classmate
Date
Page 1

⇒ OOPs Regolar Exam

⇒ PPT - Lecture 0

1 Programming Paradigms:-

- Procedural Paradigm (POP) - Code is structured around procedures or functions.
- object-Oriented Paradigm (OOP)- Focus on objects representing real-world entities.
- Functional Paradigm- Uses functions to solve problems (e.g Python, Haskell).
- Logical Paradigm- Based on formal logic.
- Structured Paradigm- Uses a clear, top-down flow with loops and conditionals.

2 OOP's Six Main Pillars:-

Ans a Class - A blueprint/template for objects.
b Objects & Methods - Instance of a class with properties and behaviors.
c Interface- A Contract for what a class can do but doesn't implement methods.
d Polymorphism- Methods or objects taking multiple forms.
e Inheritance - A mechanism to derive new classes from existing ones.
f Abstraction- Hiding Complex logic and showing essential details.

9 Encapsulation- wrapping data and methods to protect the internal State. →

3 Relationships in OOP

Ans Has-A (Composition) - one class contains another (e.g Car has a Engine)
. Is-A (Inheritance) - one class is a type of another (e.g Dog is an Animal)

★ Procedural Programming vs OOP

→ POP Example:

```
# include < stdio.h>
 int main()
{
 int arr[5], i;
 for(i=0; i<5; i++)
  {
     scanf("%d", &arr[i]);
  }
 for(i=0; i<5 ; i++)
{
    Printf("%d", arr[i]);
}
 return 0;
}
```

An

. This is simple array input / output in C.

→ OOP Example :

```
class Employee {
    int id;
    String name;

    public static void main (String [] args) {
        Employee obj = new Employee();
        obj.id = 101;
        obj.name = "John";
        S.o.p( obj.id + " " + obj.name);
    }
}
```

4 Class and object in Java

Ans Class Syntax

```
class className {
    int data; // data member
    void display() { // Method
        S.o.p("Hello");
    }
    public static void main (String [] args) {
        className obj = new className();
        obj.display();
    }
}
```

∴ The main method creates an object and calls the display method.

5 Object Creation

Ans Three ways to initialize:-

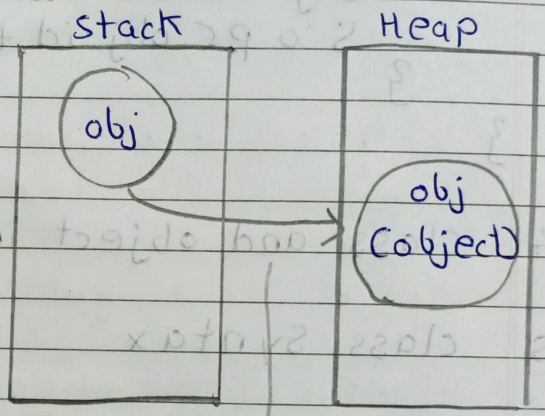
a Reference Variable :- default constructor of class

reference variable

=> Employee obj = new Employee();

classname new keyword

obj.id = 101;



A variable that points to the memory address of an object, allowing you to access and manipulate the object's data and behavior.

b Constructor

=>
```
class Employee {
    Employee() {
        S.o.p ("Constructor Called");
    }
```

c Function:-

→
```
void setData (int id, String name)
{
    this.id = id;
    this.name = name;
}
```

6 Inheritance

Ans Code:-

```
class Animal {
    void sound ()
    {
        S.o.p (" Animal Sound ");
    }
}
class Dog extends Animal {
    void Sound()
    {
        S.o.p (" Bark ");
    }
public static void main (String [] args)
{
    Dog d = new Dog();
    d.sound();
    }
}
```

→ Dog inherits from Animal and overrides the sound method.

7 Polymorphism (Method Overloading and Overriding)

Ans Method Overloading :-

→
```
class MathOperation {
    int add (int a, int b)
    {
        return a+b;
    }
    double add ( double a, double b)
    {
        return a+b;
    }
}
```

→ Method Overriding :-

```
class Parent {
    void show () {
        System.out.println ("Parent");
    }
}
class child extends Parent {
    void show() {
        s.o.p ("child");
    }
}
```

8 Encapsolation

Ans
```
class student {
    private string name;

    public string getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

→ The student class restricts direct access to the name property using getter and setter Methods.

9 Abstraction (Abstract classes and Interfaces).

Ans. Abstract class Example

-
```
abstract class shape {
    abstract void draw();
}
class Circle extends shape {
    void draw()
    {
        S.o.p("Drawing Circle");
    }
}
```

Q Interface example

⇒
```
interface Animal{
    void eat();
}
class Dog implements Animal {
    public void eat()
    {
        S.o.p("Dog eats");
    }
}
```

Q Constructors in Java:-

Ans Default Constructor:-

```
class Car {
    Car(){
        S.o.p("Car created");
    }
}
```

⇒ Parameterized Constructor:-

```
class Car {
    Car(String model)
    {
        S.o.p("Car:"+model);
    }
}
```

11 this and super key words:-

Ans this example

⇒
```
class Employee
{
    int id;
    Employee (int id)
    {
        this.id = id;
    }
}
```

b Super Example

⇒
```
class Animal
{
    String Color = "white";
}
class Dog extends Animal
{
    String Color = "Black";
    void printColor()
    {
        S.o.p(super.color);
    }
}
```

12 Difference Between Java and C++

Java	C++
· Patform Independent	· Platform Dependent.
· No goto	· Supports goto
· No Multiple Inheritance (via interface)	· Supports Multiple Inheritance.
· Automatic Garbage Collection.	· Manual memory management.
· Strictly object-oriented	· Supports both OOP and POP.

Research it

→ Lecture - 1 (Object Oriented features)

1 Abstraction

Ans Abstraction hides complex implementation details and shows only essential information to the user

· Achieved using:-

1 Abstract classes / Methods (0-100%)
2 Interfaces (100%)

↳ Example

```
class University
{
   String CollegeName;
   String Department;
   String HOD;

   protected void info ()
   {
      CollegeName = "Lords College";
      Department = "Electronics";
      HOD = "Mr. Singh";
      S.o.p ("Collegename +" " + Department +" "
                                   + HOD);
   }
}
class College extends University {
    @Override
    public void info()
{
    CollegeName = "Vivek College";
    Department = "Telecommunication";
    HOD = "Mr. Das";
    S.o.p (CollegeName +" " + Department
                 +" " + HOD);
   }
}
class Admission extends University{
   @Override
   public void info()
{
   CollegeName = "Ruia College";
```

```
Department = " ICT";
HOD = " Mr. Parlekar";
S.o.p (CollegeName + " " + Department + " " +
                    HOD);
}
}
public class AbstractionProg1 {
    public static void main (String[] args)
{
    College obj = new College();
    Admission obj2 = new Admission();
     obj.info();
     obj2.info();
```

2 Abstract classes and Methods

Ans Abstract class - Defined with the abstract keyword. Can have both abstract (no body) and normal methods.

. Abstract Method - Declared but not implemented in the abstract class. Sub classes must be override it.

↳ Example :-

```
abstract class Animal {
    public abstract void sound();
}

class Tiger extends Animal {
    public void sound() {
        S.o.p("Tiger makes sound");
}
}
```

← abstract class & Method

← Creating body of sound method.

```
public class AbstractExample{
    public static void main (String args[])
{
    Tiger ob = new Tiger();
      ob.sound();
  }
}
```

3 Interfaces

Ans Interface- Blue print for a class. Contains abstract methods that a class must implement. Supports multiple inheritance since a class can implement multiple interfaces

↳ Example :-

```
interface Person{
    int id = 1001;
    String name = "Sumit";
    String address= "Jhalte";
}
public class Department implements Person{
    public static void main (String args[])
{
    S.o.p (id +" "+name+" "+address);
  }
}
```

4 Encapsolation

Ans Encapsolation binds data and methods together while keeping the data hidden from outside Access.

- Private variables- can be accessed through public getter and setter methods.

↳ Example :-

```
class Employee {
    private int emp_id;

    public void setEmp_ID (int emp_id){
        emp_id = emp_id1;
    }

    public int getEmp-ID () {
        return emp_id;
    }
}
public class Company {
    public static void main (String [] args)
    {
        Employee e = new Employee ();
        e. set Emp_id (101);
        S.o.p (e. getEmp_Id ());
    }
}
```

5 Polymorphism

Ans Compile - Time Polymorphism - Achieved via method overloading.
Run- time Polymorphism - Achieved through method overriding.

↳ (overloading):-

```
class AddValue {
    void add (int a, int b) {
        S.o.p ("Addition :" + (a+b));
    }
    void add (double a, double b) {
        S.o.p ("Addition :" + (a+b));
    }
    public static void main (String args[])
    {
        AddValue obj = new AddValue();
        obj.add (5, 10);
        obj.add (5.5, 10.5);
    }
}
```

used pass command line argument to program

↳ (overriding):-

```
class FOverriding Test {
    void show() {
        S.o.p ("I am in test in class");
    }
}
class FOverriding extends Test {
```

```
void show(){
    S.o.p("I am in XYZ class");
}
public static void main(String[] args){
    Test t = new Test();
    t.show();
    Foverriding t1 = new Foverriding();
    t1.show();
  }
}
```

6 Inheritance

Ans Allows reuse of code by creating new classes from existing ones.

- Types of Inheritance

- Single level
- Multi - level
- Hierarchical
- Hybrid.

↳ Example of Single level

```
class Ram{
    int a, b;

    void accept(){
        a = 10;
        b = 20;
    }
}
```

```
class shyam extends Ram {
    void sum() {
        S.o.p(" Addition= " + (a+b));
    }

public static void main (String args[])
{
    shyam ob1 = new shyam();
      ob1. accept();
      ob1 . sum();
  }
}
```

↳ Lecture - 4 Core Java String

1 String in Java

Ans String is a sequence of characters (like an array of char).
- It is non primitive datatype.
- Immutable - Once created, it cannot be modified.

↳ Created using:

```
char [] c = {'a','b','c','d'};   // array of character c.
String s = new string (c);
```

↳ because String is a class in java so we have to create object.

↳ How string is stored:-

- without new → stored in String Constant Pool (SCP).
- with new → stored in heap and SCP.

↳ Example of Immutability

```
class T
{
   public static void main (String [] args)
{
    String s = new String ("welcome");
     s.concat ("Software");
      S.o.p(s);
  }
}
```

→ output :-

welcome ← We can't modified it.

★ Comparision of Strings

- Use equals() for content comparision, not ==.

→
```
class T {
   public static void main (String [] args) {
       String s1 = new string ("Raj");
       string s2 = new string ("Raj");
```

```
        S.o.p (S1 == S2) // False
        S.o.p (S1.equals(S2)); // True
    }
}
```

→ Output :-

```
False
True
```

2 String Methods

Ans a isEmpty();
→ checks if the string is empty.

∴ Code

```
class StringisEmptyfonction{
    public static void main(String [] args)
{
        String s= " ";
        S.o.p ( S.isEmpty());// True
    }
}
```

b length();
→ Returns the length of the String.

∴ Code :-

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

```
String s=" Hello world";
int len = s.length();
S.o.p ("Length:" + len); //11
 }
}
```

c replace();

→ Replaces characters in a string.

∴ Code:-

```
class StringreplaceFunction {
   public static void main (String [] args)
{
      string s=" welcome";
      S.o.p (s. replace ('e', 'o'));
  }
}
```

→ Output:-

wolcomo

d substring();

→ Extracts part of the string.

∴ Code:-

```
class Stringsubstring Function {
   public static void main (String [] args)
{
   String s=" welcome to university";
```

```
        S.o.p (s.substring(5));
        S.o.p(s.substring(2,4));
    }
}
```

→ Output :-

me to university
lc

e indexof()
→ Finds the index of the first occurence
→ of a character

∴ Code :-

```
class Stringindex of function{
    public static void main(String []args){
        string s= "welcome to university";
        s.o.p(s.indexof('i')); //13
    }
}
```

f toLowerCase() / toUpperCase():
→ Converts the string to lowercase or upper-
case.

∴ Code :-

```
class String Case function{
    public static void main(String []args){
        string s = "welcome";
```

```
        S.o.p (s.toUpperCase());
    }
}
```

9 trim():

→ Removes leading and trailing spaces.

→ Code :-

```
class StringTrimFunction {
    public static void main (String[] args) {
        String s = " Welcome ";
        S.o.p (s.trim());
    }
}
```

3 String Buffer class

Ans Mutable - Can change the value of object.
- Thread-Safe (synchronized).
- Faster than String for frequent modifications.

↳ Code :-

```
class T {
    public static void main (String[] args) {
        StringBuffer s = new StringBuffer ("Welcome
                                to Java");
        s.append (" Software");
        S.o.p (s);
    }
}
```

→ Output :-

welcome to Java Software.

4 String Buffer Methods

Ans replace() :-

→ Code :-

```
class replaceFunction {
    public static void main(String[] args)
{
    StringBuffer s = new StringBuffer("welcome");
    S.o.p(s.replace(0,6,"");
    S.o.p(S.replace(0,6,"Hello");
  }
}
```

6 reverse()

→

```
class reverseFunction {
    public static void main( String[] args)
{
    StringBuffer s= new StringBuffer ("Hi");
     S.o.p(s.reverse());
  }
}
```

5 String Builder class

Ans Same as String Buffer but not thread-safe.
Faster for single-threaded enviornments

↳ Example

```
class StringBuilderExample{
    public static void main(String[] args)
{
    StringBuilder str= new StringBuilder
    ("welcome");
    S.o.p(str.append(" University"));
    }
}
```

↳ Comparision

Feature	String	String Buffer	String Builder
Mutable/ Immutable	Immutable	Mutable	Mutable
Thread-Safe	Yes (constant)	Yes (Synchronized)	No
Performance	Slow	Fast	Fast
Usecase	Fixed data	Multi-threading	Single-threading

↳ Arrays class Notes

1 Arrays class in Java

Ans The Arrays class is part of the java.util package in Java.

- It provides several static methods to perform operations on arrays like:-
  - Sorting
  - Searching
  - Comparing
  - Filling
  - Converting arrays to Strings.

2 Key Methods in Arrays class

a Sorting Arrays-

- Arrays.sort() sorts elements in ascending order.

↳ Example

```
import java.util.Arrays;

class ArraysSortExample {
    public static void main(String[] args){
        int[] arr = {10, 8, 7, 2, 1};
        S.o.p("Array elements before sorting:");
        for(int i : arr)
```

```
{
    S.o.pC i);
}
Arrays. sort carr);

S.o.p c" Arrays elements after sorting");
  for cint i : arr)
{
    S.o.p Ci);
}
}
}
```

★ Sorting character Arrays

→ Code :-

```
import java.util. Arrays;

class ArraysSortcharEg {
    public static void main (String [] args){
        char [] arr= { 's', 'r', 't', 'f', 'c', 'a' };
        S.o.p c" Array elements before sorting");
        for (char c : arr)
        {
            S.o.pCc);
        }
        Arrays. sort carr);

        S.o.p c" Array elements after sorting");
        for (charc : arr) {
            S.o.pCc);
        }
    }
}
```

* Sorting with a range (Partial sort)

→ Code:-

```
import java.util.Arrays;

class ArraysPartialSortExample{
    public static void main(String[]args)
{
    int[] arr={10,8,7,2,1,5,3,4,11};
    S.o.p("Array elements before sorting");
    for(int i: arr){
        S.o.p(i);
    }
    Arrays.sort(arr,2,5) // Sorts from index 2
                          to 4 (exclusive of 5)
    S.o.p("Array elements after Sorting");
    for(int i: arr){
        S.o.p(i);
    }
}
}
```

2 Searching Arrays:-

Ans Arrays.binarySearch() performs a binary search on sorted arrays.

→ Code:-

```
import java.util.Arrays;
```

```
class Arrays Binary Search e.g {
    public static void main (String [] args)
{
    int [] arr = { 10, 8, 7, 2, 1, 5, 3, 4, 11};
     Arrays. sort (arr);
     S. o. p (" Sorted array");
      for (int i : arr) {
          S.o.p (i + " ");
      }
      S. o. p ();

   S.o.p (" Found at Index:" + Arrays. binary
                              Search (arr, 7));
   S.o.p (" Not found :" + Arrays . binarySearch
                              (arr, 99)); // Returns
                              -1 if not found.
    }
}
```

3 Converting Arrays to strings:-

Ans Arrays. to String () converts an array into a readable string.

→ Code:-

```
import java. util. Arrays;

class Arrays To String Eg {
    public static void main (String [] args) {
        String [] arr = {" Mukund", " Soman",
            " Rovika", " Trish", " Vicky"};
```

```
        S.o.p ("Array toString: " + Arrays.toString
                                    (arr));
    }
}
```

4 Filling Arrays

Ans Arrays.fill() fills the entire array with a specific value.

↳ Filling Example:-

```
import java.util.Arrays;

class ArraysFilleg {
    public static void main (String [] args) {
        String [] arr = {"Mukund", "Soman",
                         "Ruvika"};
        Arrays.fill (arr, "Bhavin");
        S.o.p ("Filled array:" + Arrays.toString
                                    (arr));
    }
}
```

⇒ Output:-

Filled Array: [Bhavin, Bhavin, Bhavin]

↳ Partial Fill example:-

```
import java. util. Arrays;

class Arrays PartialFillexample{
    public static void main ( String [] args)
{
    String. [] arr = {"Mukund", "Soman", "Ruvika",
                      "Krish"};
    Arrays. fill (arr , 1 , 3 , "John");
    S.o.p ("Partially Filled Array:" + Arrays.
                      toString (arr));
    }
}
```

-> Output :-

```
Partially Filled Arrays : [Mukund, John, John,
                           Krish]
```

5 Copying Arrays:-

Ans Arrays. copyof () and Arrays. copyOfRange() create copies of arrays.

↳ Copying Entire Array:-

```
import java. util. Arrays;

class Arrays Copy Example{
    public static void main (String [] args)
{
```

```
        String [] arr = {"Mukund", "Soman", "Ruvika"};
        String [] copy = Arrays.copyOf(arr, arr.length);
        S.o.p ("Copied Array:" + Arrays.toString(copy));
    }
}
```

↳ Copying with Range:-

```
import java.util.Arrays;

class ArraysCopy {
    public static void main(String [] args){
        String [] arr = {"Mukund", "Soman"};
        String [] copy = Arrays.copyOfRange(arr, 0, 2);
        S.o.p("Copied Array with Range:"
            + Arrays.toString(copy));
    }
}
```

6 Comparing Arrays:-

Ans Arrays.equals() compares two Arrays.

E.g

```
import java.util.Arrays;

class ArraysEqualsExample {
    public static void main(String [] args)
{
        String [] arr1 = {"Mukund", "Soman"};
        String [] arr2 = {"Mukund", "Soman"};
        S.o.p (Arrays.equals(arr1, arr2))
    }
}
```

↳ Output:-

Equal

↳ Array List class

1 Features of ArrayList

Ans Stores the data dynamically and allows duplicate elements.
- Maintains the order of insertion.
- Is non-synchronized (not thread-safe).
- Enables random access due to its index-based structure.
- Slower than linkedlist for certain operations
- Does not support primitive types (like int, float, etc) directly.
- Stores object in heap memory.
- Only stores objects, not primitives directly.
- Supports operations like adding, displaying, modifying, deleting and iterating over elements

↳ Key Methods and Examples:-

1 add element: Adds elements to the arraylist.

->
```
ArrayList <String> obj = new ArrayList<>();
obj.add("Somit");
S.o.p("Array List:" +obj);
```

Output:- ArrayList : [Somit]

2 add (index, element) - Inserts an element at a specific index.

→ obj. add (1, "Bhavin");

∴ Output:- ArrayList: [Somit, Bhavin]

3 addAll (Collection): Appends all elements of another collection

→ obj. addAll (obj 1);

∴ Output - Combines elements from obj 1 into obj.

4 addAll (index, Collection) - Inserts all elements of another collection at a specific index.

5 get (index) - Fetches an element at a specific index.

→ S.o.p (obj. get (0));

6 set (index, element) - Updates an element at the specified index.

→ obj. set (2, "Bhavin");

7 remove (index) / remove (element) - Deletes an element by index or value.

8 Contains (element) - check if a specific element exists.

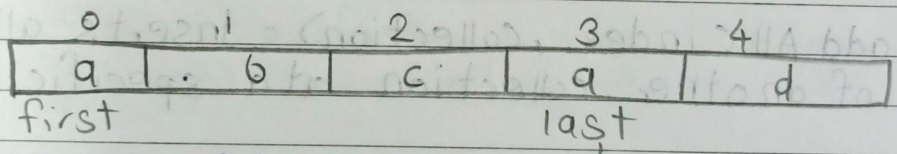
→ if(obj. contains("Milan") {
S.o.p("Exists");
}

9 Contains All (collection) - Verifies if all elements of a collection exist.

10 index of (element) - Returns the first occurrence of an element.

11 last Index of (element) - Returns the last occurrence index of an element.

12 ~~retain All (Collection) -~~



last Index of("a");

output - 3

12 retain All (collection): keeps only common elements between collections.

13 clear() - Removes all elements, leaving the list empty.

14 to Array() - Converts Array List to an array.

15 Removing Duplicates:-

- Using contains() - Iteratively checks for and skips duplicates.
- Using LinkedHashSet - Maintains order while removing duplicates.

16 Sorting and Reversing:-

- Collections.sort(obj) - Sorts the list
- Collections.reverse(obj) - Reverse the order.

↳ Vectors:-

1 Introduction to vectors:-

Ans Vector is a class in Java that implements a growable array of objects.

↳ Key Points:-

- Comes for Java's first version.
- Part of java.util Package.
- Can store different types of objects and allows dynamic resizing.
- Slower than ArrayList but synchronized (thread-safe).
- Initial capacity grows as needed (usually doubles).

2 Vector Declaration-

Ans
```
vector v = new vector();
vector v = new vector(10); // Initial capacity of 10
```

3 Basic Vector operations:-

Ans Adding elements (add and addElement):

a Method 1: add()

→
```
vector <String> v = new Vector <>();
v.add("Bhavin");
v.add("Soman");
S.o.p(v);
```

∴ Output - [Bhavin, Soman].

b Method 2: addElement()

→ v.addElement("Anil");

∴ Note- addElement is the same as add().

c Adding at a specific index:-

→ v.add(1, "Vicky");

∴ Output - [Bhavin, Vicky, Soman].

d Adding Another Collection (addAll):-

→
```
vector <String> v1 = new vector <>();
v1.add("John");
v1.add("Doe");
v.addAll(v1);
```

∴ Output- [Bhavin, vicky, Soman, John, Doe]

e Inserting at a specific Index (insertElement At):

→
```
v.insertElement At("Ramesh", 1);
```

∴ Output- [Bhavin, Ramesh, vicky, Soman].

4 Accessing Elements:-

Ans a Fetch by Index (get):-

→
```
S.o.p(v.get(2));
```

∴ Output- vicky.

b Fetch by elementAt():-

→
```
S.o.p(v.elementAt(2));
```

5 Modifying Elements:-

Ans Replace an element (set):-

→ a v.set(1, "Mukund");

∴ Output - [Bhavin, Mukund, Vicky, Soman]

6 Removing Elements

Ans By Value (removes):-

→ v.remove("Soman");

. By Index (removes):-

→ v.remove(1);

↳ output - [Bhavin, Vicky].

. Remove All:-

→ v.clear();

∴ Output - []

7 Capacity Management

Ans Checking Size:

→ S.o.p(v.size());

6 checking Capacity:

→ S.o.p(v.capacity());

∴ Defautt Capacity: 10

. Ensure Capacity

→ v.ensureCapacity(20);

8 Enumeration

Ans Iterate using Enumeration.

→ Enumeration e = v.elements();
while(e.hasMoreElements()){
S.o.p(e.nextElement());
}

9 Sorting and Reversing

Ans Sorting (Collections.sort)

→ Collections.sort(v);

b Reversing (Collections.reverse)

→ Collections.reverse(v);

10 Searching

Ans Find Index (indexof and lastIndexof)

→ S.o.p(v.indexof("Mukund"));
S.o.p(v.lastIndexof("Mukund"));

11 Contains (check if element exists):-

Ans
```
if (v.contains ("Bhavin")){
    S.o.p ("Exists");
}
```

12 Removing Duplicates

Ans Method-1:- Loop and contains check

→
```
for (String element : v){
    if (!v1.contains(element)){
        v1.add(element);
    }
}
```

b Method-2:- Using LinkedHashSet

→
```
LinkedHashSet<String> set = new LinkedHashSet
                                <>(v);
v.clear();
v.add()
v.addAll(set);
```

13 Retaining Common elements (retainAll)

Ans
```
v.retainAll(v1);
```

→ Output - Elements common in both vectors are retained.

↳ Linked List class

1 Introduction to linkedlist class

Ans
- Origin- Introduced in Java 1.2.
- Implements- List and Deque interfaces
- Package- java.util package.
- Part of - Java collection framework.
- Structure - Based on Doubly linked list.

→ Features:

- Supports null values.
- Can traverse both forward and backward.
- Non- Synchronized (must be explicitly Synchronized if needed),
- Faster than ArrayList for insertions/deletions but slower for element access.

2 Constructors

Ansa Default Constructors:

→ LinkedList <string> list= new LinkedList<>();

b Collection Constructor:

→ LinkedList <string> list= new LinkedList<>(existingCollection);

3 ↳ Key LinkedList Methods

a Adding elements

Ans add(element) - Adds at the end.

→ LinkedList<String> List = new LinkedList<>();

```
list.add("Bhavin");
list.add("Somit");
S.o.p(list);
```

∴ Output - [Bhavin, Somit]

ii add(index, element) - Inserts at a specific position.

→

```
list.add(1, "Vicky");
S.o.p(list);
```

∴ Output - [Bhavin, Vicky, Somit]

iii addFirst(element): Inserts at the begining

→

```
list.addFirst("Milan");
S.o.p(list);
```

∴ Output - [Milan, Bhavin, Vicky, Somit].

iv addLast(element) : Inserts at the end.

→

```
list.addLast("Rahul");
```

S.o.p (list);

Output - [Milan, Bhavin, Vicky, Somit, Rahul].

4 Accessing Elements

Ans get(index) - Retrieves an element by index.

→ S.o.p(list.get(2));

∴ Output - Vicky

ii getFirst() - Fetches the first element.

→ S.o.p(list.getFirst());

∴ Output - Milan

iii getLast() - Fetches the last element.

→ S.o.p(list.getLast());

∴ Output - Rahul

5 Removing elements

Ans remove() - Removes the first element.

→ list.remove();
S.o.p(list);

Output:- [Bhavin, Vicky, Sumit, Rahul]

ii remove(index) - Removes element by index.

→ list.remove(1);
S.o.p(list);

∴ Output- [Bhavin, Sumit, Rahul].

iii remove(element) - Removes the first occurrence of the specified element.

→ list.remove("Sumit");
S.o.p(list);

∴ Output - [Bhavin, Rahul].

iv removeFirst() - Removes the first element.

→ list.removeFirst();
S.o.p(list);

∴ Output - [Rahul]

v removeLast() - Removes the last element

→ list.removeLast();
S.o.p(list);

∴ Output - []

6 checking elements

Ans contains (element): check if the list contains a specific element.

→ S.o.p (list.contains ("Bhavin"));

∴ output - true or false

ii) contains All (Collection): checks if all elements of another collection exist.

→ LinkedList<String> list2 = new LinkedList<>();
list2.add ("Somit");
S.o.p (list.containsAll (list2));

∴ output - true or false

7 Capacity and Size

Ans size() - Returns the number of elements

→ S.o.p (list.size());

∴ Output - 3

ii) clear() - Removes all elements.

→ list.clear();
S.o.p (list);

∴ Out put - []

8 Conversion to Arrays

Ans to Array() - Converts to an array

→ object [] arr = list.toArray();
for(object o : arr){
S.o.p(o);
}

ii to Array(Array Name) - Converts to an array of specific type.

→ String [] arr = list.toArray(new String[0]);

9 Index Search

Ans index of(element) - Returns the index of the first occurrence.

→ S.o.p(list.indexof("Somit"));

∴ Output - 1

ii lastIndex of(element) - Returns the last occurrence index.

→ S.o.p(list.lastIndexof("Milan"));

10 Special Methods

Ans
- offer(element) - Adds to the end.
- offerFirst(element) - Adds to the front.
- offerLast(element) - Adds to the end.
- peek() - Retrieves but does not remove the first element.
- poll() - Retrieves and remove firstelement.
- push(element) - Pushes element at start.
- pop() - Removes and Returns the first element.

↳ Example - Complete Demonstration

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args){
        LinkedList<String> list = new LinkedList
                                  <>();

        // Adding elements
        list.add("Bhavin");
        list.add("Sumit");
        list.addFirst("Milan");
        list.addLast("Rahul");
        S.o.p("After adding:" +list);

        // Accessing elements
        S.o.p("first:" + list.getFirst());
        S.o.p("Last:" + list.getLast());
```

// Accessing Removing elements

```
list.remove(2);
list.remove("Somit");
S.o.p("After Removal" +list);

// checking
S.o.p("Contains Bhavin:" + list.contains
("Bhavin"));

// Size and Conversion
S.o.p("Size :" +list.size());
Object[] arr = list.toArray();
for(object el : arr){
        S.o.p("Array element:" +el);
    }
}
}
```

↳ Output :-

```
After Adding: [Milan, Bhavin, Somit, Rahul]
First : Milan
Last: Rahul
After Removal : [Bhavin, Rahul]
Contains Bhavin : true
Size : 2
Array Element: Bhavin
Array Element: Rahul
```

↳ Inheritance

1 Inheritance Concept in Java

Ans Inheritance allows a class to acquire the properties and behaviors of another class.

→ Purpose- Code reusability and method Overriding.

→ Key Benefits:-

- Reusability - Common functionalities are placed in the base class.
- Scalability - New features can be added by extending base classes.
- Hierarchy - Establishes relationships between classes.

2 Access Modifiers in Java

Ans Access Modifiers control the visibility of class Members. (variables/ Methods)

→ Types:-

1 Default - Accessible within the same package.
2 Public - Accessible from anywhere.
3 Protected - Accessible within the package and subclasses.

4 Private - Accessible only within the class.

3 Default access Specifier

Ans If no modifier is specified, it is treated as default.
. Accessible within the same package but not from outside.

→ Example :-

```
class DASpecifier {
    int id = 100;
    String name = "Sumit";

    void getData() {
        S.o.p (id);
        S.o.p (name);
    }
    public static void main (String [] args) {
        DASpecifier ob = new DASpecifier();
        ob.getData();
    }
}
```

4 Public Access Specifier

Ans Members declared public are accessible from any class or package.

→ Example

```
Class Employee {
    public int id;
    public string name;

    public void getData() {
        id=100;
        name= " Jostin";
    }
    public void display() {
        S.o.p (id);
        S.o.p (name);
    }
}
class PA Specifier {
    public static void main (String [] args) {
        Employee ob = new Employee();
        ob. getData();
        ob. display();
    }
}
```

5 Protected Access Specifier

Ans Accessible within the same package and subclasses in other packages.

→ Example:

```
class Employee {
    protected int id;
```

```
    protected String name;

    protected void getData(){
        id=100;
        name="Jostin";
    }

    protected void display(){
        S.o.p (id);
        S.o.p (name);
    }
}

class ProtectedSpecifier {
    public static void main (String []args){
        Employee ob= new Employee();
        ob.id=1001;
        ob.name="Mirza";
        ob.display();
    }
}
```

6 Private Access Specifier

Ans Members declared private are accessible only within the class.
. Most secure and prevents outside access.

-> Example :-

```
class Employee {
    private int id;
```

```
    private String name;

    public void getData(){
        id= 100;
        name= "Jostin";
    }
    public void display(){
        S.o.p (id+"   "+name);
    }
}

class DASpecifier{
    public static void main (String [] args)
{
    Employee ob= new Employee();
    ob. getData();
    ob. display();
    }
}
```

7 Inheritance in java

Ans Inheritance enables one class to inherit the properties of another.

→ Syntax :-

```
class Parent{
    int id;
    void show(){
        S.o.p ("Parent class");
    }
}
```

```
class child extends Parent {
    void display() {
        S.o.p ("child class");
    }
}
```

8 Types of Inheritance

Ans a Single level Inheritance

- one child class inherits from one parent class.

→ Example

```
class Employee {
    int id;
    String name;
    void display() {
        S.o.p (id + " " + name);
    }
}

class Department extends Employee {
    String deptID;
    void getDeptInfo() {
        deptID = "D001";
        S.o.p @C deptID);
    }
    public static void main (String [] args) {
        Department obj = new Department();
        obj.id = 1001;
```

```
    obj. name = "Sumit";
    obj. display();
    obj. getDeptInfo();
  }
}
```

6 Multilevel Inheritance

Ans A derived class inherits from another derived class.

⇒ Example:-

```
class Calculation {
    int x, y;

    void getSum() {
        x = 10;
        y = 20;
        S.o.p("Sum:" + (x+y));
    }
}
class Subtraction extends Calculation {
    void subtract() {
        S.o.p("Subtract:" + (x-y));
    }
}
class Product extends Subtraction {
    void multiply() {
        S.o.p("Product" + (x*y));
    }
}
```

```
class Multilevel{
    public static void main (String [] args){
        product ob= new Product ();
          ob. getSum ();
          ob. subtract ();
          ob. multiply ();
    }
}
```

c Hierarchical Inheritance

→ Multiple child classes inherit from one parent class.

∴ Example

```
class Animal{
      void eat(){
          S.o.p("Eating...");
      }
}
class Dog extends Animal{
      void bark(){
          S.o.p ("Barking");
      }
}
class cat extends Animal{
      void meow(){
          S.o.p("Meowing");
      }
}
```

```
public class Test Inheritance{
    public static void main (String [Jargs)
{
    Dog d = new Dog();
      d. eat();
      d. bark();

    Cat c = new Cat();
      c. eat();
      c. bark();
      c. meow();
  }
}
```

d ~~Hybrid Interface~~
d Hybrid Inheritance (Using Inheritance)

Ans
```
interface Father{
    void show();
}

interface Mother{
    void show();
}

class Parent{
    void display(){
        S.o.p("Parent class");
    }
}
class child extends Parent implements Father,
                                    Mother{
```

```
    public void show () {
        S.o.p ("child class");
    }
public static void (string [] args) {
    child obj = new child ();
    obj. display ();
    obj. show ();
    }
}
```

↳ Exception Handling

1 What is Exception Handling in java?

Ans An exception is an unexpected event that occurs during program execution, disrupting normal flow.

→ How it works :-

. When an exception occurs: -

1 An exception object is created.
2 The object holds the name, description and stack trace (line where error occurred).
3 Java's JVM handles the exception using a default handler.

2 Why do Exceptions occur?

Ans Common Causes:-

1. Code Errors - Mistyped logic.
2. Type Mismatch - Incorrect data type usage.
3. Wrong Input/Output - Invalid file paths, user inputs.
4. Device failure - Hardware issues.
5. Network Issues - Connection errors,
6. Out of Bound - Accessing arrays with invalid indexes.
7. Null References - Dereferencing null objects.
8. File errors - Accessing unavailable files.
9. Arithmetic Errors - Division by zero.
10. Database Errors - SQL query failures.

3 Errors vs Exceptions

Ans Errors - Un recoverable issues (e.g StackOverflowError). Cannot be handled by program.
- Exceptions - Recoverable problems (e.g IO exception, NullPointerException).

4 Exception Hierarchy

Ans Base class - Throwable

- Exceptions - Recoverable (e.g. IOException).
- Errors: Unrecoverable (e.g; OutOfMemoryError).

5 Types of Exceptions

Ansa checked Exceptions:-

- checked at Compile time.

- Examples- IOException, SQLException

→ Example- (File Not Found)

```
import java.io.*;
class checkedException1 {
    public static void main(String[] args)
                        throws IOException {
        FileReader file = new FileReader("file.txt");
    }
}
```

∴ Compiler checks for file existence at compile time.

b Unchecked Exceptions

- occor during runtime.
- Examples: ArithmeticException, NullPointerException

→ Example (Divide By Zero).

```
class UncheckedException1 {
    public static void main(String[] args){
        int a = 10, b = 0;
        int c = a/b;
    }
}
```

∴ Program will throw ArithmeticException at Runtime.

6 Handling Exceptions

Ans Key words for Exception Handling:-

1 try - Block of code where an exception may occor.
2 Catch - Handles exceptions that occor in the try block.
3 finally - Code that executes regardless of whether an exception occors.
4 throw - Used to explicitly throw an exception.
5 throws - Declares exceptions that a method might throw.

7 Using try - catch

Ans
```
class ExceptionExample {
    public static void main (String [] args){
        try{
            int a= 100, b= 0;
            int c= a/b;
        } catch (ArithmeticException e){
            S.o.p (" Exception occurred: "+ e);
        }
    }
}
```

8 Multiple Catch Blocks:-

Ans
```
public class MultiCatch {
    public static void main (String [] args){
```

```
try {
    int arr[] = new int [5];
    arr[10] = 50;
}
catch (ArithmeticException e){
    S.o.pc "Arithmetic Error";
}
catch (Array Index out of Bounds Exception e){
    S.o.pc" Array Index Error";
}
Catch (Exception e){
    S.o.pc" General Exception";
}
}
}
```

9 finally Block

Ans

```
class FinallyExample {
    public static void main (String [] args) {
        try {
            int x = 10 / 0;
        }
        Catch (ArithmeticException e) {
            S.o.p ("Caught Exception");
        }
        finally {
            S.o.pc" Finally block";
        }
    }
}
```

10 throw and throws

Ans throw: used to manually throw an exception.

. throws - Declares that a method can throw a Exceptions.

→ Example (throw):-

```
class Test {
  public static void validate (int age) {
    if (age<18) {
      throw new ArithmeticException ("Not
          eligible to vote");
    } else {
      S.o.p ("Eligible to vote");
    }
  }
  public static void main (String [] args) {
    validate (13);
    validate (16);
  }
}
```

→ Example (throws):-

```
import java.io.*;
class Test {
  public Static void main (String [] args)
          throws IOException {
    FileReader file= new FileReader ("data.txt");
  }
}
```

11 Common Exceptions and Examples

Ans a Arithmetic Exception:

→ int a = 10/0;

b Null Pointer Exception-

→ String str = null;
str.length();

c Array Index Out of Bounds Exception:

→ int[] arr = new int[5];
S.o.p(arr[10]);

d Class Cast Exception

→ object obj = new String("Java");
Integer i = (Integer) obj;

12 Custom Exceptions

Ans class CustomException extends Exception {
CustomException(String msg){
super(msg);
}
}
class Test {
public static void main(String[] args){
try {
throw new CustomException("This is a
custom exception");

```
} catch ( Custom Exception e) {
        S.o.p ( e.getMessage() );
    }
}
}
```

↳ Java Threading

1. What is a Thread?

Ans A thread is a small unit of execution in a program. It allows the CPU to execute multiple tasks simultaneously.

-> Key Points :-

- Thread class - Comes from java.lang.Thread.
- Independent Execution - Each thread has its path of execution from start to finish.
- Multi-threading - Multiple threads can run concurrently.

2 Process vs Thread

Ans Process - Independent unit with its own memory.

- Thread - Lightweight, shares memory with other threads within the same process.

↳ Example

- Running Word, Excel and a music player are

separate processes.

- Downloading a file while streaming music in the same app involves multiple threads.

3 why use threads?

Ans Efficiency- Threads use less memory than processes.

- Performance- Allows tasks to run in parallel, improving CPU usage.
- Communication- Threads can share data easily.

4 Multi - threading

Ans Running multiple threads simultaneously within a program.

→ Example :-

∴ Banking System-

- Customer A deposits - Thread 1.
- Customer B withdraws - Thread 2.
- Customer C transfers - Thread 3

→ All these threads run concurrently.

5 Thread lifecycle

Ans New State- Thread is created but not started.
- Runnable State- Thread is ready to run.
- Running state- CPU starts executing the thread.
- waiting / Blocked State- Thread waits for resources or I/O.
- Dead State- Threads finishes or terminated.

6 Creating Threads

Ans By Extending Thread class

→
```
class Thread Eg extends Thread {
    public void run() {
        S.o.p("Thread is running");
    }
    public static void main (String[] args) {
        Thread Eg t = new Thread Eg();
        t.start();
    }
}
```

→ By Implementing Runnable Interface

```
class Thread Eg implements Runnable {
    public void run() {
        S.o.p ("Runnable is running");
    }
    public static void main (String[] args) {
```

```
        Thread &g t= new Thread &g ();
        Thread th= new Thread (t);
         th.start ();
    }
}
```

7 Thread Methods

Ans
a run() - Code to execute inside the thread.
b start() - Starts the thread.
c sleep(ms) - Pauses thread for specified milliseconds.
d yield() - Pauses current thread to allow others to run.
e join() - waits for another thread to finish.
f isAlive() - Checks if a thread is running.
g suspend() - Temporarily pauses a thread.
h resume() - Resumes a suspended thread.

8 Using sleep() -

Ans
```
class A extends Thread {
    public void run() {
        try {
            for (int i=1; i<=3; i++) {
                S.o.p ("Hello!");
                Thread.sleep (2000);
            }
        } catch (InterruptedException e) {
            S.o.p (e);
        }
    }
```

```
public static void main (String [] args){
    Main t = ne
        At = new A();
        t.start();
    }
}
```

→ Output :-

Hello!
(wait for 2 seconds)
Hello!
(wait for 2 seconds)
Hello!

9 Using yield()

Ans
```
class thread1 extends Thread{
    public void run(){
        for(int i=1; i<=3; i++){
            S.o.p("Thread 1 is running");
            Thread.yield();
        }
    }
}
class thread2 extends Thread{
    public void run(){
        for(int i=1; i<=3; i++){
            S.o.p("Thread 2 is running");
        }
    }
}
```

```
public class Jest {
    public static void main (String [] args){
        thread1 t1= new thread1 ();
        thread2 t2= new thread 2 ();
         t1. start ();
         t2. start ();
    }
}
```

→ Output :-

```
Thread 1 is running
Thread 1 is running
Thread 2 is running
Thread 2 is running
Thread 2 is running
Thread 1 is running
```

10 join() Method

Ans

```
class A extends Thread {
    public void run () {
      for (int i= 1; i<=3 ; i++) {
          S.o.p (" Thread is running");
      }
  }
}
public class Jest {
    public static void main (String [] args)
                throws InterruptedException {
        A t1 = new A ();
         t1. start ();
```

```
        t1.join();
        S.o.p("Main thread resumes");
    }
}
```

→ Output :-

```
A thread is running
A thread is running
A thread is running
Main thread resumes
```

→ join() makes the main thread wait until t1 completes execution.

11 Synchronization (Thread Safety).

Ans Problem - Multiple threads accessing shared resources can cause data inconsistency.
. Solution - Synchronization ensures only one thread accesses the resource at a time.

→ Example :-

```
class Counter {
    int count = 0;
    public synchronized void increment() {
        count++;
    }
}
class ThreadEg extends Thread {
    Counter c;
```

```
Thread Eg ( Counter c ) {
    this.c = c;
}
public void run() {
    for(int i=0; i<1000; i++) {
        c.increment();
    }
}
}
public class Main {
    public static void main( String [] args)
            throws InterruptedException
{
    Counter c = new Counter();
    ThreadEg t1 = new ThreadEg();
    ThreadEg t2 = new ThreadEg(c);
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    S.o.p("Count: " + c.count);
}
}
```

12 wait(), notify(), and notifyAll()

Ans.
- wait()- Pauses the thread until notified
- notify()- wakes up one waiting thread.
- notifyAll()- wakes up all waiting threads

→ Example:-

```
class MovieBooking extends Thread {
    int total = 0;
    public void run() {
        Synchronized (this) {
            for (int i = 1; i <= 5; i++) {
                total += 100;
            }
            this.notify();
        }
    }
}
public class Test {
    public static void main (String [] args)
                    InterroptedException {
        MovieBooking mb = new MovieBooking();
        mb.start();
        Synchronized (mb) {
            mb.wait();
            S.o.p ("Total earnings:" + mb.total);
        }
    }
}
```

↳ Java file handling

1 File handling in java

Ans File handling in Java allows programs to create, write, read and manipulate files.

→ Key classes:-

- File - To create and manage file properties
- FileWriter - To write content to files.
- Scanner - To read data from files.
- IOException - Handles input/output exceptions

2 Creating a file in java

Ans Simple file Creation:-

```
import java.io.File;
import java.io.IOException;

public class CreateFile {
    public static void main (String [] args)
                throws IOException{

        File F = new File("D:\\testFile.txt");
        F.createNewFile();
    }
}
```

→ Eg-2 : check if file already exists

```
import java.io.File;
import java.io.IOException;

public class CreateFile{
    public static void main (String [] args){
            throws IOException {
        File F = new File ("D:\\testFile.txt");
```

```
if (F.createNewFile()){
    S.o.p("File created");
}
else{
    S.o.p("File already exists");
}
}
```

→ Eg-3 Handling exceptions

```
import java.io.File;
import java.io.IOException;

public class CreateFile{
    public static void main(String[] args){
        try{
            File F = new File("D:\\testFile.txt");
            F.createNewFile();
        }
        Catch(IOException e){
            S.o.p("Error:" + e);
        }
    }
}
```

3 writing to a file

Ans Write Content to file

→

```
import java.io.FileWriter;
import java.io.IOException;
```

```
public class CreateFile{
    public static void main (String [] args) {
            throws IOException {

        FileWriter f= new Filewriter ("D:\\testfile
                                        .txt");
        f.write ("Hello, this is Java ");
        f.close ();
    }
}
```

→ Example 2:- write the Exception Handling

```
import java.io.FileWriter;
import java.io.IOException;

public class CreateFile {
    public static void main (String [] args)
{
    try {
        FileWriter f= new Filewriter ("D:\\testFile.
                                        txt");
        f.write ("Java file handling");
        f.close ();
    }
    catch (Exception e) {
        S.o.p ("Error occurred");
    }
}
}
```

→ Writing User Input to file

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class Createfile{
    public static void main(String[]args)
            throws IOException{

        FileWrite f = new Filewriter("D:\\testfile
                                    .txt");
        Scanner sc = new Scanner(System.in);

        S.o.p("Enter StudentID");
        String sid = sc.next();

        S.o.p("Enter student name");
        Stringsname = sc.next();

        f.write(sid+"  "+sname);
        f.close();
    }
}
```

→ Writing Multiple Entries in loop

```
public class CreateFile{
    public static void main(String[] args)
                throws IOException{
        FileWriter f = new FileWriter("D:\\test
                                File.txt");
```

```
Scanner sc = new Scanner(System.in);

for(int i=0 ; i<3 ; i++){
    S.o.p("Enter ID, name, and Course");
    String SID = sc.next();
    String name= sc.next();
    String course= sc.next();
    f.write( sid + " " + name + " " + course +
                                    "\n");
  }
 f.close();
 }
}
```

4 Reading from a file

Ans

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class CreateFile{
    public static void main(String[]args)
        throws IOException{

    File f = new File("D:\\testFile.txt");
    Scanner reader = new Scanner(f);

    while(reader.hasNextLine()){
        String data= reader.nextLine();
        S.o.p(data);
    }
  }
}
```

5 File handling with user Input

Ans
```
import java.io.FileWriter;
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class StudentRecord {
    Static string sid, name, courses, branch,
                                    fee;
    static Scanner sc = new scanner (System.in);

    public static void getStudentDetails () {
        S.o.p ("Enter Details");
        Sid = sc.next();
        name = sc.next();
        course = sc.next();
        branch = sc.next();
        fee = sc.next();
    }
    public static void main (String [] args)
            throws IOException {

        FileWriter f = new FileWriter ("D:\\test
                                    File.txt");
        StudentRecord obj[] = new Student
                                    Record[3];

        for (int i = 0; i < 3 ; i++) {
            obj[i] = new StudentRecord();
            obj[i].getStudentDetails();
            f.write( obj[i].sid + " " + obj[i].name
```

```
+ "    " + obj[i].branch + "  " + obj[i].coorse
   + "    " + obj[i].fee + "\n");
}
 f.close();

// Reading from a file

File file= new File("D:\\testFile.txt");
 Scanner reader = new Scanner(file);

while(reader.hasNextLine()){
    String data = reader.nextLine();
    S.o.p(data);
  }
}
}
```

↳ JDBC in java

1 Establishing a Database Connection

Ans The first step is to establish a connection to the database using the connection objed

→ Example:-

```
import java.sql.*;

public class MyDbconnect {
    public static void main (String [] args) {
    try {
        Connection conn = DriverManager.get
            Connection ("jdbc:mysql://localhost:
                3306/db", "root", "root");
        S.o.p ("Connected with database");
    } catch (SQLException e) {
        S.o.p ("Error while connecting");
    }
}
```

→ DriverManager.getConnection establishes the connection.

2 Retrieving Data

Ans To retrieve data, use an SQL select query along with the ResultSet object.

→ Example:-

```
import java.sql.*;

public class Mydbconnect {
    public static void main (String [] args){
     try {
       Connection conn = DriverManager.get
           Connection ("jdbc:mysql://local
           host:3306/db","root","root");
       S.o.p ("Connected database success fo
                                  lly");

    Prepared Statement ps= conn.prepared Sta-
          tement ("select* from Student");
    Resolt Set rs = ps.execute Query();

    while (rs.next()){
        String rollno = rs.getString("Roll No");
        String name= rs.getString("username");
        String dept= rs.getString("Dept");
        S.o.p (rollno +"  "+name+"  "
                        +dept);
    }
  }catch (SQL Exception e){
       S.o.p ("Error while connecting");
    }
  }
}
```

3 Inserting Data

Ans To insert data, use an SQL insert query with Prepared Statement.

-> Example:-

```
import java.sql.*;

public class MyDbConnect {
    public static void main(String[] args){
        try {
            Connection conn = DriverManager.
                getConnection("jdbc:mysql://
                localhost:3306/demodb", "root",
                    "root");
            S.o.p("Connected Successfully");

            PreparedStatement ps= conn.prepared
            Statement("Insert into Student values
                (?,?)");

            ps.setString(1,"Physics");
            ps.setString(2,"1");
            ps.executeUpdate();

            S.o.p("Data Inserted");
        }
        catch(SQLException e){
            S.o.p("Error");
        }
    }
}
```

4 Updating Data

Ans To update data, use an SQL update query

→ Example:

```
import java.sql.*;

public class MyDbconnect {
    public static void main(String[]
            args) {
        try {
            Connection conn= DriverManager.
            getConnection("jdbc:mysql://localhost
            :3306/db","root","root");
            S.o.p("connected database success");

            PreparedStatement ps= conn.prepared
            Statement("update student set
                Dept =? where RollNo =?");
            ps.setString(1,"physics");
            ps.setString(2,"1");
            ps.executeUpdate();

            S.o.p("Data updated successfully");
        }
        catch(SQLException e){
            S.o.p("Error");
        }
    }
}
```

5 Deleting Data

Ans To delete data, use an SQL Delete query.

-> Example

```
import java.sql.*;

public class MyDbconnect {
    public static void main(String[] args){
        try {
            Connection conn = DriverManager.
                getConnection("jdbc:mysql://
                localhost:3306/db","root","root");
            S.o.p("Connected database success-
                                fully");

            PreparedStatement ps = conn.prepare
                Statement("Delete from student
                    where username=?");
            ps.setString(1,"Mehtab");
            ps.executeUpdate();

            S.o.p("Data deleted successfully");
        } catch(SQLException e){
            S.o.p("Error while connecting");
        }
    }
}
```

6 Using JSwing with JDBC

Ans A button in a JSwing GUI triggers JDBC operations, such as inserting data.

→ Example

```
btnNew Button.addActionListener(new
                ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        String firstName = firstname.getText();
        String userName = username.getText();
        try{
            Connection conn = DriverManager.get
                Connection("jdbc:mysql://localhost
                :3306/swing_demo","root","root");
            String query = "Insert into account
                values ('"+firstName+"', '"+
                userName+"')";
            Statement sta = conn.createStatement();
            sta.executeUpdate(query);
            JOptionPane.showMessageDialog(btnNew
                Button," Account created successfully
                !");
            conn.close();
        } catch(Exception ex){
            ex.printStackTrace();
        }
    }
});
```

↳ Lambda Expressions in Java

1 What are lambda Expressions?

Ans A lamba expression is a short way of writing a method or fonction without giving it a name. It allows you to create small, anonymous fonctions that you can use where methods are expected.

- It was introduced in Java 8 and is mainly used to simplify code, especially when working with fonctional interfaces (interfaces with a single abstract method).

2 Why Do we need Lambda Expressions?

Ans Before Java 8, if you wanted to pass behavior (logic) as a parameter, you had to use anonymous classes, which resulted in long, bulky code.

→ Example (without lambda).

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        S.o.p("Hello");
    }
};
r.run();
```

-> with Lambda E.g :-

Runnable r = () -> S.o.p ("Hello, world");
r.run();

3 Basic Syntax of Lambda Expressions

Ans A lambda expression consists of :-

- Parameter list (enclosed in parentheses ()).
- Arrow operator (->)
- Body (the logic or code to execute, enclosed in {} if more than one statement).

-> Syntax :-

(parameters) -> { body of the method }

-> Examples :-

- No para meter :-

() -> S.o.p ("Hello!");

- One parameter :-

x -> x * 2;

- Multiple parameters :-

(a, b) -> a+b;

4 Type Inference in lambda Expressions

Ans Type Inference means that Java can figure out the datatype of the parameters automatically, so you don't need to specify them explicity.

→ Example:

```
(int a, int b) -> a + b; // Explicit types
(a, b) -> a+b; // Type inferred by Java
```

5 Functional Interface Requirement:

Ans A lambda expression can only be used with a fonctional interface, which has exactly one abstract method. Common ~~exp~~ examples:

- Runnable (method: run())
- Callable (method: call())
- Comparator (method: compare())

∴ Java also provides the @FunctionalInterface annotation to ensure the interface has only one abstract method.

→ Example:

```
@FunctionalInterface
interface Greeting {
    void sayHello(String name);
}
```

// Lambda expression for this functional Interface:-

```
Greeting greeting = (name) -> S.o.p("Hello, "
                              + name);
greeting.sayHello("Alice");
```

6 Benefits of Lambda Expressions:

Ans. Less Code- Simplifies the code, making it shorter and easier to read.

- Improved Readability - Reduces the use of anonymous classes.
- Functional Programming Style- Supports functional Programming concepts in Java.
- Better Performance- often improves runtime performance due to reduced object creation.

7 Example of Lambda in Collections:-

Ans. Lambda expressions are widely used in java collections, especially with methods like forEach, filter etc.

↳ Example (using forEach):-

```
List<String> names = Arrays.asList("Alice", "Bob"
    , "charlie");
names.forEach(name -> S.o.p(name));
```