.
	4. Optionally, use remove() to delete elements during iteration (not supported by all collections).
	5. For ListIterator, use hasPrevious() and previous() to go backward.
	
	Example: Demonstrating Iterator and ListIterator:
	import java.util.*;
	
	class IteratorDemo {
	public static void main(String args[]) {
	// Create an ArrayList and add elements
	ArrayList<String> al = new ArrayList<String>();
	al.add("C");
	al.add("A");
	al.add("E");
	al.add("B");
	al.add("D");
	al.add("F");
	
	// Use Iterator to display original contents
	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 elements using ListIterator
	ListIterator<String> litr = al.listIterator();
	while (litr.hasNext()) {
	String element = litr.next();
	litr.set(element + "+");
	}
	
	// Display modified contents
	System.out.print("Modified contents of al: ");
		[06]	CO1	L2

```
      itr = al.iterator();
      while (itr.hasNext()) {
         String element = itr.next();
         System.out.print(element + " ");
      }
      System.out.println();

      // Display the list in reverse order
      System.out.print("Modified list backwards: ");
      while (litr.hasPrevious()) {
         String element = litr.previous();
         System.out.print(element + " ");
      }
      System.out.println();
   }
}
```

Output:

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

2(a)	Explain the following map classes:
	i)HashMap ii)TreeMap
	
	(i) HashMap
	
	● The HashMap class in Java is part of the java.util package. It extends AbstractMap and implements the Map interface.
	● It stores key-value pairs using a hash table.
	● The time complexity for basic operations like put() and get() is constant (O(1)), even for large data sets.
	● The class is declared as:
	class HashMap<K, V> where K is the type of keys and V is the type of values.
	
	Important Constructors:
	
	1. HashMap() – creates an empty HashMap with default capacity and load factor.
	2. HashMap(int capacity) – specifies initial capacity.
	3. HashMap(int capacity, float fillRatio) – specifies both initial capacity and load factor.
	4. HashMap(Map<? extends K, ? extends V> m) – initializes map with elements from another map.
	
	Key Features:
	
	● Keys must be unique.
	● Allows null keys and values.
	● Does not guarantee any order of elements.
	● Methods like entrySet(), getKey(), getValue(), and put() are commonly used.
	
	Example:
	
	import java.util.*;
	
	public class HashMapExample {
	
	public static void main(String[] args) {
	
	HashMap<String, Double> salaryMap = new HashMap<>();
	
	salaryMap.put("Alice", 5000.00);
	
	salaryMap.put("Bob", 12000.00);
	
	salaryMap.put("Charlie", 75000.00);
	
	salaryMap.put("Diana", 88000.00);
	
	salaryMap.put("Ethan", 9500.00);
	
	Set<Map.Entry<String, Double>> entries = salaryMap.entrySet();
		[10]	CO1	L2

```
        for (Map.Entry<String, Double> entry : entries) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }

        System.out.println();

        double currentSalary = salaryMap.get("Alice");
        salaryMap.put("Alice", currentSalary * 0.05);
        System.out.println("Alice's new salary is: " + salaryMap.get("Alice"));
    }
}
```

Output:

Alice: 5000.0

Bob: 12000.0

Charlie: 75000.0

Diana: 88000.0

Ethan: 9500.0

Alice's new salary is: 250.0

(ii)TreeMap

- The TreeMap class extends AbstractMap and implements the NavigableMap interface.
- It stores key-value pairs in a red-black tree, which is a self-balancing binary search tree.
- The keys are always kept in sorted (ascending) order.

Class Declaration:
class TreeMap<K, V>

Important Constructors:

1. TreeMap() – constructs an empty map sorted by natural ordering.
2. TreeMap(Comparator<? super K> comp) – uses a custom comparator.
3. TreeMap(Map<? extends K, ? extends V> m) – initializes from another map.

4. TreeMap(SortedMap<K, ? extends V> sm) – initializes from a sorted map.

Key Features:

- Keys are sorted automatically.
- Does not allow null keys (throws NullPointerException).
- Slower than HashMap for get() and put() (O(log n)) due to tree traversal.
- Useful when sorted map is required.

Example:

```
package javaclassss;
import java.util.*;
public class TreeMapDemo {
   public static void main(String[] args) {

      // Create a TreeMap
      TreeMap<String, Double> tm = new TreeMap<>();
      // Put elements into the map
      tm.put("John Doe", 3434.34);
      tm.put("Tom Smith", 123.22);
      tm.put("Jane Baker", 1378.00);
      tm.put("Tod Hall", 99.22);
      tm.put("Ralph Smith", -19.08);

      // Get a set of the entries and display them
      Set<Map.Entry<String, Double>> set = tm.entrySet();
      System.out.println("Account Balances:");

      for (Map.Entry<String, Double> me : set) {
         System.out.println(me.getKey() + ": " + me.getValue());
      }

      System.out.println();
      // Deposit 1000 into John Doe's account
      double balance = tm.get("John Doe");
      tm.put("John Doe", balance + 1000);
      System.out.println("John Doe's new balance: " + tm.get("John Doe"));
   }
}
```

Output:

```
Account Balances:
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Tod Hall: 99.22
Tom Smith: 123.22
John Doe's new balance: 4434.34
```

2(b)	What are comparators? Write a comparator program to sort accounts by last name.	[10]	CO1	L3

In Java, Comparators allow us to define custom sorting logic for collections like TreeMap or TreeSet.

By default:

TreeMap and TreeSet sort elements using their natural ordering (e.g., String alphabetically, Integer numerically).

To change this order, we use a Comparator.

Comparator Interface

```
interface Comparator<T> {
   int compare(T obj1, T obj2); // Required
   boolean equals(Object obj);  // Optional
}
```

Returns 0 → equal

Returns < 0 → obj1 comes before obj2

Returns > 0 → obj1 comes after obj2

Example: Sort TreeMap by Last Name Using Comparator

```
// File: TreeMapLastNameSort.java
package javaclassss;
import java.util.*;
// Comparator to sort by last name
class LastNameComparator implements Comparator<String> {
   public int compare(String aStr, String bStr) {
      int i = aStr.lastIndexOf(' ');
      int j = bStr.lastIndexOf(' ');
      String lastName1 = aStr.substring(i + 1);
      String lastName2 = bStr.substring(j + 1);
      int lastCompare = lastName1.compareToIgnoreCase(lastName2);
      if (lastCompare != 0)
         return lastCompare;
      else
         return aStr.compareToIgnoreCase(bStr); // If last names same, compare full names
   }
}
public class TreeMapLastNameSort {
   public static void main(String[] args) {
      // TreeMap sorted by last name using custom comparator
      TreeMap<String, Double> tm = new TreeMap<>(new LastNameComparator());
      // Adding entries
      tm.put("John Doe", 3434.34);
      tm.put("Tom Smith", 123.22);
      tm.put("Jane Baker", 1378.00);
      tm.put("Tod Hall", 99.22);
      tm.put("Ralph Smith", -19.08);
      // Display entries
```

```
        System.out.println("Account Balances (Sorted by Last Name):");
        for (Map.Entry<String, Double> me : tm.entrySet()) {
           System.out.println(me.getKey() + ": " + me.getValue());
        }
        System.out.println();
        // Deposit 1000 into John Doe's account
        double balance = tm.get("John Doe");
        tm.put("John Doe", balance + 1000);
        System.out.println("John Doe's new balance: " + tm.get("John Doe"));
     }
}
```

Output:

```
Account Balances (Sorted by Last Name):
Jane Baker: 1378.0
John Doe: 3434.34
Tod Hall: 99.22
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe's new balance: 4434.34
```

3(a)	Explain the String comparison functions with suitable programs.	[06]	CO2	L2

Java provides several methods in the String class to compare strings in different ways. These include:

1. equals( ) and equalsIgnoreCase( )

The equals( ) method compares two strings for equality, considering case sensitivity.

The equalsIgnoreCase( ) method ignores case when comparing two strings.

- **Syntax**: boolean equals(Object str)
- **Syntax**: boolean equalsIgnoreCase(String str)

Example:

```
class EqualsDemo {
   public static void main(String args[]) {
      String s1 = "Hello";
      String s2 = "Hello";
      String s3 = "Good-bye";
      String s4 = "HELLO";

      System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2)); // true
      System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3)); // false
      System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4)); // false
      System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " + s1.equalsIgnoreCase(s4));
// true
   }
}
```

Output-
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true

2. regionMatches( )

The regionMatches( ) method compares a specific region inside a string with another specified region of another string.

- **Returns**: true if the specified regions match; otherwise false.
- **Use Case**: Useful for partial comparisons with or without case sensitivity.
- **Syntax: boolean regionMatches(int toffset, String other, int ooffset, int len)**

  toffset – starting index in the current string

  other – the other string to compare with

  ooffset – starting index in the other string

  len – number of characters to compare

Examples:

```
public class RegionMatchesExample {
   public static void main(String[] args) {
      String str1 = "Hello World";
      String str2 = "World";

      // Comparing "World" in str1 with str2
      boolean result = str1.regionMatches(6, str2, 0, 5);
      System.out.println("Do regions match? " + result);
   }
}
```

Output-
Do regions match? True

3. startsWith( ) and endsWith( )

- startsWith( ) checks if a string starts with the given substring.

  →**Returns**: true if the string starts with the specified prefix; otherwise false.

- endsWith( ) checks if a string ends with the given substring.

  →**Returns**: true if the string ends with the specified suffix.

Example:

```
public class StartsEndsExample {
   public static void main(String[] args) {
      String str = "Foobar";

      System.out.println(str.startsWith("Foo")); // true
      System.out.println(str.endsWith("bar")); // true
      System.out.println(str.startsWith("bar", 3)); // true
   }
}
```

Output-

true
true
True

4. equals( ) Versus ==

- equals( ) compares actual content of two strings.
- == checks if both references point to the same object.

Example:

```
class EqualsNotEqualTo {
   public static void main(String args[]) {
      String s1 = "Hello";
      String s2 = new String(s1);

      System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2)); // true
      System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2)); // false
   }
}
```

Output-

Hello equals Hello -> true
Hello == Hello -> false

5. compareTo( )

- compareTo( ) is used for lexicographical (dictionary) order comparison.
- It returns:
    - 0 → if strings are equal.
    - >0 → if the first string is greater.

- <0 → if the first string is smaller.

6. compareToIgnoreCase( )

→Same as compareTo( ), but it ignores case differences.

Examples:

```
public class CompareExample {
   public static void main(String[] args) {
      String str1 = "Apple";
      String str2 = "Banana";
      String str3 = "Apple";
      // Using compareTo()
      System.out.println(str1.compareTo(str2)); // Negative value (Apple < Banana)
      System.out.println(str1.compareTo(str3)); // 0 (Apple == Apple)
      System.out.println(str2.compareTo(str1)); // Positive value (Banana > Apple)
      // Using compareToIgnoreCase()
      String str4 = "apple";
      System.out.println(str1.compareToIgnoreCase(str4)); // 0 (case ignored)
   }
}
```

Output-

```
-1
0
1
0
```

3(b)	Explain the following built in methods with respect to StringBuffer class: (i)capacity() (ii)delete() (iii)replace() (iv)append() (v)substring()	[07]	CO2	L2

StringBuffer

- StringBuffer is a **mutable** sequence of characters (unlike String which is immutable).
- It's part of java.lang and is **thread-safe**.
- It provides several methods to manipulate character data efficiently.

(i) capacity()

- **Description:** Returns the current capacity of the StringBuffer.
- **Syntax:** int capacity()
- **Example:**

```
StringBuffer sb = new StringBuffer("Hello");
System.out.println(sb.capacity());  // Output: 21
```

(ii) delete()

- **Description:** Deletes characters from start index (inclusive) to end index (exclusive).
- **Syntax:** StringBuffer delete(int start, int end)
- **Example:**

```
StringBuffer sb = new StringBuffer("HelloWorld");
sb.delete(5, 10);
System.out.println(sb);  // Output: Hello
```

(iii)replace()

- The replace() method **replaces a portion of characters** in the StringBuffer with a new string.
- Syntax:

  StringBuffer replace(int startIndex, int endIndex, String str)

→The substring from startIndex to endIndex - 1 is replaced with str.

Example:

```
StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was");
System.out.println("After replace: " + sb);
// Output: After replace: This was a test.
```

(iv) 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, such as:

  StringBuffer append(String str)

  StringBuffer append(int num)

  StringBuffer append(Object obj)

- The string representation of each parameter is obtained, often by calling String.valueOf(), and appended to the buffer.
- It **returns the same** StringBuffer **object**, enabling **method chaining**.

Example:

```
String s;

int a = 42;

StringBuffer sb = new StringBuffer(40);

s = sb.append("a = ").append(a).append("!").toString();

System.out.println(s);  // Output: a = 42!
```

(v)substring()

The substring() method is used to extract a portion (substring) from a string.

Method Forms:

1. String substring(int startIndex)
   - Returns a substring from startIndex to the end of the string.
2. String substring(int startIndex, int endIndex)
   - Returns a substring from startIndex to endIndex - 1 (i.e., endIndex is excluded).

Example:

```
public class SubstringExample {
   public static void main(String[] args) {
      String str = "HelloWorld";

      // Get substring from index 0 to end
      String sub1 = str.substring(5);
      System.out.println("Substring from index 5: " + sub1); // Output: World

      // Get substring from index 0 to 5 (excluding 5)
      String sub2 = str.substring(0, 5);
      System.out.println("Substring from index 0 to 5: " + sub2); // Output: Hello
   }
}
```

Output:

```
Substring from index 5: World
Substring from index 0 to 5: Hello
```

3(c)	Write a java program that demonstrates any four constructors of string class.
	
	In Java, a String is a sequence of characters that is implemented as an object of type String. Unlike some other languages where strings are just character arrays, Java treats strings as full-fledged objects, allowing rich and convenient manipulation.
	
	Key Points:
	
	● Java provides **built-in support** for string handling through the String class.
	● The String class offers methods for **comparison**, **searching**, **concatenation**, **case conversion**, and more.
	● Once created, a String object is **immutable** — its characters cannot be changed.
	
	Simplified Java program demonstrating **4 String constructors**:
	
	package Exam;
	
	public class A {
	  public static void main(String[] args) {
	  // 1. Creating an empty String
	  String s1 = new String();
	  System.out.println("Empty String: " + s1);
	
	  // 2. Creating a String from a character array
	  char chars[] = {'a', 'b', 'c'};
	  String s2 = new String(chars);
	  System.out.println("String from char array: " + s2);
	
	  // 3. Creating a String from a byte array
	  byte ascii[] = {65, 66, 67, 68}; // ASCII values for 'A', 'B', 'C', 'D'
	  String s3 = new String(ascii);
	  System.out.println("String from byte array: " + s3);
	
	  // 4. Creating a String from another String
	  String s4 = new String(s2);
	  System.out.println("String from another String: " + s4);
	  }
	}
	
	Output:
	
	Empty String:
	String from char array: abc
	String from byte array: ABCD
	String from another String: abc
		[07]	CO2	L3

4(a)	Write a Java program to remove duplicate characters from a given String and display the resultant String.
	
	This program removes all duplicate characters from a given string, keeping only the **first occurrence** of each character.
	
	// Program to remove duplicate characters from a String
	public class RemoveDuplicates {
	public static void main(String[] args) {
	String input = "programming";
	String result = removeDuplicates(input);
	System.out.println("Original String: " + input);
	System.out.println("String after removing duplicates: " + result);
	}
	
	static String removeDuplicates(String str) {
	StringBuilder result = new StringBuilder();
	boolean[] seen = new boolean[256]; // ASCII character set
	
	for (int i = 0; i < str.length(); i++) {
	char ch = str.charAt(i);
	if (!seen[ch]) {
	seen[ch] = true;
	result.append(ch);
	}
	}
	return result.toString();
	}
	}
	Output–
	Original String: programming
	String after removing duplicates: progamin
		[07]	CO3	L3

4(b)	Explain character extraction functions in string class.	[07]	CO2	L2

Java provides several methods in the String class to extract characters from a string. Below are the key methods along with their explanations and example programs.

1. charAt( )

The charAt(int index) method returns the character at the specified index in a string.

Syntax:

```
char charAt(int index)
```

Example:

```
public class CharAtDemo {
   public static void main(String args[]) {
      String str = "abc";
      char ch = str.charAt(1); // Extract character at index 1
      System.out.println("Character at index 1: " + ch);
   }
}
```

Output-

```
Character at index 1: b
```

2. getChars( )

The getChars( ) method extracts multiple characters at a time and stores them in an array.

Syntax:

```
void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)
```

- sourceStart → starting index in the original string (inclusive)
- sourceEnd → ending index in the original string (exclusive)
- target → destination character array
- targetStart → index in target[] at which to start placing the characters

Example:

```
class GetCharsDemo {
   public static void main(String args[]) {
      String s = "This is a demo of the getChars method.";
```

```
        int start = 10;
        int end = 14;
        char buf[] = new char[end - start];

        s.getChars(start, end, buf, 0);
        System.out.println(buf); // Prints extracted characters
    }
}
```

Output-

demo

3. getBytes( )

The getBytes( ) method converts a string into an array of bytes.

Syntax:

```
byte[ ] getBytes( )
```

Example:

```
public class GetBytesDemo {
    public static void main(String args[]) {
        String str = "Hello";
        byte[] byteArray = str.getBytes();

        System.out.println("Byte array:");
        for (byte b : byteArray) {
            System.out.print(b + " ");
        }
    }
}
```

Output-

Byte array:

72 101 108 108 111

4. toCharArray( )

The toCharArray( ) method converts the entire string into a character array.

Syntax:

```
char[ ] toCharArray( )
```

Example:

```
package Strings;


public class ToCharArrayDemo {
   public static void main(String[] args) {
      String str = "Hello";
      char[] charArray = str.toCharArray();


      System.out.println("Character array:");
      for (char ch : charArray) {
         System.out.print(ch + " ");
      }
   }
}
```

Output-

Character array:

H e l l o

4(c)	Explain constructors in Java StringBuilder class. The **StringBuilder** class in Java provides several constructors to create mutable sequences of characters. Here's a brief explanation of each: **1.** StringBuilder() ● **Description:** Creates an empty StringBuilder with the default capacity (16 characters). ● **Example:** StringBuilder sb1 = new StringBuilder(); **2.** StringBuilder(int capacity) ● **Description:** Creates an empty StringBuilder with the specified initial capacity. ● **Example:**	[06]	CO2	L2

StringBuilder sb2 = new StringBuilder(50);

3. StringBuilder(String str)

- **Description:** Creates a StringBuilder initialized with the contents of the given string.
- **Example:**

StringBuilder sb3 = new StringBuilder("Hello");

4. StringBuilder(CharSequence seq)

- **Description:** Creates a StringBuilder containing the characters from the given CharSequence (like String, StringBuffer, etc.).
- **Example:**

CharSequence cs = "World";
StringBuilder sb4 = new StringBuilder(cs);

5(a) Explain the differences between AWT and Swing. What are two key features of Swing and explain. [06] CO3 L2

Differences between AWT and Swing:

Aspect	**AWT (Abstract Window Toolkit)**	**Swing**
Platform Dependency	Platform-dependent (uses native code peers)	Platform-independent (written in pure Java)
Component Type	Heavyweight components (mapped to native OS components)	Lightweight components (no native peer used)
Look and Feel	Fixed look and feel (OS-defined)	Pluggable look and feel (user can change styles)
Consistency Across OS	Inconsistent behavior across platforms	Consistent behavior on all platforms
Flexibility and Control	Limited customization and control	Highly customizable and flexible
Integration	Basic GUI support	Rich, advanced GUI controls (tables, trees, tabbed panes)

Two Key Features of Swing:

1. Lightweight Components

Swing components are mostly *lightweight*, meaning they are written entirely in Java and **do not rely on platform-specific native peers**. This ensures:

- **Consistency** across platforms.
- **Efficiency** and **flexibility** in GUI design.
- Better support for *customization* and rendering.

2. Pluggable Look and Feel (PLAF)

	Swing supports a **pluggable look and feel**, allowing developers to change the appearance of components **without altering their behavior**. Benefits include: ● A consistent UI across all platforms. ● Ability to adopt platform-specific styles (e.g., Windows, Nimbus). ● Custom look-and-feel can be created or changed at **runtime**.			

5(b)	What is JLabel class? Explain with example of any three constructors and methods of JLabel class.
	
	JLabel Class in Java:
	JLabel is a Swing component used to display **text**, and optionally an **icon**, in a graphical user interface (GUI). It is **non-editable** and **does not respond** to user input (passive component). It is mainly used to provide labels or messages.
	Three Common Constructors of JLabel**:**
	
	1. JLabel()
	➤ Creates a label with no text or icon.
	➤ *Example:* JLabel label1 = new JLabel();
	2. JLabel(String text)
	➤ Creates a label with specified text.
	➤ *Example:* JLabel label2 = new JLabel("Welcome");
	3. JLabel(String text, int horizontalAlignment)
	➤ Creates a label with text and horizontal alignment like JLabel.LEFT, JLabel.CENTER, etc.
	➤ *Example:* JLabel label3 = new JLabel("Centered Text", JLabel.CENTER);
	
	◆ **Three Common Methods of** JLabel**:**
	
	1. setText(String text)
	➤ Sets or updates the text of the label.
	➤ *Example:* label.setText("Updated Label");
	2. getText()
	➤ Returns the current text of the label.
	➤ *Example:* String str = label.getText();
	3. setHorizontalAlignment(int alignment)
	➤ Sets the horizontal alignment of the label's content.
	➤ *Example:* label.setHorizontalAlignment(JLabel.RIGHT);
	
	◆ **Example Program:**
	`import javax.swing.*;`
	`public class JLabelExample extends JFrame {`
	`  public JLabelExample() {`
	`  setTitle("JLabel Example");`
	`  setSize(300, 150);`
	`  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
	`  JLabel label = new JLabel("Welcome to Java Swing", JLabel.CENTER);`
	`  label.setText("Updated Text");  // Using setText()`
	
	`  add(label);`
	`  setVisible(true);`
	`  }`
		[07]	CO3	L2

```
  public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> new JLabelExample());
  }
}
```

5(c) Write a Java program in swing event handling applications that creates 2 buttons ALPHA and BETA and displays the text “Alpha pressed” when alpha button is clicked and “Beta pressed” when beta button is clicked. [07] CO3 L3

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.*;

public class EventDemo {

  JLabel jlab;



  EventDemo() {

    // Create JFrame container

    JFrame jfrm = new JFrame("An Event Example");



    // Set layout manager

    jfrm.setLayout(new FlowLayout());



    // Set initial size

    jfrm.setSize(220, 90);



    // Close program when window is closed

    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);



    // Create two buttons

    JButton jbtnAlpha = new JButton("ALPHA");

    JButton jbtnBeta = new JButton("BETA");
```

```
    // Add action listener for Alpha button using anonymous inner class
    jbtnAlpha.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            jlab.setText("Alpha was pressed.");
        }
    });

    // Add action listener for Beta button using anonymous inner class
    jbtnBeta.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            jlab.setText("Beta was pressed.");
        }
    });

    // Add buttons to the frame
    jfrm.add(jbtnAlpha);
    jfrm.add(jbtnBeta);

    // Create and add label to frame
    jlab = new JLabel("Press a button.");
    jfrm.add(jlab);

    // Make frame visible
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // Launch GUI on Event Dispatch Thread
    SwingUtilities.invokeLater(new Runnable() {
```

```
            public void run() {
                new EventDemo();
            }
        });
    }
}
```

6(a)	What is JPanel class?Explain the constructors of JPanel class and give a suitable example.
	
	JPanel is a **generic lightweight container** provided by the Java Swing library. It is used to **group components** together inside a GUI and helps in organizing the layout of the components. It **does not have a title bar or border** by default.
	JPanel is often used inside a JFrame or another container and can have its own layout manager.
	🔷 **Constructors of** JPanel**:**
	
	1. JPanel()
	➤ Creates a new panel with a **flow layout** by default.
	➤ *Example:* JPanel panel1 = new JPanel();
	2. JPanel(LayoutManager layout)
	➤ Creates a panel with the **specified layout manager**.
	➤ *Example:* JPanel panel2 = new JPanel(new GridLayout(2, 2));
	3. JPanel(boolean isDoubleBuffered)
	➤ Creates a panel with or without **double buffering** to reduce flickering.
	➤ *Example:* JPanel panel3 = new JPanel(true);
	4. JPanel(LayoutManager layout, boolean isDoubleBuffered)
	➤ Creates a panel with both **specified layout** and **double buffering option**.
	➤ *Example:* JPanel panel4 = new JPanel(new BorderLayout(), true);
	
	`import javax.swing.*;`
	`import java.awt.*;`
	`public class JPanelExample extends JFrame {`
	`  public JPanelExample() {`
	`  setTitle("JPanel Example");`
	`  setSize(300, 200);`
	`  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
	`  // Create a panel with FlowLayout`
	`  JPanel panel = new JPanel();`
	
	`  // Add buttons to the panel`
	`  panel.add(new JButton("Button 1"));`
	`  panel.add(new JButton("Button 2"));`
	
	`  // Add panel to the frame`
	`  add(panel);`
	`  setVisible(true);`
	`  }`
	`  public static void main(String[] args) {`
	`  SwingUtilities.invokeLater(() -> new JPanelExample());`
	`  }`
	`}`
		[06]	CO3	L2

6(b)	What is JCheckBox class? Explain the constructors of JCheckBox class and give a suitable example.
	
	JCheckBox is a **Swing component** used to create a **two-state button** (checked or unchecked).
	It allows the user to **select or deselect multiple independent options**.
	It extends JToggleButton and generates an ItemEvent when its state changes.
	Constructors of JCheckBox **class:**
	
	1. JCheckBox()
	➤ Creates an **unchecked** checkbox with **no label**.
	➤ Example: JCheckBox cb1 = new JCheckBox();
	2. JCheckBox(String text)
	➤ Creates an **unchecked** checkbox with the specified **label**.
	➤ Example: JCheckBox cb2 = new JCheckBox("Java");
	3. JCheckBox(String text, boolean selected)
	➤ Creates a checkbox with the given label and **selected state**.
	➤ Example: JCheckBox cb3 = new JCheckBox("C++", true);
	4. JCheckBox(Icon icon) *(rarely used)*
	➤ Creates a checkbox with an icon but no text.
	➤ Example: JCheckBox cb4 = new JCheckBox(myIcon);
	5. JCheckBox(String text, Icon icon)
	➤ Creates a checkbox with both **label** and **icon**, unchecked by default.
	6. JCheckBox(String text, Icon icon, boolean selected)
	➤ Creates a checkbox with label, icon, and selected state.
	
	Example Program:
	`import javax.swing.*;`
	`import java.awt.*;`
	`public class JCheckBoxExample extends JFrame {`
	`  public JCheckBoxExample() {`
	`  setTitle("JCheckBox Example");`
	`  setSize(300, 200);`
	`  setDefaultCloseOperation(EXIT_ON_CLOSE);`
	`  setLayout(new FlowLayout());`
	`  // Create checkboxes`
	`  JCheckBox cb1 = new JCheckBox("Java");`
	`  JCheckBox cb2 = new JCheckBox("Python", true); // pre-selected`
	`  JCheckBox cb3 = new JCheckBox("C++");`
	`  // Add to frame`
	`  add(cb1);`
	`  add(cb2);`
	`  add(cb3);`
	`  setVisible(true);`
	`  }`
	`  public static void main(String[] args) {`
	`  SwingUtilities.invokeLater(() -> new JCheckBoxExample());`
	`  }`
	`}`
		[07]	CO3	L2
				
				L2

6(c)	What is JFrame class?Explain constructors and methods of JFrame class.	[07]	CO3	

JFrame is a **top-level container** used in Swing applications to create a GUI window. It provides a standard window with a title bar, borders, and control buttons like minimize, maximize, and close. JFrame is commonly used to build desktop applications using Swing.

Constructors of JFrame**:**

JFrame(String title)

- Creates a new JFrame with the specified title.

JFrame jfrm = new JFrame("A Simple Swing Application");
Other commonly used constructors include:

- JFrame() – Creates a frame with no title.
- JFrame(GraphicsConfiguration gc) – With graphics config.
- JFrame(String title, GraphicsConfiguration gc) – With title and graphics config.

Common Methods of JFrame **(based on explanation):**

Method	**Description**
setSize(int width, int height)	Sets the width and height of the window in pixels.
setDefaultCloseOperation(int)	Defines what happens when the close button is clicked (e.g., exit app).
add(Component c)	Adds a component (like a label) to the JFrame's content pane.
setVisible(boolean b)	Displays the frame when set to true.
getContentPane()	Returns the content pane (older approach before Java 5).

Example:

```
import javax.swing.*;
class SwingDemo {
  SwingDemo() {
    JFrame jfrm = new JFrame("A Simple Swing Application");
    jfrm.setSize(275, 100);
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JLabel jlab = new JLabel("Swing means powerful GUIs.");
    jfrm.add(jlab);
    jfrm.setVisible(true);
  }
  public static void main(String args[]) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new SwingDemo();
      }
    });
  }
}
```

L2

7(a)	Explain the life cycle of Servlet **Life Cycle of a Servlet** The **life cycle of a servlet** is managed by the **web container** (e.g., Apache Tomcat). It involves **three main methods**: init(), service(), and destroy(). These methods define how a servlet is **initialized**, **handles client requests**, and is finally **destroyed**. **Steps in Servlet Life Cycle:** 1. **Loading and Instantiation** When a client sends a request to a servlet for the first time, the web container **loads** the servlet class and **creates an instance** of it. 2. **Initialization (**init()**)** The web container calls the init(ServletConfig config) method **once**, just after instantiation. This method is used for **initial setup**, like reading initialization parameters. **public void init(ServletConfig config) throws ServletException** 3.Request Handling (service()) Each time the servlet receives a client request, the service(ServletRequest req, ServletResponse res) method is called. This method is responsible for **processing client requests** and **generating responses**. This method is called **multiple times**—once for each client request. **public void service(ServletRequest req, ServletResponse res)** **throws ServletException, IOException** **4.Destruction (destroy())** When the servlet is no longer needed or the server is shutting down, the web container calls the destroy() method. This is where you release resources (like DB connections, file handles, etc.). **public void destroy()** **Diagram: Life Cycle of a Servlet** **+----------+** **| Client sends request |** **+----------+----------+** **|** **v** **+----------+** **| Servlet class is loaded |** **+----------+** **|** **v** **+----------+** **| Servlet instance created |** **+----------+** **|** **v** **+----------+**	[06]	CO4	

	| init() is called once | **+----------+** **|** **v** **+----------+** **| service() called for each |** **| request from the client |** **+----------+** **|** **v** **+----------+** **| destroy() called once when |** **| servlet is taken out |** **+----------+**			

7(b)	Write a java Servlet program to display the name,USN, and total marks by accepting student details.
	
	Servlet Code:
	package com.example;
	import java.io.IOException;
	import javax.servlet.ServletException;
	import javax.servlet.annotation.WebServlet;
	import javax.servlet.http.*;
	@WebServlet("/login")
	public class Addservlet extends HttpServlet {
	private static final long *serialVersionUID* = 1L;
	protected void doPost(HttpServletRequest request, HttpServletResponse response)
	throws ServletException, IOException {
	try {
	String name = request.getParameter("name");
	String usn = request.getParameter("usn");
	float mrks1 = Float.*parseFloat*(request.getParameter("mrks1"));
	float mrks2 = Float.*parseFloat*(request.getParameter("mrks2"));
	float total = mrks1 + mrks2;
	HttpSession session = request.getSession();
	session.setAttribute("name", name);
	session.setAttribute("usn", usn);
	session.setAttribute("total", total);
	response.sendRedirect("login.jsp");
	} catch (Exception e) {
	e.printStackTrace();
	}
	}
	}
		[07]	CO4	L3

.HTML file–

```
<!DOCTYPE html>
<html>
<head>
  <title>Student Details Form</title>
</head>
<body>
  <h2>Student Details Form</h2>
  <form action="login" method="post">
    <label>Name:</label>
    <input type="text" name="name" required><br><br>
    <label>USN Number:</label>
    <input type="text" name="usn" required><br><br>
    <label>Marks in Subject 1:</label>
    <input type="number" name="mrks1" step="0.01" required><br><br>
    <label>Marks in Subject 2:</label>
    <input type="number" name="mrks2" step="0.01" required><br><br>
    <input type="submit" value="Submit">
  </form>
</body>
</html>
```

.jsp file–

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
  <title>Student Details</title>
</head>
<body>
  <h2>Student Details</h2>
<%
  String name = (String) session.getAttribute("name");
  String usn = (String) session.getAttribute("usn");
  Float total = (Float) session.getAttribute("total");
  if (name != null && usn != null && total != null) {
%>
  <p><strong>Name:</strong> <%= name %></p>
  <p><strong>USN Number:</strong> <%= usn %></p>
  <p><strong>Total Marks:</strong> <%= total %></p>
<%
  } else {
%>
  <p>No student data found.</p>
<%
  }
%>
</body>
</html>
```

7(c)	Describe the core interfaces that are provided in Javax.Servlet.http package. **Core Interfaces in javax.servlet.http Package:** 1. HttpServletRequest ● Extends ServletRequest. ● Enables a servlet to read data from an HTTP request. ● Provides methods like getMethod(), getHeader(), getParameter(), getCookies(), etc. 2. HttpServletResponse ● Extends ServletResponse. ● Enables a servlet to build and send an HTTP response. ● Provides methods like addCookie(), sendError(), setStatus(), etc. 3. HttpSession ● Allows reading and writing session-related data. ● Enables the server to track a user's session using getSession().	[07]	CO4	L3

8(a)	What is JSP? Explain the various types of JSP Tags with example. JSP (JavaServer Pages) is a server-side program that is similar in design and functionality to a Java servlet. A JSP is called by a client to provide a web service. It processes the request using built-in logic or by calling other Java web components like servlets. Once the request is processed, the JSP sends the result back to the client. Unlike Java servlets, which are written entirely in Java code, JSP pages are mostly written using HTML, XML, or other client-side formats mixed with JSP tags and Java code. **Types of JSP Tags:** **1. Directive Tag (<%@ ... %>)** ● Provides global information to the JSP engine. ● **Example:** <%@ page import="java.util.Vector" %> <%@ include file="header.html" %> **2. Declaration Tag (<%! ... %>)** ● Used to declare variables, methods, arrays, or objects. ● **Example:** <%! int age = 29; %> **3. Scriptlet Tag (<% ... %>)** ● Used for writing Java code such as loops or control statements. ● **Example:** <% if (age > 18) { %> You are an adult. <% } %>	[10]	CO4	L2

4. Expression Tag (<%= ... %>)

- Outputs the result of an expression to the client.
- **Example:**

```
<p>Your age is: <%= age %></p>
```

Complete Program to Demonstrate All JSP Tags:

```
<%@ page import="java.util.*" %>
<html>
<head>
  <title>JSP Tag Demonstration</title>
</head>
<body>

<%-- Declaration Tag --%>
<%!
  int studentId= 101;
  String studentName = "Alice";
  int[] marks = {85, 78, 92};

  float calculateAverage(int[] scores) {
    int sum = 0;
    for (int score : scores)
      sum += score;
    return sum / (float) scores.length;
  }
%>

<h2>Student Info</h2>
<p>Student ID: <%= studentId %></p>   <%-- Expression Tag --%>
<p>Name: <%= studentName %></p>

<%-- Scriptlet Tag --%>
<%
  float average = calculateAverage(marks);
  String result;
  if (average >= 80) {
    result = "Distinction";
  } else if (average >= 60) {
    result = "First Class";
  } else {
    result = "Needs Improvement";
  }
%>

<p>Average Marks: <%= average %></p>
<p>Result: <%= result %></p>
</body>
</html>
```

	Expected output in browser– Student Info Student ID: 101 Name: Alice Average Marks: 85.0 Result: Distinction			

8(b)	**What are cookies? How are cookies handled in JSP? Write a JSP program to create and read a cookie**
	A **cookie** is a small piece of information stored on the **client's browser** by the server. It is used to maintain the **state of the user** (e.g., user preferences, login info) between multiple HTTP requests, which are stateless by default.
	How Cookies Are Handled in JSP?
	In JSP, cookies can be created, sent to the client, and retrieved using the javax.servlet.http.Cookie class.
	● **Creating and Sending a Cookie**: Use Cookie class and response.addCookie() method.
	● **Reading a Cookie**: Use request.getCookies() method to get all cookies sent by the client.
	JSP Program to Create and Read Cookie
	1. File: CreateCookie.jsp
	<%@ page language="java" %>
	<html>
	<body>
	<%
	String name = "user";
	String value = "Alice";
	Cookie cookie = new Cookie(name, value);
	cookie.setMaxAge(60*60); // cookie will exist for 1 hour
	response.addCookie(cookie);
	%>
	 Cookie named user with value Alice has been set.
	Click here to read the cookie
	</body>
	</html>
	
	2. File: ReadCookie.jsp
	<%@ page language="java" %>
	<html>
	<body>
	<%
	Cookie[] cookies = request.getCookies();
	if (cookies != null) {
	for (int i = 0; i < cookies.length; i++) {
	if (cookies[i].getName().equals("user")) {
	out.println(" Cookie Found - Name: " + cookies[i].getName() +
	", Value: " + cookies[i].getValue() + " ");
		[10]	CO4	L2

```
        }
      }
    } else {
      out.println("<p>No cookies found!</p>");
    }
%>
</body>
</html>
```

9(a) What are database drivers? Explain the different JDBC driver types. [10] CO5 L2

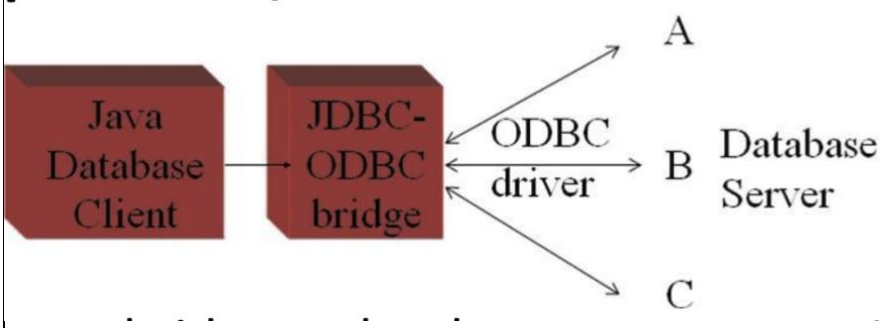
A **database driver** is a software component that enables Java applications to interact with a database. JDBC (Java Database Connectivity) API uses these drivers to send SQL commands to the database and retrieve results.

JDBC drivers implement the interfaces defined in java.sql package and are essential for establishing a **database connection**, **executing queries**, and **retrieving results**.

There are four types of JDBC drivers, classified based on how they interact with the database:

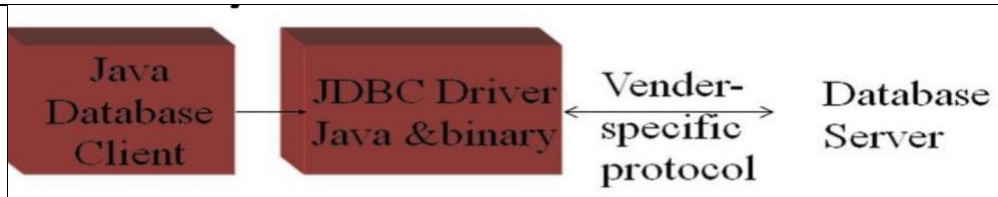
Type 1: JDBC-ODBC Bridge Driver

- **Example:** Sun's JDBC-ODBC Bridge.
- This driver uses **bridge technology** to connect a Java client to a third-party API such as **ODBC (Open Database Connectivity)**.
- It is dependent on the ODBC driver and works through native libraries.



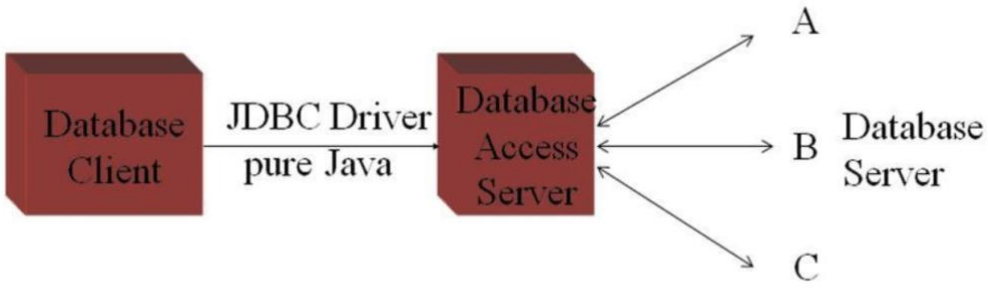
Type 2: Java/Native Code Driver

- **Example:** Oracle Call Interface (OCI) driver.
- This type of driver **wraps a native API** (written in languages like C/C++) with Java classes.
- It requires **native database libraries** to be installed on the client machine.



Type 3: Network Protocol/All-Java Driver

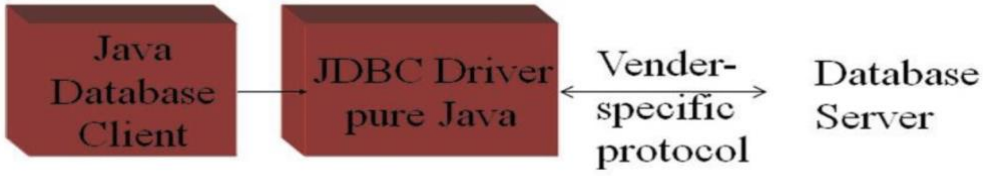
- This driver uses a **network protocol** to communicate with a **middle-tier server**.
- The **middle tier** then communicates with the database.
- Useful for internet-based applications where the JDBC client and database server are not directly connected.



Type 4: Native Protocol/All-Java Driver

- This driver translates **JDBC calls directly into database-specific network protocols**.
- Java programs can connect **directly to the database** without any native code or middleware.
- It is **written entirely in Java** and is platform-independent.

Driver



9(b)	Describe the various steps of JDBC with code snippets.	[10]	CO5	L2

Java Database Connectivity (JDBC) enables Java applications to interact with relational databases. The JDBC process follows a standard series of steps:

1. Load the JDBC Driver

The JDBC driver must be loaded before establishing a connection. This is done using:

Class.forName("com.mysql.jdbc.Driver");

This step may throw a ClassNotFoundException if the driver is not found.

2. Connect to the DBMS

Use the DriverManager.getConnection() method to establish a connection to the database:

Connection con = DriverManager.getConnection("jdbc:mysql://localhost/TEST", "root", "password");

- "jdbc:mysql://localhost/TEST" is the database URL.
- "root" is the username.
- "password" is the password.

3. Create and Execute a Statement

Once connected, create a Statement object to send SQL queries:

Statement stmt = con.createStatement(); ResultSet rs = stmt.executeQuery("SELECT * FROM Employees");

The executeQuery() method returns a ResultSet object containing the results.

4. Process the Result

Use the ResultSet object to iterate through the data returned by the query:

while(rs.next()) { System.out.println(rs.getInt(1) + " " + rs.getString(2) + " " + rs.getString(3) + " " + rs.getString(4)); }

Or using column names:

do { String name = rs.getString("name"); int age = rs.getInt("age");
System.out.println(name + " -- " + age); } while(rs.next());

5. Terminate the Connection

Once done, close the connection to free up resources:

con.close();

10(a)	Write any two syntax of established a connection to a database. **Establishing Connection in JDBC** In JDBC (Java Database Connectivity), establishing a connection to the database is the first and most essential step. The DriverManager class is used to manage a set of JDBC drivers. It provides methods to establish a connection between a Java application and the database. **Syntax 1: Using Only URL** Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/yourDatabase"); ● This method is used when no username and password are required or are included in the URL. **Syntax 2: Using URL with Username and Password** Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/yourDatabase", "root", "password"); ● This is the most commonly used syntax. ● The url specifies the database location. ● "root" is the username and "password" is the login password for the database.	[6]	CO5	L2

10(b)	What is connection pooling?Explain connection pooling with a neat diagram with snippets **Connection pooling** is a technique used to **optimize database access** in Java EE applications. Instead of creating a new connection every time a client accesses the database (which is resource-heavy and slow), a pool of pre-created **physical connections** is maintained and **reused** across multiple clients. **Why is it needed?** ● Opening and closing connections repeatedly is **slow and resource-consuming**. ● A database can handle only a **limited number** of concurrent connections. ● Connection pooling **improves performance** and **reduces overhead**. **How it works:** There are **two types of connections**: 1. **Physical Connection** – Created by the application server and held by PooledConnection objects. 2. **Logical Connection** – Given to the client via DataSource.getConnection() and maps internally to a physical one. When the client calls .close(), only the **logical connection** is closed. The **physical connection** stays in the pool and is reused. **Code Snippet (Connection Pooling Example):** Context ctext = new InitialContext(); DataSource pool = (DataSource) ctext.lookup("java:comp/env/jdbc/pool"); Connection db = pool.getConnection(); // Interact with the database	[07]	CO5	L2

	// ... db.close(); // Only logical connection is closed **Diagram (Text Representation):** +--------------------------+ | Connection Pool | | (Managed by App Server) | +--------------------------+ | | | [P1] [P2] [P3] <- Physical connections (PooledConnection objects) Client 1 ---> LogicalConnection --> P1 Client 2 ---> LogicalConnection --> P2 Client 3 ---> LogicalConnection --> P3 When db.close() is called by Client 1, LogicalConnection is closed, but P1 goes back to pool and can be reused by another client. **Key Interfaces Used:** ● javax.sql.DataSource – Used to retrieve connections. ● javax.naming.Context – Used to perform JNDI lookup. **Advantages of Connection Pooling:** ● Reduces connection overhead ● Improves response time ● Controls maximum DB connections ● Ensures better scalability in enterprise apps			

10(c)	Describe the following concepts: i)Callable Statements ii)Transaction Processing i)Callable Statements: **CallableStatement Object** ● CallableStatement is used to call stored procedures from a Java application. ● Stored procedures are precompiled blocks of SQL (written in PL/SQL, T-SQL, etc.) stored in the database. ● CallableStatement supports IN, OUT, and INOUT parameters. ● **It is created using:** **CallableStatement cs = con.prepareCall("{call procedure_name(?, ?, ?)}");** ● IN Parameter – Data passed into the procedure using setXxx() methods. ● OUT Parameter – Data returned from the procedure using registerOutParameter().	[07]	CO5	L2

- INOUT Parameter – Used both for sending and retrieving values.

Example use-case: Getting employee salary based on ID from a stored procedure.

ii)Transaction Processing:

Transaction processing in JDBC refers to executing a group of SQL operations as a single unit. These operations either all succeed (commit) or all fail (rollback) to maintain data consistency and integrity.

Default Behavior in JDBC:
By default, JDBC connections are in auto-commit mode, which means each SQL statement is committed immediately after it is executed.

```
Connection con = DriverManager.getConnection(...);
System.out.println(con.getAutoCommit()); // returns true
```

Steps for Transaction Processing:

1. **Disable auto-commit:**

```
con.setAutoCommit(false);
```

2.Execute multiple SQL statements:

```
Statement stmt = con.createStatement();
stmt.executeUpdate("UPDATE Account SET balance = balance - 1000 WHERE acc_no =
101");
stmt.executeUpdate("UPDATE Account SET balance = balance + 1000 WHERE acc_no =
102");
```

3.Commit the transaction:

```
con.commit();
```

4.Handle exceptions and rollback if any failure occurs:

```
catch(Exception e) {
   con.rollback();  // Undo all changes
   System.out.println("Transaction failed. Rolled back.");
}
```

Complete Example:

```
try {
   Connection con = DriverManager.getConnection(...);
   con.setAutoCommit(false); // Begin transaction
   Statement stmt = con.createStatement();
   stmt.executeUpdate("UPDATE Account SET balance = balance - 1000 WHERE acc_no =
101");
   stmt.executeUpdate("UPDATE Account SET balance = balance + 1000 WHERE acc_no =
102");
   con.commit(); // Commit transaction
   System.out.println("Transaction successful!");
} catch (Exception e) {
   con.rollback(); // Rollback transaction on error
   System.out.println("Transaction failed! Changes rolled back.");
}
```

Advantages of JDBC Transaction Processing:

	• Maintains data consistency • Prevents partial updates • Useful in banking, booking systems, etc			