ALL PROGRAMS

According to sheetal pareira, array of objects, class with multiple classes (object ka ek and ek for the actual functionality of the thing), jagged array, do logic wale, etc aayenge)

- Java Program to Add Two Integers (entered by user)
- Java Program to Swap Two Numbers
- Java Program to Check Whether a Number is Even or Odd
- Java Program to Check Whether an Alphabet is Vowel or Consonant
- Java Program to Find Factorial of a Number
- Java Program to Check Whether a Number is Prime or Not
- Java Program to Find Factorial of a Number **Using Recursion**

<preferred>

```
//java program to add 2 integers
import java.util.*;
class addition{
    public int Add(int a, int b)
    {
        return a+b;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter two numbers: ");
        int n1 = sc.nextInt();
        int n2 = sc.nextInt();
addition obj = new addition(); // create object as add is not a static method
        int r = obj.Add(n1, n2);
        System.out.println("Sum of two numbers is "+ r );
    }
}
```

OR <less preferred:>

```java
//java program to add 2 integers
import java.util.*;
class addition{
    public static int Add(int a, int b)
    {
        return a+b;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter two numbers: ");
        int n1 = sc.nextInt();
        int n2 = sc.nextInt();
        int r = Add(n1, n2);
        System.out.println("Sum of two numbers is "+ r );
    }
}
```

A number whose sum of factors (excluding the number itself) is equal to the number is called a perfect number. In other words, if the sum of positive divisors (excluding the number itself) of a number equals the number itself is called a perfect number. WAJP to check if the number entered is perfect.

```java
// A number whose sum of factors (excluding the number itself)
// is equal to the number is called a perfect number.
// In other words, if the sum of positive divisors (excluding the number itself)
// of a number equals the number itself is called a perfect number.
import java.util.*;
class test{
    static Vector<Integer> arr = new Vector<>();

    public static Vector<Integer> divisors(int n)
    {
        for(int i=1;i<n;i++)
        {
            if(n%i == 0)
            {
```

```
                arr.add(i);
            }
        }
        return arr;
    }

    public static int addition(Vector<Integer> v)
    {
        int sum=0;
        for(int i: v)
        {
            sum+=i;
        }
        return sum;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number.");
        int n = sc.nextInt();
        divisors(n);
        if(addition(arr) == n)
        {
            System.out.println("Perfect");
        }
        else{
            System.out.println("Not perfect");
        }
    }
}
```

#1. WAJP to find area of circle and rectangle using a) method overloading, b) method overriding

Overloading: Use same name ka func and diff parameters

```
import java.util.Scanner;

class shape{
```

```
    public double area(int length, int breadth)
    {
        return length*breadth;
    }

    public double area(int side)
    {
        return 3.14 * side * side;
    }
}

class area{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        shape s = new shape();
        while (true){
            System.out.println("Enter 1 for rectangle and 2 for circle. Enter -1
to exit.");
            int choice = sc.nextInt();
            switch(choice)
            {
                case 1:
                System.out.println("Enter length and breadth");
                int l = sc.nextInt();
                int b = sc.nextInt();
                System.out.println("Area is "+s.area(l,b));
                break;
                case 2:
                System.out.println("Enter side");
                int side = sc.nextInt();
                System.out.println("Area is "+s.area(side));
                break;
                case -1:
                System.out.println("Exiting...");
                return;
                default:
                System.out.println("Invalid choice");
            }
        }
```

```
    }
}
```

Overriding: usually uses extend keyword

Points to note:

- this.variable = variable
- Func definition in superclass
- Declaration of subclasses
- Use of @Override
- How the functions inside subclasses dont have parameters
- How the objects are declared inside switch

```
import java.util.Scanner;

class shape{
    double area()
    {
        return 0; //generic, only to tell method definition
    }
}

class Circle extends shape{
    public
    int side;

    Circle(int side)
    {
        this.side = side;
    }

    @Override
    double area()
    {
        return 3.14 * side * side;
    }
}

class Rectangle extends shape{
    public
```

```
    int length;
    int breadth;

    Rectangle(int length, int breadth)
    {
        this.length = length;
        this.breadth = breadth;
    }

    @Override
    double area()
    {
        return length*breadth;
    }
}

class area{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (true){
            System.out.println("Enter 1 for rectangle and 2 for circle. Enter -1
to exit.");
            int choice = sc.nextInt();
            switch(choice)
            {
                case 1:
                System.out.println("Enter length and breadth");
                int l = sc.nextInt();
                int b = sc.nextInt();
                Rectangle s1 = new Rectangle(l, b);
                System.out.println("Area is "+s1.area());
                break;
                case 2:
                System.out.println("Enter side");
                int side = sc.nextInt();
                Circle c1 = new Circle(side);
                System.out.println("Area is "+c1.area());
                break;
                case -1:
```

```
                System.out.println("Exiting...");
                return;
                default:
                System.out.println("Invalid choice");
            }
        }
    }
}
```

#2. WAJP to demonstrate Simple Parameterized Constructor For i) Finding Prime Number or ii) finding area of rectangle

```
import java.util.Scanner;

class Rectangle {
    int length, breadth;

    // Parameterized constructor
    Rectangle(int length, int breadth) {
        this.length = length;
        this.breadth = breadth;
    }

    int area() {
        return length * breadth;
    }
}

class AreaDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter length and breadth:");
        int l = sc.nextInt();
        int b = sc.nextInt();

        Rectangle rect = new Rectangle(l, b);
        System.out.println("Area is " + rect.area());
    }
```

```
}
```

Prime

```
import java.util.*;

class Prime {
    public
    int num;

    Prime(int num) {
        this.num = num;
    }

    boolean isPrime() {
        if (num <= 1) return false;
        for (int i = 2; i <= num / 2; i++) {
            if (num % i == 0) return false;
        }
        return true;
    }
}

class PrimeDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a number:");
        int n = sc.nextInt();

        Prime p = new Prime(n);
        if (p.isPrime())
            System.out.println(n + " is a prime number.");
        else
            System.out.println(n + " is not a prime number.");
    }
}
```

#3. WAJP to illustrate constructor overloading to print the names of students by creating a Student class. If no name is passed while creating an object of Student class, then the name should be "hello world", otherwise the name should be equal to the String value passed while creating object of Student class.

```
class Student {
    String name;

    // Constructor with no parameter
    Student() {
        name = "hello world";
    }

    // Constructor with a parameter
    Student(String name) {
        this.name = name;
    }

    void display() {
        System.out.println("Student name: " + name);
    }
}

public class StudentDemo {
    public static void main(String[] args) {
        // Object with no name
        Student s1 = new Student();
        s1.display();

        // Object with a name
        Student s2 = new Student("Ashwera");
        s2.display();
    }
}
```

```
Student name: hello world
```

Student name: Ashwera

Q. WAJP to have an Employee class that has employee Id (empId) and employee name (name) as fields and ‘setData’ & ‘showData’ as methods that assign data to employee objects and display the contents of employee objects respectively.

```java
// Q. WAJP to have an Employee class that has employee Id (empId) and
// employee name (name) as fields and ‘setData’ & ‘showData’ as
// methods that assign data to employee objects and display the
// contents of employee objects respectively.

import java.util.Scanner;

class Employee{
    public
    long empId;
    String name;

    Scanner sc = new Scanner(System.in);

    Employee() {
    }

    Employee(long empId, String name){
        this.empId = empId;
        this.name = name;
    }

    void setData()
    {
        System.out.println("Enter ID and name.");
        long id = sc.nextLong();
        sc.nextLine();
        String n = sc.next();
        empId=id;
        name=n;
    }

    void showData(){
```

```
        System.out.println("ID of the employee: "+empId);
        System.out.println("Name of the employee: "+name);
    }
}

class Management{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter number of employees");
        int num = sc.nextInt();
        while(num-->0)
        {
            Employee e1 = new Employee();
            e1.setData();
            e1.showData();
        }
    }
}
```

WAJP to create a Rectangle class to implement the basic properties of a Rectangle. Declare an array of five Rectangle objects, to store data namely length and width (datatype double). Take values from user and print areas of all 5 rectangles.

```
import java.util.Scanner;

// WAJP to create a Rectangle class to implement the basic properties of a
Rectangle.
// Declare an array of five Rectangle objects, to store data namely length and
width
//  (datatype double). Take values from user and print areas of all 5 rectangles.

class Rectangle{
    public
    double length;
    double breadth;
```

```
    public Rectangle(double length, double breadth) {
        this.length = length;
        this.breadth = breadth;
    }

    double area()
    {
        return length*breadth;
    }
}

class ArrayOfRectangles{
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        Rectangle[] array = new Rectangle[5];
        for(int i=0;i<5;i++)
        {
            System.out.println("Enter length and breadth");
            double l = sc.nextDouble();
            double b = sc.nextDouble();
            array[i] = new Rectangle(l, b);
            System.out.println("Area of this rectangle: "+ array[i].area());
        }
    }
}
```

2. WAJP to create the Groceries class, declare an array of Groceries objects to pass the product
name, price and expiry date (dd/mm/yyyy format) as attributes for 10 products. Display the data
as output. To set the data to the Groceries object use the following two approaches:
a. Have a separate member method in the Groceries class
b. Use a constructor

```
import java.util.Scanner;

// Groceries class
```

```
class Groceries {
    String name;
    double price;
    String expiryDate;

    // a) Member method to set data
    void setData(String n, double p, String exp) {
        name = n;
        price = p;
        expiryDate = exp;
    }

    // b) Constructor to set data
    Groceries(String n, double p, String exp) {
        name = n;
        price = p;
        expiryDate = exp;
    }

    void display() {
        System.out.println("Name: " + name + ", Price: " + price + ", Expiry: " +
expiryDate);
    }
}

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

        // Using member method
        Groceries[] products1 = new Groceries[10];
        System.out.println("Enter 10 products using setData():");
        for (int i = 0; i < 10; i++) {
            products1[i] = new Groceries("", 0, ""); // create object
            System.out.println("Product " + (i+1) + " Name:");
            String n = sc.nextLine();
            System.out.println("Price:");
            double p = sc.nextDouble();
            sc.nextLine(); // consume newline
```

```
            System.out.println("Expiry date (dd/mm/yyyy):");
            String exp = sc.nextLine();
            products1[i].setData(n, p, exp);
        }

        System.out.println("\nDisplaying products (setData method):");
        for (Groceries g : products1) {
            g.display();
        }

        // Using constructor
        Groceries[] products2 = new Groceries[10];
        System.out.println("\nEnter 10 products using constructor:");
        for (int i = 0; i < 10; i++) {
            System.out.println("Product " + (i+1) + " Name:");
            String n = sc.nextLine();
            System.out.println("Price:");
            double p = sc.nextDouble();
            sc.nextLine(); // consume newline
            System.out.println("Expiry date (dd/mm/yyyy):");
            String exp = sc.nextLine();
            products2[i] = new Groceries(n, p, exp); // set data using
constructor
        }

        System.out.println("\nDisplaying products (constructor):");
        for (Groceries g : products2) {
            g.display();
        }
    }
}
```

The only difference is ke class method use karte time you create object with random default parameters before, and tehn populate after u get the data with the function. By using a parameterized constructor, you just pass once.

3. Create a Jagged array for a student db where every student has taken different number of classes. Eventually show the average marks of each student.

```
import java.util.Scanner;

class Student {
    long id;
    int[] marks;
    double avg;

    Student(long id, int n) {
        this.id = id;
        marks = new int[n];
    }

    void calculateAverage() {
        int sum = 0;
        for (int m : marks) sum += m;
        avg = (double) sum / marks.length;
    }
}

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

        System.out.println("Enter number of students:");
        int n = sc.nextInt();

        Student[] students = new Student[n]; //stores student data
        int[][] StudentRecord = new int[n][]; // jagged array where n is the
number of rows

        for (int i = 0; i < n; i++) {
            System.out.println("Enter student ID and number of subjects:");
            long id = sc.nextLong();
            int nos = sc.nextInt();
```

```
            Student s = new Student(id, nos);
            StudentRecord[i] = new int[nos]; // initialize jagged array row

            System.out.println("Enter marks for " + nos + " subjects:");
            for (int j = 0; j < nos; j++) {
                int mark = sc.nextInt();
                s.marks[j] = mark;
                StudentRecord[i][j] = mark; // store in jagged array
            }
            s.calculateAverage();
            students[i] = s;
        }

        System.out.println("\nJagged array of marks:");
        for (int i = 0; i < StudentRecord.length; i++) {
            for (int m : StudentRecord[i]) {
                System.out.print(m + " ");
            }
            System.out.println();
        }

        System.out.println("\nStudent averages:");
        for (Student s : students) {
            System.out.println("ID " + s.id + " | Average: " + s.avg);
        }
    }
}
```

WAJP using a for-each loop to print the lowest marks in an array storing marks of 10 students

```
import java.util.Scanner;

public class LowestMarks {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int[] marks = new int[10];
```

```
        System.out.println("Enter marks of 10 students:");
        for (int i = 0; i < 10; i++) {
            marks[i] = sc.nextInt();
        }

        int lowest = marks[0]; // assume first is lowest
        for (int mark : marks) { // for-each loop
            if (mark < lowest) {
                lowest = mark;
            }
        }

        System.out.println("Lowest mark is: " + lowest);
    }
}
```

Q. WAJP to traverse ArrayList elements (store names of 5 fruits) using the Iterator interface. Also demonstrate the use of add(), get() and set() methods.

```
import java.util.*;

public class FruitDemo {
    public static void main(String[] args) {
        // Create ArrayList and add 5 fruits
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
        fruits.add("Orange");
        fruits.add("Grapes");

        // Demonstrate get() and set()
        System.out.println("Fruit at index 2: " + fruits.get(2)); // Mango
        fruits.set(1, "Papaya"); // Replace Banana with Papaya
        System.out.println("After set(), ArrayList: " + fruits);
```

```
        // Traverse using Iterator
        System.out.println("Traversing using Iterator:");
        Iterator<String> it = fruits.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

4. Demonstrate the use of 'super' keyword using a Java program. Here, Employee class (store 'salary' here) should inherit Person class, so all the properties (id, name) of Person should be inherited to Employee by default. To initialize all the properties, use parent class constructor from child class. Display id, name and salary of 3 employees as an output.

```
import java.util.Scanner;

// Parent class
class Person {
    int id;
    String name;

    Person(int id, String name) {
        this.id = id;
        this.name = name;
    }
}

// Child class
class Employee extends Person {
    double salary;

    Employee(int id, String name, double salary) {
        super(id, name); // call parent constructor
        this.salary = salary;
    }
```

```
    void display() {
        System.out.println("ID: " + id + ", Name: " + name + ", Salary: " +
salary);
    }
}

public class EmployeeDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Employee[] employees = new Employee[3];

        for (int i = 0; i < 3; i++) {
            System.out.println("Enter ID, Name and Salary for employee " + (i+1)
+ ":");
            int id = sc.nextInt();
            sc.nextLine(); // consume newline
            String name = sc.nextLine();
            double salary = sc.nextDouble();
            employees[i] = new Employee(id, name, salary);
        }

        System.out.println("\nEmployee Details:");
        for (Employee e : employees) {
            e.display();
        }
    }
}
```

Write a Java program to find the longest Palindromic Substring within a string

```
import java.util.Scanner;

public class LongestPalindrome {
    // Function to expand around center
```

```
    static String expand(String s, int left, int right) {
        while (left >= 0 && right < s.length() && s.charAt(left) ==
s.charAt(right)) {
            left--;
            right++;
        }
        return s.substring(left + 1, right);
    }

    static String longestPalindromicSubstring(String s) {
        String longest = "";
        for (int i = 0; i < s.length(); i++) {
            String odd = expand(s, i, i);
            if (odd.length() > longest.length()) longest = odd;
            String even = expand(s, i, i + 1);
            if (even.length() > longest.length()) longest = even;
        }
        return longest;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter a string:");
        String str = sc.nextLine();

        String result = longestPalindromicSubstring(str);
        System.out.println("Longest Palindromic Substring: " + result);
    }
}
```

Array of objects

```
// Q. Write a JAVA program to create a class Book with the following details:
// bookId
// title
// price
// Accept information about n books from the user using an array of objects.
// Your program should provide functionalities to:
```

```java
// Add book details
// Display all book details
// Display the details of the book having the highest price.

import java.util.Scanner;

class Book{
    public
    long bookID;
    String title;
    double price;

    Book(long bookID, String title, double price)
    {
        this.bookID = bookID;
        this.title = title;
        this.price = price;
    }
}

class arrayofobjects{

    public static void display(Book[] books)
    {
        for(Book b: books)
        {
            System.out.println("Book ID: "+b.bookID+" Title: "+b.title+" Price:
"+b.price);
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter number of books");
        int n = sc.nextInt();
        Book[] books = new Book[n];

        int highestindex=-1;
        double highestcost=0;
        for(int i=0;i<n;i++){
```

```
            System.out.println("Enter book ID, title, and price");
            long id = sc.nextLong();
            sc.nextLine(); // consume newline
            String title = sc.nextLine();
            double price = sc.nextDouble();
            books[i] = new Book(id, title, price);
            if(price>highestcost)
            {
                highestcost = price;
                highestindex = i;
            }
        }

        display(books);
        System.out.println("Book with highest price:");
        System.out.println("Book ID: "+books[highestindex].bookID+" Title:
"+books[highestindex].title+" Price: "+books[highestindex].price);
        sc.close();
    }
}
```

```
// Write a JAVA program to create a class Movie with fields:
// mid (Movie ID)
// mname (Movie name)
// rating (double)
// Store details of n movies using a Vector of Movie objects.
// Your program should:
// Add movie details
// Display all movie details
// Display the movie with the highest rating

import java.util.Scanner;
import java.util.Vector;

class Movie{
    public
    long mid;
    String mname;
    double rating;
```

```
    Movie(long mid, String mname, double rating)
    {
        this.mid = mid;
        this.mname = mname;
        this.rating = rating;
    }
}

class arrayofobjects{

    public static void display(Vector<Movie> movies)
    {
        for(Movie m: movies)
        {
            System.out.println("Movie ID: "+m.mid+" | Movie Name: "+m.mname+" |
Rating: "+m.rating);
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter number of movies");
        int n = sc.nextInt();
        Vector <Movie> movies = new Vector<>();

        int highestindex=-1;
        double highestrating=0;
        for(int i=0;i<n;i++){
            System.out.println("Enter movieID, movie name, and rating");
            long id = sc.nextLong();
            sc.nextLine();
            String title = sc.nextLine();
            double rating = sc.nextDouble();
            sc.nextLine();
            Movie m = new Movie(id, title, rating);
            movies.add(m);
            if(rating>highestrating)
            {
                highestrating = rating;
```

```
                highestindex = i;
            }
        }

        display(movies);
        System.out.println("Movie with highest rating:");
        System.out.println("Movie ID: "+movies.get(highestindex).mid+" | Movie
Name: "+movies.get(highestindex).mname+" | Rating:
"+movies.get(highestindex).rating);
        sc.close();
    }
}
```

Write a program to display volumes of sphere and hemisphere . Make use of abstract class.

Q1. WAJP to demonstrate the occurrence of ArithmeticException caused by dividing a number by zero and handle the exception using a try and catch block.

Q2. WAJP to demonstrate a scenario where ArrayIndexOutOfBoundsException occurs.

Q3. WAJP to demonstrate the occurrence of ArithmeticException caused by non terminating Big Decimal and handle the exception.

WAJP to create a Thread, and use atleast 4 methods that represent its life cycle; like, 'yield', 'sleep', 'join', start', 'interrupt', 'run', etc.

WAJP to demonstrate the use of 'synchronized' keyword in multithreading.

https://docs.google.com/document/d/1tM6La9VpnZNpBbQ9tcV11IMaogQTkrpxjKrWp1_O1EU/edit?tab=t.0

Question;

Write a Java class to represent a car with private fields for the make, model, and year. Implement getter and setter methods for these fields and demonstrate their usage in the main method.

```java
class Car {
    private String make;
    private String model;
    private int year;

    // getters
    public String getMake() { return make; }
    public String getModel() { return model; }
    public int getYear() { return year; }

    // setters
    public void setMake(String make) { this.make = make; }
    public void setModel(String model) { this.model = model; }
    public void setYear(int year) { this.year = year; }
}

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

        Car c = new Car();

        c.setMake("Toyota");
        c.setModel("Corolla");
        c.setYear(2020);

        System.out.println(c.getMake());
        System.out.println(c.getModel());
        System.out.println(c.getYear());
    }
}
```

Overriding and Polymorphism

Write a Java program to create a class known as "BankAccount" with methods called deposit() and withdraw(). Create a subclass called SavingsAccount that overrides the withdraw() method to prevent withdrawals if the account balance falls below one hundred.

```
// Write a Java program to create a class known as "BankAccount" with methods
//  called deposit() and withdraw(). Create a subclass called SavingsAccount
//  that overrides the withdraw() method to prevent withdrawals if the account
//   balance falls below one hundred.

class BankAccount{
    protected int balance;

    BankAccount(int balance) {
        this.balance = balance;
    }

    void deposit(int amt){
        balance+=amt;
    }
    void withdraw(int amt){
        if(amt > balance){
            //print not poss
            System.out.println("Withdrawal not possible. Insufficient balance.");
        }
        else{
            balance -= amt;
                System.out.println("Withdrawal successful. Amount withdrawn: " +
amt);
            //print withdrawn
        }
    }
}

class SavingsAccount extends BankAccount{

    public SavingsAccount(int bal) {
```

```
        super(bal);
    }

    void withdraw(int amt){
        if(balance-amt <= 100){
            // print not poss
                System.out.println("Withdrawal not possible. Balance cannot fall
below 100.");
        }
        else{
            balance-=amt;
            //print withdrawn
                System.out.println("Withdrawal successful. Amount withdrawn: " +
amt);
        }
    }
}

class Main{
    public static void main(String[] args) {
        SavingsAccount ac = new SavingsAccount(0);
        System.out.println(ac.balance);
        ac.deposit(1000);
        System.out.println(ac.balance);
        ac.withdraw(500);
        System.out.println(ac.balance);
        ac.withdraw(550);
    }
}
```

Abstraction/abstract class

Write a Java program to create an abstract class Shape3D with abstract methods calculateVolume() and calculateSurfaceArea(). Create subclasses Sphere and Cube that extend the Shape3D class and implement the respective methods to calculate the volume and surface area of each shape.

```
// Write a Java program to create an abstract class Shape3D
// with abstract methods calculateVolume() and calculateSurfaceArea().
// Create subclasses Sphere and Cube that extend the Shape3D class and implement
//   the respective methods to calculate the volume and surface area of each
shape.

abstract class Shape3D{
    abstract double calculateVolume(int x);
    abstract double calculateSurfaceArea(int x);
    abstract void display();
}

class Sphere extends Shape3D{
    double area,vol;
    double calculateSurfaceArea(int side){
        area = 4 * 3.14 * side * side;
        return area;
    }
    double calculateVolume(int side){
        vol = (4/(double)3) * 3.14 * side * side * side;
        return vol;
    }
    void display(){
        System.out.println("Area: "+area);
        System.out.println("VOL:" + vol);
    }
}

class Cube extends Shape3D{
    double area,vol;
    double calculateSurfaceArea(int side){
        area = 6 * side * side;
        return area;
```

```
    }
    double calculateVolume(int side){
        vol = side * side * side;
        return vol;
    }
    void display(){
        System.out.println("Area: "+area);
        System.out.println("VOL:" + vol);
    }
}

class Main{
    public static void main(String[] args) {
        Shape3D s1 = new Sphere();
        Shape3D s2 = new Cube();
        s1.calculateSurfaceArea(5);
        s1.calculateVolume(5);
        s2.calculateSurfaceArea(4);
        s2.calculateVolume(4);
        s1.display();
        s2.display();
    }
}
```

Interface

Write a Java program to create an interface Drawable with a method draw() that takes no arguments and returns void. Create three classes Circle, Rectangle, and Triangle that implement the Drawable interface and override the draw() method to draw their respective shapes.

One file:

```java
// Write a Java program to create an interface Drawable
//  with a method draw() that takes no arguments and returns void.
//  Create three classes Circle, Rectangle, and Triangle that implement the
//   Drawable interface and override the draw() method to draw their respective
shapes.

public interface Drawable{
    public void draw();
}

class Circle implements Drawable{
    public void draw(){
        System.out.println("O");
    }
}

class Rectangle implements Drawable{
    public void draw(){
        System.out.println("[]");
    }
}

class Triangle implements Drawable{
    public void draw(){
        System.out.println("A");
    }
}
```

Other file:

```java

class Main{
    public static void main(String[] args) {
        Circle c1 = new Circle();
```

```
        Rectangle r1 = new Rectangle();
        Triangle t1 = new Triangle();
        c1.draw();
        r1.draw();
        t1.draw();
    }
}
```

Basic trycatch

Write a Java program that throws an arithmetic exception and catch it using a try-catch block.

```
// Write a Java program that throws an arithmetic exception a
// nd catch it using a try-catch block.

class Main{
    public static void main(String []args){
        int a=4;
        int c;
        int b=0;
        try{
            c=a/b;
            System.out.println("Result: "+c);
        }
        catch(Exception e){
            System.out.println("Division by zero not allowed");
        }
    }
}
```

Exception

Write a Java program to create a method that takes an integer as a parameter and throws an exception if the number is odd.

```
class Main {

    static void check(int n) throws Exception {
        if (n % 2 != 0) {
            throw new Exception("Number is odd");
        }
    }

    public static void main(String[] args) {

        int x = 4041;

        try {
            check(x);
            System.out.println("Number is even");
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
        finally {
            System.out.println("yahoo");
        }
    }
}
```

Custom exception

Create a custom exception class called **NegativeNumberException**. Write a program that takes an integer input from the user and throws this exception if the number is negative.

```
// Custom exception
class NegativeNumberException extends Exception {
    NegativeNumberException(String msg) {
        super(msg);
    }
}

class Main {

    static void check(int x) throws NegativeNumberException {
        if (x < 0) {
            throw new NegativeNumberException("Number is negative");
        }
    }

    public static void main(String[] args) {

        int y = 4;
        int z = -4;

        try {
            check(z);
            System.out.println("Positive");
        }
        catch (NegativeNumberException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

module 1

Questions:
https://docs.google.com/document/d/1b01irEApY9YiaZvZW99k5NS8Cb4CcedZtOSC3jM8p0g/edit?tab=t.0
Summary: skimmable module, nothing to takeaway really apart from definitions etc. Very basic if someone already knows basics of Java etc. High volume of content but very basic to understand.

OOPM Introduction

Unstructured Programming

- Has just one program that has everything, called main prog
- Example: assembly language

Example code:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int a=2;
    int b=2;
    cout << a+b << endl;
}
```

Structured Programming

- Also called procedural programming
- For separate tasks, ‘procedures’ are created (that we now know as functions). These procedures are called in main.

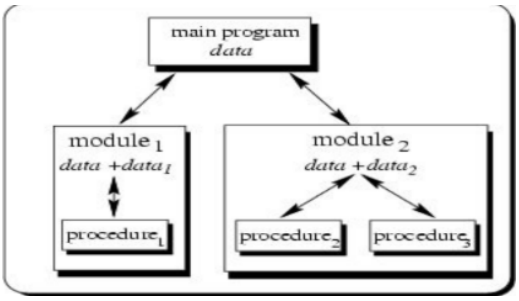
Example code:

```
#include <bits/stdc++.h>
using namespace std;

int add(int a, int b)
{
    return a+b;
}

int main()
```

```
{
    int a,b;
    cout << "enter two numbers";
    cin >> a >> b;
    cout << add(a,b) << endl;
}
```

Modular Programming

- Procedures (functions) with common functionality are grouped together into “modules”
- So a program breaks down into several modules, and each module gets broken down into procedures and has its own data.



Object oriented programming (OOP)

- Works on objects: the smallest unit of OOP languages
- Focus is more on data, rather than procedures

```
#include <bits/stdc++.h>
using namespace std;

class Addition{
    int a,b,c;
    public:
    void read()
    {
        cin >> a;
        cin >> b;
    }
    void add()
    {
        c=a+b;
    }
    void display()
    {
        cout << "The sum is:" << c;
```

```
    }
};

int main(){
    Addition obj;
    cout << "Enter the numbers";
    obj.read();
    obj.add();
    obj.display();
}
```

Limitations of Procedural Programming

1. Does not model the real world very well
2. Does not give importance to data
3. Zero privacy, zero true reusability
4. Functions and data should be treated equally.

Difference between Procedural and OOP

Feature	Procedure Oriented	Object Oriented
Division	Divided into functions	Divided into objects
Importance	Importance is given to functions and the sequence of actions, not to data	Importance is given to the data rather than procedure, because it works as a real world model
Approach	Top down	Bottom up
Access specifier	No access specifier	Access specifier like public, protected, and private
Data Moving	Data can move from function to function	Objects can communicate thru member functions
Expansion	Adding new function and data is not easy	Adding new function and data is easy
Data Access	All data can be accessed	Data can be kept private

	freely	or public so access is controlled
Data Hiding	Less secure	More secure
Overloading	Not possible to overload	Overloading is possible as function overloading and operator overloading

Object Oriented Features

Intro

- C++ is statically typed, compiled, case sensitive, general purpose, free-form programming language that supports POP and OOP.
- It is middle level language, it comprises both HLL and LLL

C++ Class definition

- Template/format/blueprint
- A class definition starts with the keyword class -> class name -> class body(within {})->semicolon
- It contains data members and member functions

Example:

```
class Box{
    public:
    double length; // Length of a box
    double breadth; // Breadth of a box
    double height; // Height of a box
};
```

C++ Objects

- A class provides blueprint for objects
- Object is created from class
- Object is a variable of class
- Object is declared like a variable:
  Box b1,b2;
- Both boxes have different copies of data members(height, length, breadth)

Accessing data members

- Any **public** data member can be accessed using the **direct member class operator (.)**

Example:

```
class Box{
    public:
    double length; // Length of a box
    double breadth; // Breadth of a box
    double height; // Height of a box
};

int main()
{
    Box b1,b2;
    double vol = 0;
    b1.length = 5;
    b1.breadth =6;
    b1.height = 4;
    b2.length = 7;
    b2.breadth =8;
    b2.height = 9;
    vol = b1.height * b1.breadth * b1.length ; //notice how we accessed members
    cout << "Volume of box 1: " << vol << endl;
    vol = b2.height * b2.breadth * b2.length ; //notice how we accessed members
    cout << "Volume of box 2: " << vol << endl;
}
```

Members of class

- Definition: Function that has its definition within the class
- Members are accessed using the dot operator
- Can be defined within class or separately using a **scope resolution vector ::**

Example:

Way 1

```
class Box{
    public:
```

```
    double length; // Length of a box
    double breadth; // Breadth of a box
    double height; // Height of a box
    double getVolume(){
        return length * breadth * height;
    }
};
```

Way 2

```
class Box{
    public:
    double length; // Length of a box
    double breadth; // Breadth of a box
    double height; // Height of a box
    double getVolume();
};

double Box::getVolume(){
return length * breadth * height;
}
```

Accessing this:

```
int main()
{
    Box b1,b2;
    double vol = 0;
    b1.length = 5;
    b1.breadth =6;
    b1.height = 4;
    cout << "Volume of box 1: " << b1.getVolume() << endl;
}
```

Data encapsulation

- **Definition:** The wrapping up of data and functions into a single unit called class is called encapsulation.

- OOP encapsulates data and functions into packages called objects
- Notice how every object IRL has some attribute (data) and a function. A pen has the attributes: blue, metal, gel and function: writing, drawing. Similarly, each object in OOP has functions and attributes. The box example we earlier took had attributes of length, height, and width, and we associated a function of getting its volume. Getting the area could be another function and so on.
- **Encapsulation enforces abstraction.**
- It separates external view from the internal view and protects the integrity of object's data. For example, to know the volume of a box, you don't need to know what its parameters are. You can directly call for volume and get that, without being concerned with implementation details.

Benefits of Encapsulation

1. Abstraction between object and clients
2. Protects objects from unwanted access
3. Can change the class implementation later on
4. Can constraint object ka state: lets you choose to only have positive balance wale Bank Accounts or months of a calendar from 1 to 12 *only*.

Data Abstraction

- Abstraction refers to representing essential features without including the background details
- Classes use the concept of abstraction and are defined as a list of abstract attributes, functions operate on these attributes.
- Consider a car, you hit the reverse gear and the car starts moving in the backward direction. You do not know (and usually don't care) how the mechanics behind this work. This is data abstraction.

Information Hiding

- Data hiding is one of the important features of Object Oriented Programming
- It prevents the functions of a program to access the internal representation of a class
- The access restriction is labelled by public, private, and protected (access specifiers)
- *The default access for members and classes is private*
- A class can have multiple labelled sections as per the access specifiers

- The access specifiers apply to both data members (variables) and functions.
- **ACCESS SPECIFIERS**
  public: Accessible anywhere outside the class, within the same prog
  private: Accessible only in the class and within friend functions. Outside the class, these cannot be viewed OR accessed.
  protected: Similar to private, except protected members and attributes can be accessed in child classes (explained later) which are called derived classes.

Inheritance

More in mod 4 doc

- **Definition:** Inheritance is the process by which objects of one class acquired the properties of objects of another class.
- Promotes the idea of reusability
- Can add new features to an existing class without changing it.
- Consider how an upgrade is made to Instagram. The reels repost feature is added onto the existing app, without changing the earlier functionality. You didn't have to build the entire app again to add one feature, you could just reuse the older code and add your feature's code only.
- **superclass:** Parent class being extended (instagram main app code)
- **Subclass:** this is the child class that inherits behaviour from parent class/superclass. (this is the reels repost code). Each subclass gets a copy of every field and method from the superclass.
- Each object of the subclass is also an object of the superclass.

Example in code:
Syntax: public class subclass extends superclass{}

Usage: public class Feature237 extends Instagram{...}

By extending Instagram, every Feature237 object:
1. Receive a copy of each method from Instagram
2. Feature237 can also replace behaviour from Instagram. This process is called **overriding.**

//more in mod 4

Polymorphism

- Polymorphism, a Greek term, means the ability to take more than one form.

- **Definition:** Ability for the same code to be used with different types of objects and behave differently is called polymorphism.
- The process of making an operator exhibit different behaviors in different instances is known as operator overloading.
  Consider the area operator, its task is to give you the area of a shape. Now depending on the shape, the formula changes but you don't want to have ten names for the same functionality. Polymorphism allows you to have different functions with the same name that act differently with different instances.
- Using a single function name to perform different type of task is known as **function overloading**.

```
class Shape {
    public void draw() {
    }
}

class Square extends Shape {
    public void draw() {
        System.out.println("Drawing Square");
    }
}

class Circle extends Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

public class Shapes {
    public static void main(String[] args) {
        Shape a = new Square();
        Shape b = new Circle();
        a.draw();
        b.draw();
    }
}
```

DIFFERENCE BETWEEN RUNTIME AND COMPILE TIME POLYMORPHISM!!!!

Polymorphic Parameters

You can use the same function for different objects using polymorphism.

Runtime Polymorphism in Java

- Also called **dynamic method dispatch.**
- Resolves a call to an overridden method at runtime
- Use of reference variable of a superclass to call for an overridden method.

Benefits of Runtime Polymorphism in Java

1. Enabling a class to offer its specialisation to another method.
2. Implementation transfer of one method to another can be done without copy pasting
3. The call to an overridden method can be resolved dynamically during runtime

Method Overriding

1. Runtime polymorphism ko implement karne ka way is by method overriding.
2. For this to happen, objects should have the same name, arguments, and type as their parent class but can have different functionality.

Upcasting

Definition: When an overridden method of child class is called through parent reference.
The object type represents the method or functionality invoked.
Upcasting happens in runtime.
Also called dynamic method dispatch. Why? The method functionality is decided dynamically.
Also called late binding. Why? The process of binding the method w object happens after compilation.

Rules for runtime polymorphism

1. Both child and parent class should have same method names
2. Same parameter
3. Not possible to override **private method** of parent class
4. Not possible to override static methods/final methods

Example:

```java
class Shape {
    public void draw() {
        System.out.println("Drawing...");
    }
}

class Square extends Shape {
    public void draw() {
        System.out.println("Drawing Square");
    }
}

class Circle extends Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
      public void erase(){
      System.out. println(“Erased”);
      }
}

public class test {
    public static void main(String[] args) {
        Shape a = new Square();
        Shape b = new Circle();
        a.draw();
        b.draw();
        b.erase(); //gives error as we called Circle with parent ref Shape and
                   //Shape does not have an erase func
    }
}
```

Reusability concept

1. Code reusability is an important feature of polymorphism, and it’s implemented using inheritance, polymorphism, and abstraction
2. Helps developers save time and effort by avoiding writing the same code again and again.
3. Makes programs more efficient and easier to maintain.

Coupling and Cohesion

Cohesion

Definition: Parameter of how effectively elements work together to achieve a single purpose **within** a module or object.
A high level of cohesion = elements are tightly integrated

Coupling

Definition: Degree of interdependence between software modules.
Tight coupling = Modules are v closely connected = changes in one module can affect others.
Loose coupling = Modules are independent = changes in one module has a very minimal effect on another module.

Best practice = loose coupling and high cohesion
Why? Helps create flexible classes or modules that are still less prone to break when changes are made.

Low cohesion

Module or object within the system has multiple functionalities that are not related closely. A single module/object performs a range of independent functions.

1. Makes the code harder to understand
2. Leads to confusion, makes it challenging to update the code

High cohesion

Each module within a system has a single well defined function. Each component is highly focused on a specific task and has all important info about that function.
To make it more cohesive, use the idea of Single Responsibility Principle. This means a class should have only one reason to change.

Tight Coupling

A situation when classes or modules have many dependencies on each other.

- Makes it hard to modify/extend our modules without affecting each other
- Leads to increased risk of bugs / side effects

Loose Coupling

Situation where classes and modules have very minimal dependencies on each other. Change in one class/module is very unlikely to affect the other.

- Makes code modular and reduces risk of introducing bugs
- Makes the code easier to maintain and extend

Types of Cohesion

1. **Functional cohesion:** Performing a single, well defined task without any unintended consequences.
2. **Informational cohesion:** Represents a set of data / group of operations for this data.

Points 1 and 2 are highly desirable cohesion.

3. **Procedural cohesion:** Series of tasks that must be performed in a certain order.
4. **Temporal cohesion:** Set of tasks that must be performed around the same time.
5. **Logical cohesion:** Among a related set of tasks, user can choose which task to perform
6. **Coincidental cohesion:** Set of tasks without a logical connection. They're grouped together by chance.

Types of Coupling

1. **Content coupling:** Module modifies/relies on internal details of another module
2. **Control coupling:** one module controls flow of control in another module
3. **Data coupling:** One module passes data to another, without knowing its internal structure
4. **Stamp coupling:** One module uses only a part of other module's state
5. **Common coupling:** Multiple modules share a common data structure
6. **External coupling:** Module depends on external resources.
7. **Message coupling:** Modules communicate with each other thru messages

How to achieve high cohesion?

1. Each class should have a single responsibility. All methods and data should be related to that responsibility.
2. Encapsulate data and methods that belong in a class.
3. Aim to design the system as a collection of small and reusable modules.
4. Use proper design patterns.

How to check cohesion?

1. Identify the responsibilities. What is the main purpose, what does it do?
2. Analyse methods in the class. Do they contribute to the same main responsibility?
3. Check if the methods use the same instance variables / share common functionality
4. Determine if there's duplicated code
5. Class should have a single, clear responsibility.

How to achieve loose coupling?

1. Define interfaces: ensures that the objects communicate with each other sahi se.
2. Dependency injection: create objects with necessary dependencies, rather than having objects with their own dependencies.
3. Use design patterns, provide a unified interface for interacting with multiple objects.
4. Use encapsulation to limit the visibility of data and methods, avoid exposing implementation details.

How to check coupling

1. Look at classes used by a particular class. If this class is dependent on a lot of other classes, this indicates high degree of coupling.
2. Count number of method calls. Large number of calls = high coupling
3. Level of abstraction. Low level = high degree of coupling

Comparison between C, C++, and Java

//not for exam

COMPARISION	C	C++	JAVA
INHERITANCE	Not Supported in C.	Single, Mutli level & Multiple inheritance.	Single, Multilevel & Hierarchical inheritance. Multiple not Permitted.
POLYMORPHISM	Not Supported in C.	Method Overloading & Operator Overloading are allowed.	Method overloading is allowed but not operator overloading.
MULTI THREADING	No built in Support for Multithreading.	No built in Support for Multithreading.	Java provides built in support for Multithreading.
Header files	Supported. Header file has .h extension.<stdio.h>	Supported . <iostream>	Not Supported.
Compilation	Compiler	Compiler	Compiler & Interpreter.

Metrics	C	C++	Java
Programming Paradigm	Procedural language	Object-Oriented Programming (OOP)	Pure Object Oriented
Origin	Based on assembly language	Based on C language	Based on C and C++
Developer	Dennis Ritchie in 1972	Bjarne Stroustrup in 1979	James Gosling in 1991
Translator	Compiler only	Compiler only	Interpreted language (Compiler + interpreter)
Platform Dependency	Platform Dependent	Platform Dependent	Platform Independent
Code execution	Direct	Direct	Executed by JVM (Java Virtual Machine)
Approach	Top-down approach	Bottom-up approach	Bottom-up approach
File generation	.exe files	.exe files	.class files
Pre-processor directives	Support header files (#include, #define)	Supported (#header, #define)	Use Packages (import)
keywords	Support 32 keywords	Supports 63 keywords	50 defined keywords
Datatypes (union, structure)	Supported	Supported	Not supported
Inheritance	No inheritance	Supported	Supported except Multiple inheritance
Overloading	No overloading	Support Function overloading (Polymorphism)	Operator overloading is not supported
Pointers	Supported	Supported	Not supported
Allocation	Use malloc, calloc	Use new, delete	Garbage collector
Exception Handling	Not supported	Supported	Supported
Templates	Not supported	Supported	Not supported
Destructors	No constructor neither destructor	Supported	Not supported

Exception Handling

More in mod5 tab

Exception

Definition: An event that occurs during the execution of a program that disrupts the flow of the program.
Java exceptions are special events that indicate when something bad happens.

Why handle an exception?

1. To maintain the normal desired flow of the program
2. If unhandled, the program crashes/requests fail.
3. Leads to loss in business and customers if this happens often.
4. To make the interface robust

Tracking Exceptions in Java

1. Record all events in the application log file
2. For small scale apps, going thru a log file is easy.
3. Enterprise apps have a LOT of requests. This makes manual analysis cumbersome.

How to handle?

1. **try** to execute statements contained within a block of code.
2. If an exceptional condition is detected, **throw** an exception
3. **catch** and process the exception object
4. You execute a block of code, designated by **finally,** which needs to be executed whether or not an exception occurs.

The try catch is the simplest method of handling exceptions.

```
try{
      }
catch(exception ex){
      //lines of code
}
```

Java: Intro

Features

1. Automatic type checking
2. Automatic garbage collection
3. Simplifies pointers
4. Simplified network access
5. Multithreading

Working

Java source (.java) -> Java compiler -> Java bytecode (.class)

class loader -> java interpreter -> java just in time compiler -> runtime system -> operating system -> hardware

Java is independent ONLY because it depends on the java virtual machine. Code is compiled to **byte code,** which is interpreted by the resident JVM. JIT compilers attempt to increase speed.

Java security:

- Pointer denial reduces chances of programs corrupting the host
- Applets are not allowed to run local executable, read or write to the system, and communicate with another server.

Java supports polymorphism, inheritance, encapsulation, and hence is an OOPL. Java programs contain only definitions and instantiations of classes: everything is encapsulated.

Advantages

1. Portable: write once run anywhere
2. Security
3. Robust memory management
4. Designed for network programming
5. Multi threaded (multiple simultaneous tasks can run)
6. Dynamic and extensible: has a lot of libraries, classes are stored in separate files, loaded only when needed.

Must know hereafter:

1. Primitive data types: what and examples, how to declare
2. Java sets primitive variables to zero or false
3. All objects are initially initialized to null
4. An array is always an object
5. If no value is assigned before usage, what is the output of a data type? Error.
6. Float has how many decimal places, what about double?
7. Java assignment are right associative
8. Operators in java, their precedence
9. How do you terminate a statement? Using a semicolon
10. How do you enclose a block? {}
11. Can blocks contain other blocks? Yes.
12. Flow of control of statements in java-> in the order they’re written
13. If, Nested if-else, If-else if-else statement and how they work
14. Relational operators and their meaning (==, !=, >=, <=, <, >)

15. Switch statements
16. For loops and their working
17. While loops and working
18. Do while loops and working : minimum/maximum number of times a loop is executed.
19. Break, continue statements
20. Each class definition in java is coded in a separate .java file
21. Initialisation and usage of objects
22. JDK components:
    javac: compiler
    java: interpreter
    jdb: java debugger
    appletviewer: tool to run the applets
    javap: print the java bytecodes
    javaprof: java profiler
    javadoc: documentation generator
    javah: creates C header files

Objects

Object: An entity that encapsulates data and behaviour
Data: variables inside the object
Behaviour: methods inside the object

Constructing an object: Type obj = new Type(parameters);
Calling an object: obj.method(parameters);

Class

Class: A program entity represents either a program or a template for a new type of objects. Consider:

Template:

```
class Banker{

    String name;

    long acc;

    double balance;



    Banker(String n, long a, double b)
```

```
    {
        name = n;
        acc = a;
        balance = b;
    }
}
```

A program or module

```
class arrayofobjects{

//functions

}
```

What is object oriented programming?

Programs that perform their behaviour as interaction between objects.
Abstraction: Separation between concepts and details.

Principles of OOP

1. Encapsulation: Objects hide their functions and data.
2. Inheritance: Each subclass inherits all variables of its superclass.
3. Polymorphism: Interface is same despite different data types

Methods

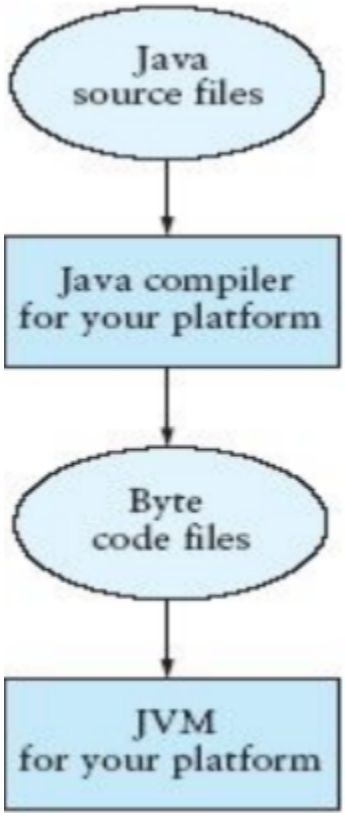
1. It is a named sequence of code that can be called by other code
2. A method takes some parameters. Performs some computation. Optionally returns a value/object.
3. Can be used as part of an expression statement
4. Method signature: specifies the name of the method, the type and name of each parameter, the type of value returned by the method, the checked exceptions thrown by the method, method modifiers (public, private, protected)
   example: public boolean setUserInfo (int i, int j, String name) throws IndexOutOfBoundsException {}

Public and private

- Methods and data might be declared public and private
- Good practice is to keep most data and method private
- Well-defined interface between classes helps eliminate errors

What are the salient features of Java?

1. Platform independent
2. Object oriented programming
3. It is a core language and a rich collection of commonly available packages
4. Can be embedded in webpages



compilation and execution of java program

Grouping Classes: The Java API

Api stands for application programming interface
Java = core + collection of packages
A package consists of
1. Swing: GUI package
2. AWT: application window toolkit

3. Util: utility ds

The **import** statement tells the compiler to make classes and methods of another package available
Main method indicates where the prog begins executing

//should know how to use import, main, processing, and running code ka format

Syntax definition: the set of legal structures or commands that can be used in a particular language. Example, ending a statement with ;, or enclosing blocks within {}

Syntax error is when the syntax rules are not followed in writing a program. Example: inconsistent brackets, missing semicolon, spelling mistakes, illegal data types etc.

Data Types

Primitive Data Types

Primitive-type data is stored in primitive-type variables
Example: byte, short, int, long etc
//should know inka range

Reference Data Type

Reference variables store the address of an object

Operators

- Pre and post increment operators
- Mathematical
- Binary (for concatenation of strings)
- Signed and unsigned bit shifts >> , <<
- Comparison
- Equality comparison
- Bitwise operators & | and ~
- Logical operators
- Conditional: ternary operator
- Assignment and compound assignment
  x += 1;

Type compatibility and conversion

- In operations on mixed-type operands, the numeric type of the smaller range is converted to the numeric type of the larger range
- byte-> short
- Short ->long etc.
- Larger sized variables (compatible) can hold smaller sized variable ka value without explicit typecasting
  Example:
  Int a=2;
  Double s = a*a;

Random information

- The new operator creates an instance of the class.
- The dot operator is used to call this instance.
- Static keyword indicates a static or class method
- All method arguments are call by value
- **An escape sequence** is a sequence of two characters beginning with \. Each holds different meaning

Sequence	Meaning
\n	Start a new output line
\t	Tab character
\\	Backslash character
\"	Double quote
\'	Single quote or apostrophe
\u*dddd*	The Unicode character whose code is *dddd* where each digit *d* is a hexadecimal digit in the range 0 to F (0–9, A–F)

- An **InputStream** is a sequence of characters that represent program input data. Similarly **OutputStream** is the output ka representation.
- Console keyboard stream is System.in, and for console window it is System.out
- The JVM approach enables a Java program written on one machine to execute on any other machine that has a JVM
- The Java String and StringBuffer classes are used to reference objects that store character strings

End of theory

module 2

Questions
https://docs.google.com/document/d/1PMlszpmPAoslcN_l86qbNLcxn-ZLzRj3bTE9GhcrL2U/edit?tab=t.0

Class, Object, Method, and Constructor

Class

Blueprint from which objects are created. To allocate memory to an object, the new keyword is used.

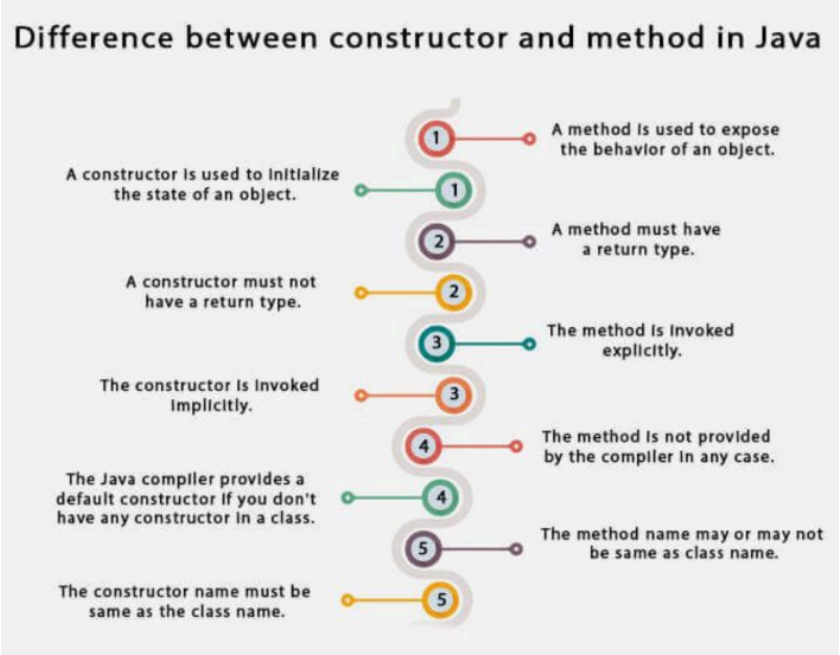
Object

Definition:
An entity that encapsulates data and methods. It is an instance of the class.
Data: variables
Methods: functions

Construction:

Student s1 = new Student();

Calling:
s1.marks; //call variable
s1.display(); //call method



Characteristics:

1. State: represents the data/value of the object
2. Behavior: Represents the functions of the obj
3. Identity: An object identity is implemented via a unique ID. The value of the ID is not visible to the external user but used internally by the JVM to identify each object uniquely.

Accessing data members

- The public data members of objects of a class can be accessed using the direct member access operator (.)
  Example:
  box1.length; where box1 is the object, length is the class variable.
- A member function of a class is a function that has its definition or its prototype within the class. Members are accessed using dot operator(.)
  Example
  box1.volume();
  These examples have alr been discussed in mod1

A Java program is a collection of classes. Each class definition (usually) in its own .java file. The file name must match the class name. All the instances of the same class have these same characteristics.

instanceOf operator

This operator is used to tell if an object belongs to a certain class.
Usage:
Book b1 = new Book(“The Kite Runner”, 2019);
Sopln(b1 instanceOf Book);

Output = true

Sopln(tfios instanceOf Book);

Output = false

Java iterator

An Iterator is an object that can be used to loop through collections like arrays etc.
Import from java.util package
iterator() method can be used to get an iterator for any collection.

Methods in iterator

1. hasNext(): returns true if the iterable has more elements. Returns boolean.
2. next(): returns the next element. Throws NoSuchElementException if no element is present. Return type is object
3. remove(): removes next element in iteration. Returns void.

```
// demo of iterator
import java.util.*;

class Student{
    String name;
    long roll;
    Student(String name, long roll){
        this.name = name;
        this.roll = roll;
    }
}

class Main{
    Vector<Student> V = new Vector<>();
    void addStudent(String name, long roll){
```

```
        Student S = new Student(name, roll);
        V.add(S);
    }
    void Show(){
        Iterator<Student> it = V.iterator();
        while(it.hasNext()){
            Student S = it.next();
            System.out.println(S.name);
            System.out.println(S.roll);
        }
    }
    public static void main(String[] args) {
        Main obj = new Main();
        obj.addStudent("Alice", 101);
        obj.Show();
    }
}
```

Selector

Selector describes which operation to perform.
Selector invocation refers to a call of the selector. One argument of the call is used to determine which method is used.

Receiving object: the object used for determination of method by selector invocation.

Scope of access specifiers/modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Non access modifiers

Modifier	Description
`final`	The class cannot be inherited by other classes (You will learn more about inheritance in the Inheritance chapter)
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters)

For **attributes and methods**, you can use the one of the following:

Modifier	Description
`final`	Attributes and methods cannot be overridden/modified
`static`	Attributes and methods belongs to the class, rather than an object
`abstract`	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters
`transient`	Attributes and methods are skipped when serializing the object containing them
`synchronized`	Methods can only be accessed by one thread at a time
`volatile`	The value of an attribute is not cached thread-locally, and is always read from the "main memory"

Transient: some variables are not stored during initialization. For example, in a form, passwords are not stored (skipped) but username etc is.
Synchronized: only one path of execution can access it
Volatile: tells Java that a variable's value is always read directly from main memory, not from a thread's local cache.

Super reference

1. Child constructor is used to call parent constructor
2. First line should use keyword super to call the parent
3. Can also be used to reference other variables and methods.

Example

```
class Animal {
```

```
    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    void sound() {
        super.sound(); // calling parent method by super
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    Cat() {
        super(); // called constructor
        System.out.println("Constructor called");
    }
}

public class Demo { //main class
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();
    }
}
```

Difference:

- super(); → runs the parent constructor → so whatever code is inside Animal's constructor executes.
  If parent constructor is non-parameterized, you need not call super() as the compiler automatically calls it at the time of object creation. But if it is parameterized, you need to call it with super(parameters) and pass parameters.

- super.sound(); → runs only the parent's sound() method, not the constructor or other code.

This keyword

- Used as an **instance** method to refer to the object that has this method. Refer to the current object.

```
class Student {
    String name;
    int age;

    Student(String name, int age) {
        this.name = name;   // 'this' refers to the italicized (for
differentiation) keyword
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + this.name);
        System.out.println("Age: " + this.age);
    }
}
```

Java garbage collection

Java mein garbage = unreferenced objects
Garbage collection is a process of reclaiming unused memory. Used to destroy unused objects.
Use the free() func in c amd delete() in cpp, but java me it happens automatically. So we have better memory management.

- The Java runtime environment deletes objects when it determines that they are no longer required.
- Before an object gets destroyed, the garbage collector lets the object clean up by the finalize() method. This process is known as finalization.

Two approaches of garbage collection

1. Reference counting
    - Reference counting maintains a reference count for every object.
    - Its count changes as references take or leave the object.
    - Ref count=0, object delete
2. Tracing
    - Traces entire set of objects from root
    - all objects having reference on them are marked in some way.

- Objects unmarked after this are marked as unreferenced, and deleted.
- Also called **mark and sweep**

Advantages

1. No worry about cleaning up memory
2. Ensures integrity in programs
3. User cannot access free memory incorrectly
4. It makes java **memory efficient** because the garbage collector removes the unreferenced objects from heap memory.

Disadvantages

1. Have to keep track of which objects are being referenced by the executing program and which are not being referenced
2. More work for finalization and freeing memory of the unreferenced objects.
3. Take up more CPU time compared to manual memory freeing by users

How to unreference an object?

1. Null the reference
   ```
   Employee e = new Employee();
   e = null;
   ```
2. Assign the reference to another object
   ```
   Employee e = new Employee();
   Employee f = new Employee();
   e = f //first object e is free now
   ```
3. By an anonymous object
   ```
   new employee(); //obj has no name
   ```

Finalize() method

//notice the american spelling!!

- finalize() method is invoked before object is “garbage collected”
- Can be used to perform cleanup processing.
- Example:
  ```
  protected void finalize(){}
  ```
- If garbage collector of jvm cleans up objects automatically, whats the use of finalize? JVM can only clear objects that were declared using the new keyword. So if you have objects that were created without new, use finalize() to perform cleanup.

gc() method

- Used to invoke garbage collector
- Found in system and runtime classes
- Example:
  public static void gc(){}

Example code of garbage collection:

```
public class TestGarbage1{
public void finalize(){System.out.println("object is garbage collected");}
public static void main(String args[]){
 TestGarbage1 s1=new TestGarbage1();
 TestGarbage1 s2=new TestGarbage1();
 s1=null;
 s2=null;
 System.gc();
}
}
```

Here, system.gc asks JVM to run the garbage collector. If it runs, it calls the finalize() function. It wont run if the objects are unreachable (not nulled).

Final keyword

- Indicates that a variable/class/method cannot be changed.
- Final variable is a constant. Like double pi=3.14;
- We cannot change post initialization.
  double pi=3.14;
  pi = 4; //not possible, compile time error dega
- Methods that are declared with final modifier cannot be overridden
  Example
  public final void display()
- These methods can be inherited, but not polymorphic.
- Classes with final modifier cannot be extended
  final public class Parent{}
  No inheritance, no polymorphism
- If attempted inheritance/polymorphism, you get compile time error.
- **Performance:** The use of final can sometimes improve performance, as the compiler can optimize the code more effectively when it knows that a variable or method cannot be changed.
- **Security**: final can help improve security by preventing malicious code from modifying sensitive data or behavior.

Aim of final:

1. Variables = constant
2. Classes = prevent inheritance
3. Method = prevent overriding

Static method

Can call a static method without creating object.

```
public static void main(String args[])
{
StaticExample obj = new StaticExample();
// This is object to call non static function

obj.nonstatic(); //method name: nonstatic

// static method can called directly without an object
display();
}
```

Multiple classes

1. Can create as many classes as u want in a single program
2. Number of .class files = number of classes in the program

Class Constructor

1. Special member func of a class which is executed at the time of object creation of that class.
2. Has the **exact same name** as the class.
3. No return type, not even void
4. Three types: argument constructor, parameterized constructor, copy constructor.

Copy constructor

Example:

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

```
    // normal constructor
    Student(int i, String n) {
        id = i;
        name = n;
    }

    // copy constructor
    Student(Student s) {
        id = s.id;
        name = s.name;
    }

    void display() {
        System.out.println(id + " " + name);
    }

    public static void main(String[] args) {
        Student s1 = new Student(101, "Ashwera");
        Student s2 = new Student(s1); // using copy constructor

        s1.display();
        s2.display();
    }
}
```

Why use copy constructor?

1. Create a new object from the existing one
2. Useful for cloning without manual work
3. Safe duplication, v useful for complex objects

Class Destructor

- A destructor is another special member func of a class.
- Executed when an object goes out of scope or the delete expression is applied.
- Same name as the class, prefixed with ~
- No return type, no parameter
- Useful for releasing resource before exiting the program

Overloading

Method overloading

- Can have many diff methods of the same name
- Based on arguments, java decides which method to use
- Example, area(double r) and area (double length, double breadth)
  For circle, first area is invoked, for rectangle, the second area.
- **Definition:** If a class has multiple methods of the same name but different parameters, it is called method overloading.
- Increases readability
- Can differentiate bw methods by changing number of parameters or changing the data type of parameters
- Method overloading is not possible by just changing the return type.
- main() method can also be overloaded

Constructor overloading

- Have multiple constructors with the same name but different signature.
- No return type (cuz constructor)
- These constructors are called overloaded constructors.
- An application of polymorphism.
- Overloading a constructor allows flexibility when creating array list objects for example
- Lets you easily convert one data structure to another. Array list -> set for ex

Overriding

- A child class can override the definition of an inherited method
- The new method should have the same signature (parameter number and type) but can have different body
- The type of the object executing the method determines which version of the method is invoked
- Parent method can be called by using **super**
- Methods marked with **final** cannot be overridden
  Shadowing variables: Variable shadowing happens when a local variable in a method or block has the same name as an instance/class variable, hiding it.

```
class Test {
    int x = 10; // instance variable


    void show() {
```

```
        int x = 20; // local variable shadows instance variable
        System.out.println(x);      // prints 20
        System.out.println(this.x); // prints 10
    }
    public static void main(String[] args) {
        new Test().show();
    }
}
```

- Best practice = avoid shadowing code.

Overriding	Overloading
Run-time polymorphism	Compile-time polymorphism
Occurs between two classes, using inheritance	Occurs within the class
Array elements are stored in contiguous locations, making it easy to determine relative locations of elements.	Insertion and deletion operations are slow, as elements are supposed to be stored sequentially, index-wise.
Methods involved must have the same name and same signature	Methods involved must have the same name but different signatures

Recursion

- Process of defining something in terms of itself
- Method calls itself recursively (baar baar)
- Code example

```
class RecursionExample {
    static int fact(int n) {
        if (n == 0) return 1;   // base case
        return n * fact(n - 1); // recursive call
    }

    public static void main(String[] args) {
        System.out.println(fact(3)); // prints 6
```

```
    }
}
```

Base case = where the recursion calls stop

Working:
If fact(3) is called, it checks if n=0, no, move on and return
3*fact(n-1)
What does fact(n-1) return? 2*(fact(n-1))
So eventually, 3*2*1*1 is returned and since we don’t call anything at n=0, the loop stops.
Can also write base case at n=1.

End of theory

module 3

Questions:
https://docs.google.com/document/d/1i-KLjofTusm1GPDzWPSLhtacaMWeNFns2e_sWqnuVnU/edit?tab=t.0

Arrays

Definition

1. Group of continuous or related data
2. Declaration:
   arr[10];

One dimensional array

1. Uses one subscript
2. Called single subscripted/one dimensional
3. Declare:
   arr[10];
   int[] arr = new int[10];
   Where 10 is the size of the array.
4. a[i] = 2; populating the array.
   Where i is the index, can run from 0 to 9.

Features

- Access array element beyond size = error
  arr[10]=invalid
- Can also be initialized as other variable
  Int var = arr[1];
- Array initializer is a list of values separated by commas and surrounded by curly braces.
  int[]arr={1,2,3,4};
- To access length of array, use arr.length;

Two dimensional array

1. First index=row
2. Second index=column
3. declaration:
   int ar[][];
   ar = new int[2][3];
4. Creates a matrix of 2 rows and 3 columns
5. Refer 1st row ka 1st coln ka element, a[0][0]
   To its left, col changes, so a[0][1] and so on.

//should know declaration, population, and iteration of arrays in code

Array of objects

1. Can have array ka data type as an object
2. Declaration:
   Student arr[] = new Student[10];
3. Add object and add to list simultaneously
   Student[0] = new Student(“ashwera”); //if string s was a parameter in object

//array of objects program vv imp

Jagged array

//prg v imp

- multidimensional array = array of arrays
- Jagged array = multidimensional array such that the size of array of array is variable. Say you wanna store marks of each student but every student took different number of classes. Jagged array.
- Dynamic allocation: Jagged arrays allow you to allocate memory dynamically
- Space utilization: Jagged arrays can save memory, using only enough as needed, unlike dd array.
- Useful when the number of elements in each sub-array is not known in advance.
- Jagged arrays can be faster than rectangular arrays for certain operations, due to more compact memory layout

```
import java.util.*;
class Cricket{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of players.");

        int n = sc.nextInt();
        int x[][] = new int[n][]; //created jagged array here

        int[] sum = new int[n];
        float[] avg = new float[n];
        for(int i=0;i<n;i++)
        {
```

```
            System.out.println("Enter number of matches for player "+(i+1));

            int matches = sc.nextInt();
            x[i] = new int[matches];
            System.out.println("Enter runs scored in match "+(i+1));

            for(int j=0;j<matches;j++)
            {
                int run = sc.nextInt();
                x[i][j] = run;
                sum[i] += x[i][j];
            }
            avg[i] = (float)sum[i]/matches;
        }
        for(int i=0;i<n;i++)
        {
            System.out.println("Player number "+(i+1));

            System.out.println("Runs scored across all matches " + sum[i]);

            System.out.println("Batting average: " + avg[i]);
        }
    }
}
```

Enhanced for loop/ for each loop

1. Syntax: for(int x: marks){//statements}
2. Literal meaning, for each integer x in marks, do this
3. Cannot modify an element using this loop. Gives logic error.

Array List

Definition: The ArrayList class is a resizable array, which can be found in the java.util package.
Size of an array couldn't be modified, array list ka size is changeable.

Create an array list obj

```
ArrayList<String> cars = new ArrayList<String>();
ArrayList<Integer> marks = new ArrayList<Integer>();
```

```
//always use Object datatype
```

Add items

```
cars.add(“Aston Martin DBR1”);
cars.add(“Alto”);
System.out.println(cars); prints entire list
```

Java iterator

Definition: An Iterator is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

- Import from java util
- iterator() method se get an iterator
- Alr discussed in mod2 smh

Vectors

- Util package
- Dynamic array
- Objects can be heterogeneous
- Declaration:
  Vector v = new Vector();
  Vector v = new Vector(size);

Advantages
1. Convenient to store objects
2. Can add and delete as required

Disadvantages:
1. Cannot directly store simple data type
2. Can only store objects

Some Vector Methods are given as follows:

Vector Methods	Description
list.addElement(item)	It adds the item specified to the list at the end.
list.elementAt(n)	It gives the name of the nth object.
list.size()	It gives the number of objects present
list.removeElement(item)	It removes the specified item from the list.
list.removeElementAt(n)	It removes the item stored in the nth position of the list.
list.removeAllElements()	It removes all the elements in the list.
list.copyInto(array)	It copies all items from list to array.
list.insertElementAt(item, n)	It inserts the item at nth position.

Declaring vector

```
Vector<TYPE> list = new Vector()<>;
Vector<TYPE>list = new Vector(size);
Vector<TYPE>list = new Vector(int size, int increment);
```

Functions(contd from image)

- int capacity() = returns capacity of vector
- boolean contains(object element)
- datatype elementAt(index) //returns whatever element is present at i index
- Object firstElement()
- boolean isEmpty()
- Object lastElement()
- void setElementAt (object element, int index) //put element at index
- void setSize(size)
- int size() =return size

To use, syntax:
v.function(); where v is the variable for that vector. If the func has a return type like int size(), store the value in another variable.
example: mysize = v.size();

Enumeration Interface

Definition: The methods by which you can enumerate (visit each item) elements in a collection of objects.

```
boolean hasMoreElement();
Object nextElement();
```

Code implementation:

```java
// Java Program to Iterate Vector using Enumeration
import java.util.*;

public class test {
    public static void main(String a[])
    {
        //creation of vector. string is the data type
        Vector<String> v = new Vector<String>();

        v.add("Welcome");
        v.add("To");
        v.add("Misal");
        v.add("Pav");

        Enumeration<String> x = v.elements();

        while (x.hasMoreElements()) {

         System.out.println(x.nextElement());
        }

        // enumeration is an interface that works like an iterator
    }
}
```

Methods of iterating all elements in a vector

1. For each loop
   ```
   for (Integer x : v) {
                System.out.print(x + " ");
            }
   ```
2. Enumerator
3. Regular for loop. Get size by int n = v.size() then run loop from i=0 to n-1;
4. Using iterator
   ```
   Iterator<Integer> itr = v.iterator();
     // Check until iterator has not reached end
   ```

```
while (itr.hasNext())
{
    System.out.print(itr.next() + " ");
}
```

Difference Between Iterator and Enumeration

Feature	Iterator	Enumeration
Supported Collections	Universal, applicable to all collection classes in the Java Collections Framework.	Limited to legacy classes like `Vector` and `Hashtable`.
Operations Supported	Supports read and remove operations.	Only supports read operation.
Methods	`hasNext()`, `next()`, `remove()`.	`hasMoreElements()`, `nextElement()`.
Fail-Fast Behavior	Provides fail-fast behavior during iteration.	Does not support fail-fast behavior.
Usage in Modern Collections	Preferred for traversing modern collections like `HashMap`, `ArrayList`, etc.	Obsolete in modern applications; primarily used for legacy collections.

Fail-fast: if the collection is structurally modified after the iterator is created (like add/remove), the iterator immediately throws ConcurrentModificationException.

Comparator

Not so important, good to know. Should ideally not specify ke sorting comparator se hi karna hai

```
class SortbyRoll implements Comparator<Student>
{
    // Compare by roll number in ascending order
    public int compare(Student a, Student b) {
        return a.rollno - b.rollno;
    }
} //declaration

Collections.sort(students, new SortbyRoll());//call in main
```

Comparator<Class> lets you define a custom ordering for all objects of the Student class.

Inside, you declare a function compare(a,b) which takes a and b ka roll number and returns unka difference.
If the difference between a and b is negative, it means a is smaller than b, so it tells the Collections.sort(..) function to keep the order. If the difference is positive, we need to swap. If the difference is 0, there is no swap.

The new keyword creates a SortbyRoll object, because Collections.sort needs an actual Comparator instance, not just the class name.
students is a list of objects (or array of objects)

String

- Strings are essentially arrays of characters
- Strings in Java are immutable and final
- The string class provides many functions for manipulating strings
- char s = new char[6]; //declare an array of chars of size 6 or a string of size 6.
- Else, use string class
  String name = new String ("Ashwera"); //by creating an object using new
  String name = "Ashwera"; //or normally
- String is the only class with implicit initialization
- Defined in java.lang package
- For mutable strings, we use StringBuffer class. //implementation nahi ayega
- Indexing same as array. 0 to size-1

Concatenation

```
String s1 = "Ashwera";
String s2 = "Hasan";
String s3 = s1+s2;
System.out.println(s1+" "+s2);//Ashwera Hasan
System.out.println(s3); //AshweraHasan
System.out.println(s1.concat(s2)); //AshweraHasan
```

Array of strings

String arr[]=new String[5];
String array of size 5.

Methods of String

- length() → returns string length.
- charAt(i) → returns char at index i
- substring(i,j) → returns part of string
- indexOf(s) → first index of substring s.
- lastIndexOf(s) → last index of substring s.
- equals(s) → checks exact equality with case.
- equalsIgnoreCase(s) → equality ignoring case.
- compareTo(s) → lexicographic compare. (ash < azh)
- compareToIgnoreCase(s) → compare lexicographic ignoring case.
- toUpperCase() → all uppercase.
- toLowerCase() → all lowercase.
- trim() → remove spaces at ends.
- replace(a,b) → replace chars.
- replaceAll(regex,r) → replace using regex.
- split(regex) → split into array. //see working neeche
- startsWith(s) → true if prefix.
- endsWith(s) → true if suffix.
- contains(s) → true if substring present.
- isEmpty() → true if length 0.
- concat(s) → append string.
- valueOf(x) → convert value to string.

Split function

```
String arr[size] = s.split(,);
```

now, arr is an array of datatype string which contains elements that were separated by a comma in original string s.
s = india, pakistan, nigeria hasan,lovely singh, bodyguard
arr = ["india", " pakistan", " nigeria hasan", "lovely singh", " bodyguard"]

Compare function

Returns a number like it did in comparator. Based on the sign of this integer, compare results can be processed.

String buffer

String buffer is to String what Vector was to array.
Lets you create a string of flexible length jiska length and content is modifiable. Automatically grows to make more room.

Functions

Underlined functions: unique to buffer.
rest= same as string

- length() → returns number of characters.
- charAt(i) → char at index.
- setCharAt(i, c) → set char at index.
- append(s) → add string at end.
- insert(i, s) → insert string at index.
- delete(i,j) → remove substring from i to j-1.
- deleteCharAt(i) → remove char at index.
- replace(i,j,s) → replace substring i to j-1 with s.
- reverse() → reverse entire string.
- capacity() → current buffer capacity.
- ensureCapacity(n) → increase buffer to at least n.
- setLength(n) → set buffer length.
- substring(i) → substring from i to end.
- substring(i,j) → substring from i to j-1.

Note:

1. String is a reference type and not compared by ==. It is important to use the function only
2. String toString() is the built-in method of java.lang which return itself a string. So here no actual conversion is performed. Since toString() method simply returns the current string without any changes, there is no need to call the string explicitly, it is usually called implicitly.

   ```
   Car car = new Car("Toyota", "Corolla", 2020, "Sedan");
   System.out.println(car);
   // With toString(): Make: Toyota, Model: Corolla, Year: 2020,
   Description: Sedan
   // Without toString(): Car@6d06d69c

   Useful if you want to print obj directly
   system.out.println(obj);
   Without string toString, you can access parameters one by one
   System.out.println(obj.name);
   ```

End of theory.

//refer to wrapper classes in java + java adv sorting tab

module 4

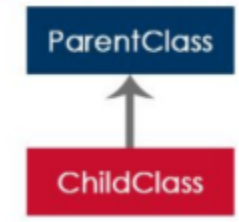
Questions:
https://docs.google.com/document/d/1LndLj63SsTlr9F_Y1cuPHb2v2y9W1jwrWIFMS79-9X8/edit?tab=t.0

Inheritance

Definition: The property of object oriented programming that allows reusability of code. Inheritance helps the derived classes to “inherit” the properties of the base class and hence increases efficiency.

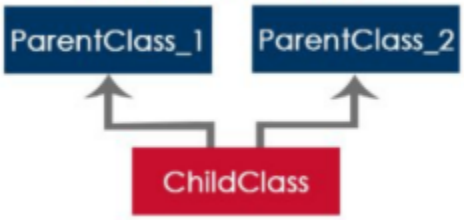
Types

1. Simple inheritance



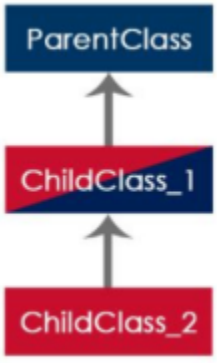
one child one parent

2. Multiple inheritance



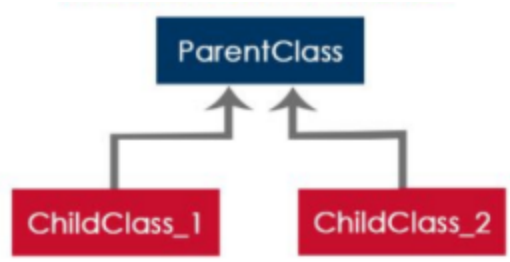
one child more than one parents

3. Multi-level inheritance



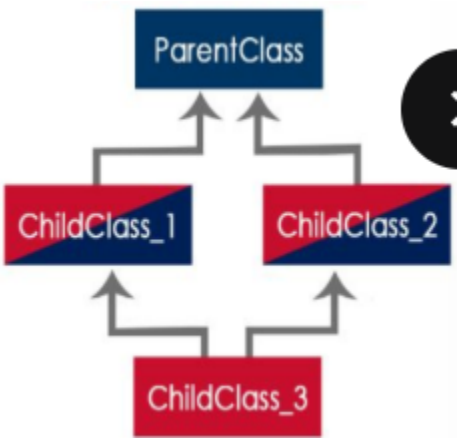
ancestral relationship

4. Hierarchical inheritance



two children same parent

5. Hybrid inheritance



little bit of all

Java does not support multiple inheritance. !!!

Syntax:
class DERIVEDCLASS extends BASECLASS{
... }

Simple inheritance

```
class hyundai{
    String model="Unknown";
    String color = "white";
    int year = 2025;
      double power=160;
}

class car extends hyundai{
    String model = "Creta";
    void display(){
        System.out.println("Car Model: " + model);
        System.out.println("Car Color: " + color);
        System.out.println("Car Year: " + year);
    }
}
```

```
Car Model: Creta
Car Color: white
Car Year: 2025
```

```
public class Main {
    public static void main(String[] args) {
        car myCar = new car();
        myCar.display();
      System.out.println(myCar.power);
    }
}
```

- We can access the members and functions of the parent class in the child class directly.
- If the child class and parent class have the same named member/function, the child class's variable is called
- If you want to call the parent class's variable, use keyword super
  In the above example, if we used super.model in our display() function, we would have printed Unknown instead of Creta.
- If you are creating an object of the child class and have a variable you did not use from the parent class inside the child, you can still call it under your main function. Example: power variable

Multilevel inheritance

```
 class Employee {
    double salary = 50000;
void work() {
System.out.println("Employee is working");
}
}

class Manager extends Employee {
    double salary = super.salary + 30000;
void manageTeam() {
System.out.println("Manager is managing the team");
}
}

class SeniorManager extends Manager {
    double salary = super.salary + 20000;
void makeStrategy() {
System.out.println("Senior Manager is making strategy decisions");
}
```

```
Employee is working
Manager is managing the team
Senior Manager is making strategy decisions
Senior Manager's Salary: 100000.0
Employee is working
Manager is managing the team
Manager's Salary: 80000.0
Employee is working
Employee's Salary: 50000.0
```

```
}

class Main {
public static void main(String[] args) {
SeniorManager seniorManager = new SeniorManager();
seniorManager.work(); // Inherited from Employee
seniorManager.manageTeam(); // Inherited from Manager
seniorManager.makeStrategy(); // Defined in SeniorManager
System.out.println("Senior Manager's Salary: " + seniorManager.salary);

Manager manager = new Manager();
manager.work(); // Inherited from Employee
manager.manageTeam(); // Defined in Manager
System.out.println("Manager's Salary: " + manager.salary);

Employee employee = new Employee();
employee.work(); // Defined in Employee
System.out.println("Employee's Salary: " + employee.salary);
}
}
```

- Here, Employee is the basemost class. (The ancestor)
- Its child is the Manager and Manager's child is Senior Manager
- Basically, the child can always be categorized under its parent
- A senior manager is also a manager, a manager is also an employee
- When we call super.salary at each level, the variable of class JUST above current class in hierarchy is called.
  So salary of employee is 50k, manager is 80k, and senior manager is 1Lakh
- Functions as discussed in simple inheritance can be called in child class anytime.
- Multilevel inheritance = Repeated, stacked simple inheritance

Hierarchical Inheritance

- When two children have the same parent
- Remember how every child class can be categorized under its parent?
- So if the parent is fruit, both apple and mango can be children
- This is the case of hierarchical inheritance

```
//implement hierarchical inheritance
```

```java
class Fruit{
    String season;
    double cost=10;
    void display(){
        System.out.println("Fruit Season: " + season);
        System.out.println("Fruit Cost: " + cost);
    }
}

class Mango extends Fruit{
    String variety= "Alphonso";
    String season = "Summer"; //overriding season variable
    double cost = 20; //overriding cost variable
    void show(){
        System.out.println("Mango Variety: " + variety);
    }
}
class Apple extends Fruit{
    double cost = 15; //overriding cost variable
}

public class Main {
    public static void main(String[] args) {
        Mango mango = new Mango();
        mango.display();
        mango.show();

        Apple apple = new Apple();
        apple.season = "Fall";
        apple.display();
    }
}
```

```
javac Main.java } ; if (
Fruit Season: null
Fruit Cost: 10.0
Mango Variety: Alphonso
Fruit Season: Fall
Fruit Cost: 10.0
```

NOTE:
Fruit season for mango is null

1. Because we declared summer in childclass, which would have reflected if we printed it again in childclass.
2. Since we did not refer to the season variable in child class alone, it did not get called to the updated value

3. Fields do NOT override in Java. display() uses Fruit.season, not Mango.season. This is also why the cost of apple and mango is not updated but printed as per the fruit costs only.

How do modifiers affect inheritance

1. Public -> can be inherited
2. Private -> cannot be inherited
3. Protected -> allows inheritance and provides better encapsulation than public.
4. Default -> allows inheritance ONLY if subclass is in the same package as base class.

Is there a way to access a private member anyway?
Yes-> If you're able to declare a PUBLIC/PROTECTED/DEFAULT function inside the base class that uses the private variable, then you can simply call this function in its bounds and access the private variable indirectly.
Note: final classes can also not be inherited / overridden

Code:

```
class something{
    private int privateVar = 20;
    public void access(){
        System.out.println("Accessing private variable: " + privateVar);
    }
}

class Main{
    public static void main(String[] args) {
        something obj = new something();
        System.out.println(obj.privateVar);
        obj.access();
    }
}
```

//gives error

```
Main.java:11: error: privateVar has private access in something
        System.out.println(obj.privateVar);
                           ^
```

Remove the line:

```
class something{
    private int privateVar = 20;
```

```
    public void access(){
        System.out.println("Accessing private variable: " + privateVar);
    }
}

class Main{
    public static void main(String[] args) {
        something obj = new something();
        // System.out.println(obj.privateVar);
        obj.access();
    }
}
```

Outputs correctly.

```
ccessing private variable: 20
```

Role of Constructor

- constructor of parent class is a function, but its NOT inherited by the child class.
- When a child class object is created, the parent's constructor is automatically called first
  So if in the parent constructor you are initializing some values as age=10, size=34, etc and then the child constructor is being called that overrides age=18, then the final variables hold 18 and 34
- Java automatically calls the parent constructor, but if you want to call it explicitly, use super(); [in case u want to pass values to a parameterized parent constructor]

Code:

```
class parent{
    parent(){
        System.out.println("parent class constructor");
    }
}

class child extends parent{
    child(){
        // super(); <<<<<<<<<<<<<<<<<<<<<<<,
```

```
        System.out.println("child class constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        child c = new child();
    }
}
```

The code runs same with and without the super();

```
parent class constructor
child class constructor
```

If the parent does not have a default constructor, the child has to call the parent's parameterized constructor by passing required values.

- super(); -> parent constructor
- super.method() -> function from parent class
- super.var; -> variable from parent class

When the child and parent have functions with the same name, you can call both together by using reference of object: c.display()
Inside child's display function, add a super.display(); reference.

```
Display method in Parent class
Display method in Child class
```

```
class Parent {
    void display() {
        System.out.println("Display method in Parent class");
    }
}
class Child extends Parent {
    void display() {
        super.display(); // Call to Parent's display method
        System.out.println("Display method in Child class");
    }
}
public class Main {
    public static void main(String[] args) {
```

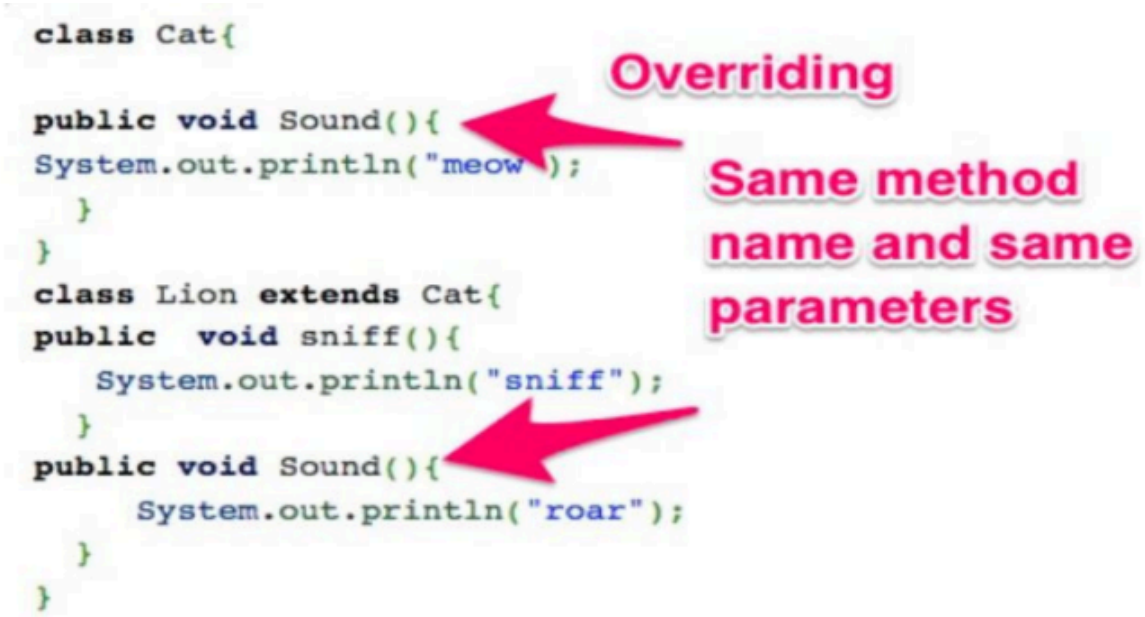
```
        Child c = new Child();
        c.display(); // This will call Child's display method, which in turn
calls Parent's display method
    }
}
```

When super.display() is removed from child class's display func, only the child class print statement is printed.

- The super variable is used to call the constructor of parent class
- The this variable is used to call another constructor in the same class.
- You cannot use both this and super in the same definition. Just one.

Overriding methods

Meaning: When the child and parent have the same method name and same parameters, the child class's method *overrides* the parent class method.



Here, when a Lion object is created and sound function called, it will print roar.

What is the difference between initialization types

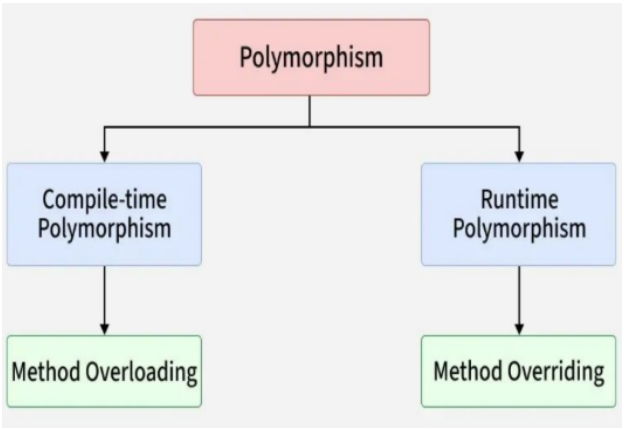
In the above example, we can create objects:

1. Cat a = new Cat();
   This creates an object of compile-time type cat and runtime-type cat also. So this object can access all variables of Cat
2. Lion a = new Lion();
   Same. But if a super method or variable is accessed from Cat, it will run, too. If overridden, only Lion's methods run
3. Cat a = new Lion();
   Compile time type = cat
   Runtime-type = lion
   In this case, you can only use the methods declared in Cat class but the Lion version of that method runs. So in this case, sniff() gives an error and sound prints roar.

Polymorphism

Meaning: Same method or object behaves differently based on context. This is decided based on the parameter number or type passed in said object or func.

Features



1. Multiple behaviour
2. Method overriding: child can redefine parent's methods. **Runtime**
3. Method overloading: Multiple methods, same name, different parameters
   Like how to calculate area of different shapes we need different types/number of parameters. Circle needs 1 (radius), square needs 1(side) but rectangle .needs 2(length and breadth), etc.
   Compile-time
4. Runtime Decision: At runtime, Java decides which function to call based on object's class

Why use

1. Code reusability: why write more code when less code do trick
   Increases usability of code.
2. Flexibility: Allows object of difference classes to be treated as an object of a common superclass.
3. Abstraction: Allows you to work with general types (superclasses and interfaces) instead of concrete types (specific subclasses) thus simplifies the interaction

4. Dynamic behaviour: lets you select appropriate method at runtime, makes program dynamic.

The dynamic method dispatch is the method of selecting which overridden method is to be called at runtime. Based on the object on the right side.
Cat a = new Tiger();
Here, the Tiger class decides the method (method of tiger class is called)

The variables are resolved at compile time and are based on what reference (left side) they're passed with. Variables are NEVER overridden.

```
class Animal {
    int x = 10;
    void show() { System.out.println("Animal show"); }
}

class Cat extends Animal {
    int x = 20;
    void show() { System.out.println("Cat show"); }
    void jump() { System.out.println("Cat jump"); }
}

//print this in main func
Animal a = new Cat();
System.out.println(a.x);
```

Output: 10
Reason: x is a variable and it depends on the reference (left side = animal), hence 10 is printed.

The methods are resolved at compile time and are based on the object (right) that is used to call them.

Same code, run this in main
Animal a = new Cat();
a.show();
Output: Cat show.
Reason: show is a method and it depends on the object(right side = cat)

Conclusion: variables are not polymorphic. If you access it using parent reference you get parent, else you get child.

Method Overloading	Method Overriding
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
Access specifier can be changed.	Access specifier must not be more restrictive than original method(can be less restrictive).
Static methods can be overloaded which means a class can have more than one static method of same name.	Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
private and final methods can be overloaded	private and final methods cannot be overridden.
Return type of method does not matter in case of method overloading, it can be same or different.	*Return type must be same or covariant* in method overriding.
overloading is a compile-time concept	Overriding is a run-time concept

Method Overloading	Method Overriding
Static binding happens when method overloaded .	Dynamic binding happens when method overriding.
Performance better than Overriding. In the case of performance, method overloading gives better performance when compared to overriding because the binding of overridden methods is being done at runtime.	Performance is not better than Overloading.
Method overloading add or extend more to the method functionality.	Method overriding is to change the existing functionality of the method.

Final Keyword

- Used to indicate that no further alteration is allowed
- Final variable becomes a constant after initialization
  final double pi = 3.14;
  pi = 3.147; //throws error
- Cannot be overridden in a subclass
  Cus that just means we are editing the value -> not allowed
  class

```
class Parent {
    final void display() {
        System.out.println("Display method in Parent class");
    }
}
class Child extends Parent {
```

```
    void display() { //wrong
        System.out.println("Display method in Child class");
    }
}
```

Throws error

- Final class cannot be extended:

```
final class Parent {
    void display() {
        System.out.println("Display method in Parent class");
    }
}
class Child extends Parent {
    void display() {
        System.out.println("Display method in Child class");
    }
}
```

Error

Abstract class

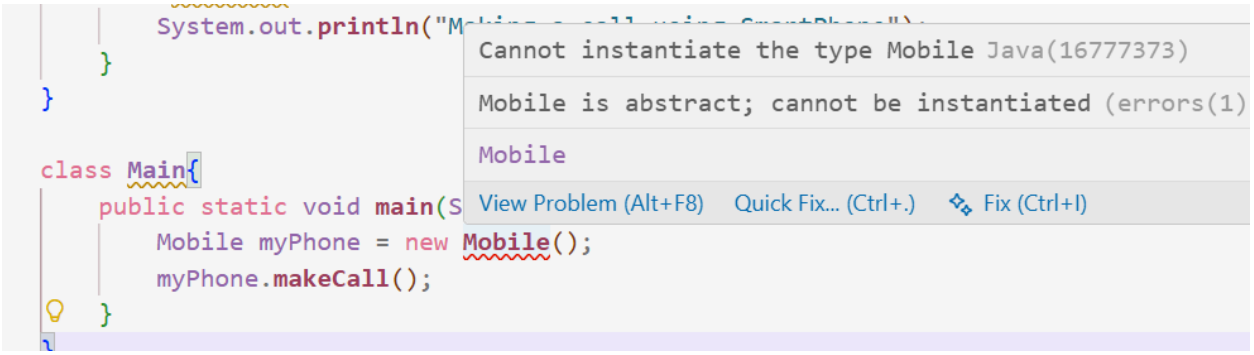
- When a method is ALWAYS redefined in a subclass, it makes overriding compulsory
- Keyword: abstract
- If a class contains an abstract method, the class should be declared abstract
- An abstract class cannot be used to reference objects during initialization:

Correct:

```
abstract class Mobile{
    abstract void makeCall();
}

class SmartPhone extends Mobile{
    void makeCall(){
        System.out.println("Making a call using SmartPhone");
    }
}

class Main{
    public static void main(String args[]){
        Mobile myPhone = new SmartPhone();
```

```
        myPhone.makeCall();
    }
}
```

Wrong:



- Cannot declare an abstract constructor
- Cannot declare an abstract static method
- Any subclass of an abstract class should either
    - Implement all of its superclass functions
    - Itself be an abstract class
- It is not mandatory for an abstract class to ONLY have abstract methods. It can also have regular methods.

Abstract Classes	Abstract Methods
An abstract class is a class that is declared with an abstract keyword.	An abstract method is a method that is declared without implementation.
An abstract class may have both abstract methods (methods without implementation) and concrete methods (methods with implementation)	An abstract method must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
Any class that contains one or more abstract methods must also be declared with an abstract keyword.	An abstract method is a method that is declared without an implementation (i.e., without a body) and is meant to be overridden in a subclass.

Interfaces

- Tells what must be done, now how it is done

```
interface Happiness{
      void makeMoney();
      void eatGood();
}
```

- Any class of this interface should provide the implementation, but the plan remains the same

```
interface Remote {
      void powerOn();
      void volumeUp();
}
```

Samsung and Sony can both implement the interface Remote, but they might have different workings of the methods.

Aim of Interfaces

- Achieve abstraction
- Enable multiple inheritance
- Standardization of methods

Implementation

```
interface Vehicle {
void start();
void stop();
}

class Car implements Vehicle {
public void start() {
System.out.println("Car started");
}
public void stop() {
System.out.println("Car stopped");
}
}

class Main{
public static void main(String[] args) {
Vehicle myCar = new Car();
myCar.start();
myCar.stop();
}
}
```

Prints car started car stopped.
You can also implement multiple interfaces at once:

```
interface Vehicle {
void start();
void stop();
}

interface Electric {
void charge();
}

class Car implements Vehicle, Electric {
public void start() {
System.out.println("Car started");
}
public void stop() {
System.out.println("Car stopped");
}
public void charge() {
System.out.println("Car is charging");
}
}

class Main{
public static void main(String[] args) {
Vehicle myCar = new Car();
myCar.start();
myCar.stop();
}
}
```

module 5

Exception handling, Packages and Multithreading

Exception Handling

Exception: An event that occurs during the execution of the program and disrupts the normal instruction flow.
Java exception: An unexpected event has happened, and now we either need recovery or exit.

Why handle

1. Helps maintain normal desired flow even in an unexpected situation
2. Without java exception handling, programs may crash
3. Data is lost

How to handle

1. **try:** You **try** to execute statements within a block of code
2. If an exception is detected, **throw** an exception
3. Then you optionally execute something, designated by **finally**, which is executed regardless of the exception occurring.
4. **throw** keyword is used to throw an exception. The exception object should contain info about the exception, including type and state when this is called.
   A method creates an exception object and throws it. JVM then looks for a matching catch block. It starts searching in the same method, then goes up the call stack. A catch matches if it handles the same exception type or a parent type. If nothing matches, the program ends.

A try block can be followed by one or more catch blocks, each specifying a different type.

The Throwable Class Is The Superclass Of All Errors And Exceptions In The Java Language. Only Objects That Are Instances Of This Class (Or One Of Its Subclasses) Are Thrown By The Java Virtual Machine Or Can Be Thrown By The Java Throw Statement. Example: java.lang.ArithmeticException: / by zero

Threads

- A thread in Java is simply one independent path of execution inside a program.
- When a Java program starts, the JVM automatically creates one thread called the main thread.
- That’s why your main() method runs without you creating any thread manually.
- If you create more threads, the program can run several pieces of code at the same time.
- The method sleep() is used to halt the working of a thread for a given amount of time.

How create?

- Thread class
- Runnable interface
- When you call start(), the JVM schedules that thread and runs its run() method on its own execution path.

If a thread throws an exception and that exception is not caught inside a try-catch within that thread, the thread will terminate. Since there are multiple threads in the program, the program goes on.

Defining exception types

- Must extend Throwable / its subclass (Error/exception)
- Throwable is the root class of exception hierarchy
- Error consists of abnormal conditions. They cannot be handled, and always lead to program termination. Example: StackOverFlowError
- Exception consists of abnormal conditions that can be handled explicitly.

Difference between Checked and Unchecked Exception

Checked Exceptions	Unchecked Exceptions
Occur at compile time.	Occur at runtime.
The compiler checks for a checked exception.	The compiler doesn't check for exceptions.
Can be handled at the compilation time.	Can't be caught or handled during compilation time.
The JVM requires that the exception be caught and handled.	The JVM doesn't require the exception to be caught and handled.
Example of Checked exception- 'File Not Found Exception'	Example of Unchecked Exceptions- 'No Such Element Exception'

Advantages

1. Provision to complete program execution
2. Easily identify code
3. Propagates errors
4. Meaningful error reporting
5. Identify error type -> easy debug

Checked and unchecked exception

Unchecked

- Exceptions instantiated from RuntimeException class are called unchecked exceptions.
- You can optionally handle or ignore these
- If you ignore and it occurs, program terminates
- If you handle it, the code you write under try-catch runs.

Checked

- Instantiated from the Exception class.

- Must be handled using try-catch or be put under a throws clause so that any method can **throw** them
- Cannot be ignored!!
- If not handled, the program does not compile in the first place.

If the code in a method can throw a checked exception, and the method does not provide an exception handler for the type of exception object thrown, the method must declare (using throws) that it can throw that exception.

Users of a method must know about the exceptions that a method can throw so they can handle them. Thus, you must declare the exceptions that the method can throw in the method signature.

A method must consider exceptions it throws and exceptions thrown by any methods it calls (and those methods call).

Constructors under Throwable

1. Throwable ()
2. Throwable (String msg)
3. Throwable (String msg, Throwable clause)
4. Throwable (Throwable clause)

throw keyword

Used to explicitly throw an exception.
Can be used where according to your intended logic, an error is to be thrown.

```
public class ExceptionDemo {
    static void canVote(int age){
    if(age<18)
        {
            try
            {
                throw new Exception();
            }
            catch(Exception e)
            {
                System.out.println("you are not an adult!");
            }
        }
    else System.out.println("you can vote!");
    }
```

```
    public static void main (String[] args) {
        canVote(20);
        canVote(10);
    }
}
```

```
you can vote!
you are not an adult!
PS C:\Users\sveda\OneDri
```

Java throws keyword

- Used when you dont want to handle the exception but you want to state that it throws some exception/error.
- You tell the caller that this function throws an exception, it's upto the caller to deal with it as they want.
- Used to handle checked exceptions as the compiler wont allow code to compute till they've been handled.

```
public class ExceptionDemo {
    static void func(int a) throws Exception
    {
        System.out.println(10/a);
    }
public static void main (String[] args) {
    try{
        func(5);
        func(0);
    }
    catch(Exception e){
        System.out.println("can't divide by zero");
    }
    }
}
```

```
2
can't divide by zero
```

Here, function func stated that it throws an exception. This exception is well known and checked, that terminates program and prints cannot divide by

zero (by JVM). But since we mentioned that it throws an exception, the caller dealt with it using a try-catch block.

Java Throw vs Throws

Throw	Throws
This keyword is used to explicitly throw an exception.	This keyword is used to declare an exception.
A checked exception cannot be propagated with throw only.	A checked exception can be propagated with throws.
The throw is followed by an instance and used with a method	Throws are followed by class and used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions

User defined exceptions

- All exceptions must be a child of Throwable.
- If you want your custom exception to be a checked exception, Java requires you to extend the Exception class (not RuntimeException). Only then the compiler will enforce the handle-or-declare rule for it.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

Syntax:

```
class myException extends Exception{
} //FOR CHECKED EXCEPTION

class myException extends RuntimeException{
}//for unchecked exceptions
```

Why use custom exceptions

1. Predefined java exceptions generate report in their predefined format only.
2. After generating report, it terminates program.

3. You can generate exception report (highly specific and unique to different applications) in your desired format and you automatically resume the prog after generating the error report.

Example

Main prog:

```
public class CheckingAccount {
    private double balance;
    private int number;

    public CheckingAccount(int number) {
        this.number = number;
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount <= balance) {
            balance -=amount;
        } else {
            double needs = amount - balance;
            throw new InsufficientFundsException(needs);
        }
    }

    public double getBalance() {
        return balance;
    }
    public int getNumber() {
        return number;
    }
    public static void main(String[] args) {
        CheckingAccount account = new CheckingAccount(12345);
        account.deposit(500);
        System.out.println("Account Number: " + account.getNumber());
        System.out.println("Initial Balance: Rs." + account.getBalance());

        try {
```

```
            System.out.println("Withdrawing Rs. 600...");
            account.withdraw(600);
        } catch (InsufficientFundsException e) {
            System.out.println("Insufficient funds. You need Rs." + e.getAmount()
+ " more to complete this transaction.");
        }

        System.out.println("Final Balance: Rs." + account.getBalance());
    }
}
```

Exception handling prog:

```
// File Name InsufficientFundsException.java
public class InsufficientFundsException extends Exception {
    private double amount;

    public InsufficientFundsException(double amount) {
        this.amount = amount;
    }
    public double getAmount() {
        return amount;
    }
}
```

Output

```
javac CheckingAccount.java } ; if ($?) { java CheckingAccount }
Account Number: 12345
Initial Balance: Rs.500.0
Withdrawing Rs. 600...
Insufficient funds. You need Rs.100.0 more to complete this transaction.
Final Balance: Rs.500.0
PS C:\Users\syeda\OneDrive\Desktop\personal>
```

Important: the exception class is created separately in a different file and the main program that throws this exception is in a different file!!!

Array out of bounds exception

Whenever we are trying to access any item of an array at an index which is not present in the array.
Example:

- arr[-1] //array indices start at 0
- arr[5] when size of array=5 (array index ends at size-1)

Nested try

Try inside a try

```
class Main {
    public static void main(String[] args) {
        try {
            try {
                int[] a = {1, 2, 3};
                System.out.println(a[3]);
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Out of bounds");
            }

            System.out.println(4 / 0);
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException : divide by 0");
        }
    }
}
```

```
javac Main.java } ; if ($?) { java
Out of bounds
ArithmeticException : divide by 0
```

Summary

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Packages

Definition: A collection of related classes. Can also contain subpackages.
Use:

1. Group related classes together
2. Provide a layer of protection
3. Keep pieces of project down to manageable sizes
4. Avoid name collisions
5. Reusability of classes

Types:

1. User defined
2. System packages/ java API (Application programming interface)

System packages

- These are built in packages that contain multiple classes and interfaces.
- Classes are predefined and ready to use. Just have to import the relevant package in the program.
  Example: Util package contains the Scanner class that helps us take input. We simply import java.util.Scanner and create a scanner object to begin working with this class, never having to worry about how the Scanner class is implemented.
- Examples
  - Java.lang: Automatically imported. Include classes for primitive datatypes like string, math functions, exceptions, and thread
  - java.util: utility classes like vector, hash tables, random numbers, date, scanner
  - java.io: input output classes.
  - java.awt: used to implement GUI. classes for windows buttons lists etc
  - java.net: classes for networking.
  - java.applet: for creating and implementing applets.

User-defined packages

Packages are organized in hierarchy. The users can create their own packages, called user-defined packages. These can also be imported into other classes the same way JVM packages are, and can be used similarly.

Example

```
package myPackage;
```

```
public class Trigonometry {
        public static double Sin(int x){
            switch(x){
                case 0: return 0;

                case 30: return 0.5;

                case 45: return 0.7071;

                case 60: return 0.8660;

            }
            return -1;
        }
        public static double Cos(int x){
            switch(x){
                case 0: return 1;

                case 30: return 0.8660;

                case 45: return 0.7071;

                case 60: return 0.5;

            }
            return -1;
        }
        public static double Tan(int x){
            switch(x){
                case 0: return 0;

                case 30: return 0.5774;

                case 45: return 1;

                case 60: return 1.7321;

            }
```

```
        return -1;
    }
    public static double Cot(int x){
        switch(x){
            case 0: return Double.POSITIVE_INFINITY;

            case 30: return 1.7321;

            case 45: return 1;

            case 60: return 0.5774;

        }
        return -1;
    }
    public static double Csc(int x){
        switch(x){
            case 0: return Double.POSITIVE_INFINITY;

            case 30: return 2;

            case 45: return 1.4142;

            case 60: return 1.1547;

        }
        return -1;
    }
    public static double Sec(int x){
        switch(x){
            case 0: return 1;

            case 30: return 1.1547;

            case 45: return 1.4142;

            case 60: return 2;

        }
```

```
            return -1;
        }
    }
```

This is the package class.

```
import myPackage.Trigonometry;

public class Main {
    public static void main(String[] args) {
        int angle = 45; // angle in degrees

        double sineValue = Trigonometry.Sin(angle);
        double cosineValue = Trigonometry.Cos(angle);
        double tangentValue = Trigonometry.Tan(angle);

        System.out.println("Sine of " + angle + " degrees: " + sineValue);
        System.out.println("Cosine of " + angle + " degrees: " + cosineValue);
        System.out.println("Tangent of " + angle + " degrees: " + tangentValue);
    }
}
```

This is the class that imports the package class
Important: the package has to be inside a folder called myPackage or something. Then, the classes that import these have to be OUTSIDE the myPackage folder. myPackage can have multiple classes in the same folder. Then, all these classes can be imported by importing myPackage.*;

Advantages

1. Useful to arrange related classes and interfaces in a group
2. Packages hide the classes and interfaces to avoid accidental deletion/interference
3. Isolated from the classes of other packages: can use the same name for classes of two different packages
4. A group of packages = library. Can be reused again and again.

Accessing:

```
import java.lang.Math
//imports entire math class.
```

```
import java.lang.*
//imports entire lang package
import static java.lang.Math.sqrt;
//imports only the sqrt method
```

Multithreading

Process

A process is a program in execution. It is a unit of computation. It contains the code to be run, the data on which it executes, and the status of the execution, as well as the resources required to run the process.

Thread

It is a lightweight process. It shares the same address space with other threads. A thread is the smallest segment of an entire process. In process execution, it involves a collection of threads, and each thread shares the same memory. Each thread performs the job independently of another thread.
Example:
One process = kitchen.
Many threads = cook making rice, cook frying veggies, cook making soup.
All work in the same kitchen (shared space) but do their tasks independently.
If the rice-cook passes out, it does not affect the vegetables from being fried or soup from being made.

Concurrency and Parallelism

Concurrent multithreading systems give the appearance of several tasks executing at once, but these tasks are actually split up into chunks that share the processor.
Example: On a single-core laptop, you browse the web, play music, and download a file. They look simultaneous, but the CPU quickly switches between them in tiny time slices.

In parallel systems, two tasks are actually performed simultaneously.
Parallelism requires a multi-CPU system.

Multitasking

Multitasking OS run multiple programs simultaneously. Each of these has at least one thread within it. One thread = single threaded process.

- The process begins at a well known point: in java and c/c++ at the main function
- Execution follows in an ordered sequence
- While executing, the process has access to data

A program with multiple threads is a multitasking system within an OS

- Each thread's execution begins at a predefined known location. For the main thread, this location is main function. For the other threads, a particular location is decided by the programmer when the code is written.
- The thread executes in an ordered, predefined sequence
- Thread works independently
- Threads have simultaneous execution to a specific degree.

Why multithreading?

1. Speeds up computation
   Two threads = each solves half program = combines result = speed high
   Only works in multiprocessor systems
2. Faster response
   One thread computes while the other handles the UI. One thread loads an image while the other performs computation. Work split = faster result

Thread support in Java

Thread class

Can either be done by extending the lang.Thread class or by implementing the lang.Runnable interface.
Action of thread takes place at the run() method
Execution of thread begins at start() method.

Functions

- sleep(long time) - causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- yield()- causes the currently executing thread object to temporarily pause and allow other threads to execute.
- wait()- causes current thread to wait for a condition to occur (another thread invokes the notify method or the notifyAll method for this object).
- notify() - notifies a thread that is waiting for a condition that the condition has occurred.
- notify All()- like the notify() method, but notifies all the threads that are waiting for a condition that the condition has occurred.
- start()- initiates execution of the thread
- currentThread()- returns reference to the currently executing thread object
- run()- triggers an action for thread

- isAlive()- to verify if the thread is alive
- suspend()- instantly suspend thread execution
- resume() resume execution of a suspended thread
- interrupt()-triggers an interruption to the current thread
- destroy()- destroys the execution of a group of threads
- stop()- stop the execution of a thread.

Lifecycle of thread

1. start()- creates the system resources necessary to run the thread. Calls the thread's run() method.
2. The thread becomes not runnable when
    a. The sleep() method is invoked
    b. The thread calls the wait() method
    c. The thread is blocked from IO ops.
3. Automatically dies when run() exits

States of a thread

- New: thread allocated and waiting for start()
- Runnable: can begin execution
- Running: currently executing
- Blocked: waiting for an event
- Dead: thread finished

Thread priority

- On a single CPU, threads run once at a time and provide an illusion of concurrency.
- Execution of multiple threads on a cpu in some order is called scheduling
- The java runtime supports an algorithm for scheduling, which schedules threads based on priority.
- The runtime system chooses the runnable thread with the highest priority first.

Java thread types

1. User
2. Daemon
   Low-priority threads that run in the background to perform crass tasks like cleaning garbage or helping other user threads

Cycle:
All threads finish -> daemon threads terminate -> main prog terminates

The **scheduler** decides which runnable threads it should run. It is based on thread priority.
The **scheduling policy** can be of two types: non-preemptive/cooperative scheduling and preemptive scheduling.

1. Non-preemptive:
   Threads continue till thread terminates.
   CPU is given to a process and it keeps running until it finishes or blocks. CPU won't be taken away forcibly.
2. Preemptive:
   CPU can be taken from a running process at any time by the OS. Used when a higher-priority process arrives.

Advantages

1. Improves performance and reliability
2. Minimizes execution period
3. Smooth and hassle free GUI
4. Lower software maintenance costs

Disadvantages

1. Poses complexity in debugging
2. Increases probability of deadlock
3. Unpredictable results
4. Can have complications if code is being ported

Usage:

```
public class newThread extends Thread {
    public void run() {
    for (int i = 0; i < 3; i++){
        System.out.println(i);
        try {
            sleep((int)(Math.random()*1000)); // 8 secs
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        }
    }
    public static void main(String[] args) {
    new newThread().start();
    new newThread().start();
    System.out.println("Done");
```

```
    }
}
```

Thread vs. Runnable

1. If you extend the Thread class, you cannot extend another class as multiple inheritance is not allowed in Java.
   But if you implement the Runnable interface, you can extend other base classes
2. Basic functionality like yield(), interrupt() etc are in the Thread class only.
3. Using Runnable gives you an object that can be shared.

Race condition

A race condition is a condition of a program where its behavior depends on relative timing or interleaving of multiple threads or processes.

Nondeterministic means the outcome is not fixed or predictable.
Different runs can give different results, even with same inputs. <example is the code above with 8000ms>

Thread-safe is a program that is free of race conditions.

Thread Synchronization

- Solves problem of race condition
- When two or more threads work on the same data simultaneously, they need to be synchronized to prevent unpredictable output.
- Example:
  Two threads are trying to update the same shared variable simultaneously:
  The result is unpredictable.
  The result depends on which of the two threads was the last one to change the value.
  The competition of the threads for the variable is called race condition.
  The first thread is the one who wins the race to update the variable.

Synchronize Method

Java has synchronized methods. Once a thread enters this block, no other thread can call that method on the same object. All threads have to wait until the thread finishes and comes out of the synchronized block. This way, you get a multithreaded application that yields predictable results.

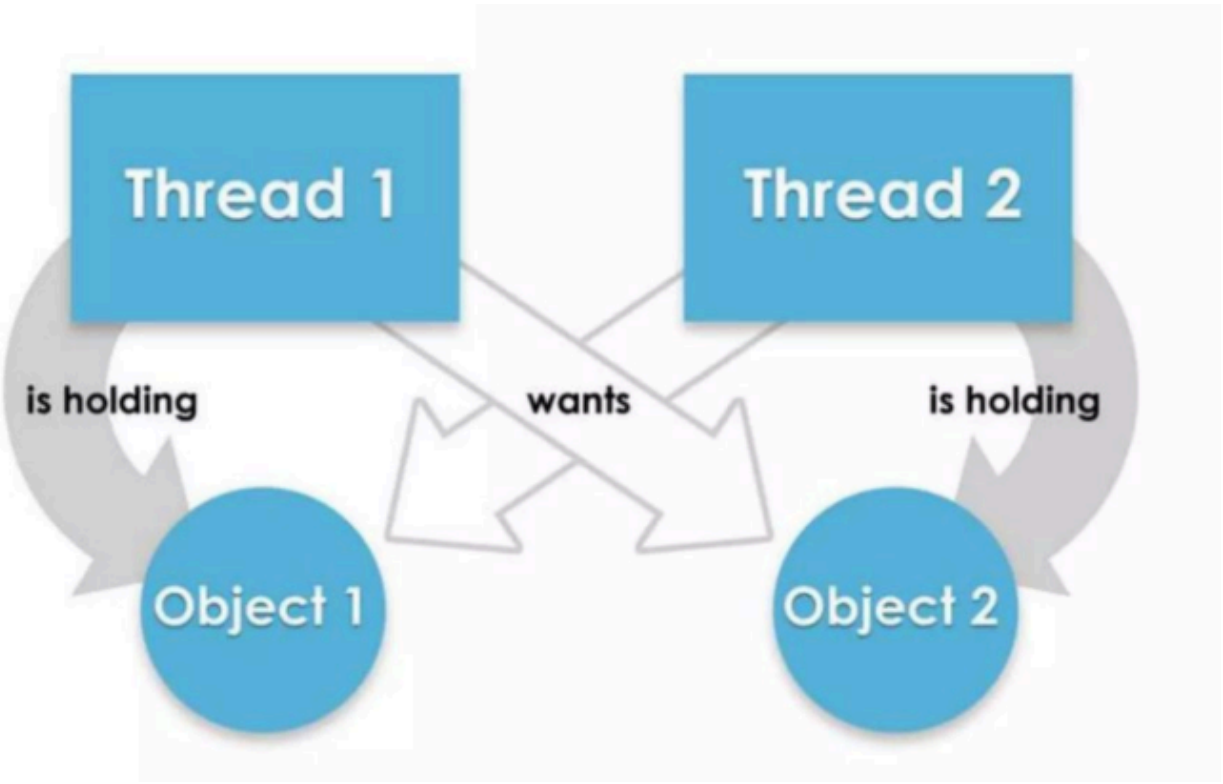
Syntax:
Synchronized(object)
{
}

Usage:

```
public class Counter {
    private static int count = 0;

    public synchronized int getCount() {
        return count;
    }

    public synchronized void setCount(int count) {
        this.count = count;
    }
    public static void main(String[] args) {
        Counter counter = new Counter();
        counter.setCount(5);
        System.out.println("Count: " + counter.getCount());
    }
}
```

Deadlock

When two or more threads are blocked forever and are waiting for each other.


Thread 1
Thread 2
is holding
wants
is holding
Object 1
Object 2

Classroom Codes

```
#include <iostream>

class Addition
{
    int a,b;
    public:
    void read()
    {

    }
    void display()
    {

    }
    void main()
    {

    }
    //object : functions
    //their property: int a,b
    //class: addition
    Abstraction: show some properties and hide the rest. example: doors in a car and engine of car. purpose: information remains unleaked
    Access specifiers:
    1. public: everyone
    2. protected: access to an extent
    3. private: only to user/default access
    Encapsulation: wrapping of data and function into a single unit. protects integrity of objects data. separates internal from external
    Inheritance: reusability
Types:
1. Single: one parent, one child
2. Multiple: Single parent many children
3. Multilevel : Hierarchy of ancestors
};

class Base
{
    public:
    protected:
```

```
    private:
};

//define a class called employee add out public private and protected data
members of this class. make sure your class
//also has key methods.

class Employee
{
    public:
    char* name;
    char* email;
    char* dept;

    private:
    long mobile;
    char* address;
    char* password;

    protected:
    long salary;
    char* role;
    long id;
}
```

```
namespace: avoid name conflict
automatic conversion: implicit
int z=10;
double y = x+5.5; //x converts to double

explicit conversion:
int a=5,b=2;
double result = double(a)/b; //returns 2.5 as one element is double

in a switch case, the variable that goes inside switch is called selector.
switch(x)
{
    case 1: break; //etc. x is the cylinder
}
```

```
loops in cpp:
1. while
int i=0;
while(i<5)
{
    //do something
} i++;
2. for
for(int i=0;i<n;i++)
{
    // do something
}
3. do while
int i=0;
do{
    // do something
} while(i<n);

// functionality of while and for are the same.
// dowhile will run minimum once, and the rest can run 0 times also.
//for loops can be infinite when the condition and/or update statement is
left empty.
// in practice, for is slower than while


FUNCTIONS IN CPP

ind add(int a,int b)
{
    return a+b;
}

//syntax:
datatype funcname(paremeters)
{
    // do something
    return datatype;
}
```

```
//calling func:

int main()
{
    cout << add(2,3);
    // prints whatever func add returns
}

if you predeclare a variable:
function with default arguments/parameters
ind add(int a,int b=4)
{
    return a+b;
}

int main()
{
    cout << add(2);
    // prints 7 as b is already 5
    cout << add(2,3);
    //prints 5. earlier written b=5 is overwritten by b=3
}

function overloading:
using the same fnction name but with different parameters
example of implementation of polymorphism

int add(int a,int b)
{
    return a+b;
}
double add(double a, double b)
{
    return a+b;
}

which function is called if the name is same?
depends on which value user sends.
if user sends two different value, and they can be implicitly converted to
the other {eg- int to double}, then the double func
```

will be called

question: use the concept of function overloading to compute the area of circle rectangle and triangle.

```
#include <bits/stdc++.h>
using namespace std;

double area(int side)
{
    cout << 3.14*side*side << endl;
}

double area(int l, int b)
{
    cout << l*b << endl;
}

double area(double base, double height)
{
    cout << .5*base*height << endl;
}

int main() {
    cout << "enter shape: " << endl;
    int op;
    cin >> op;
    switch (op)
    {
    case 1:
        cout << "enter side of circle\n";
        int s;
        cin >> s;
        area(s);
        break;
    case 2:
        cout << "enter sides of rectangle\n";
        int s1,l;
        cin >> s1 >>l;
        area(s1,l);
```

```
        break;

    case 3:
        cout << "enter sides of triangle\n";
        int a,b;
        cin >> a >>b;
        area(a,b);
        break;

    default:
        break;
    }
}
```

question: write a program to reverse a two digit number

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int num,rev;
    cin >> num;

    do{
        int dig = num%10;
        num/=10;
        rev = (rev*10) + dig;
    } while(num>0);

    cout << rev << endl;
    return 0;
}
```

25/7/25

Simple math operations

```
// write a program using java to print roots of quadratic equation
import java.util.Scanner;
```

```
class roots{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the equation's a, b, and c parameters as
per ax2+bx+c");
        double a = sc.nextDouble();
        double b = sc.nextDouble();
        double c = sc.nextDouble();
        double discriminant = (b*b)-(4*a*c);
        discriminant = Math.sqrt(discriminant);
        double x = (-b + discriminant)/(2*a);
        double y = (-b - discriminant)/(2*a);
        if(x<0||x>0 && y<0 ||y>0)
        System.out.println("Roots are "+x + " and  "+y);
        else
        System.out.println("Roots do not exist");
    }
}
```

Loops

```
// write a program using java  first 10 prime numbers
import java.util.Scanner;
class roots{

    static Boolean isPrime(int x)
    {
        if(x<2) return false;
        for(int i=2;i<x;i++)
        {
            if(x%i==0)
            return false;
        }
        return true;
    }

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

```
        int n=2;
        for (int i = 0;i<10;) {
            if(isPrime(n)==true)
            {
                System.out.println(n);
                i++;
            }
            n++;
        }
    }
}
```

Parameterised Constructor

```
// create a default cnostructor for calss student
//define parameterised constrcutor

class Student{
    int roll;
    public Student()
    {
        roll=0;
    }

    public Student(int x)
    {
        int rollnumber = x;
    }

    public Student(OtherStudent other) //otherstudent is the other class where
from copy the cosntructor elements
    {
        this.data = other.data;
    }

    //in cpp: copy constructor:
    // Student(const Student &obj) {
    //     data = obj.data;
```

```
    // }

    public static void main(String args[])
    {
        Student s1 = new Student();
        Student s2 = new Student(101); //create object using constructor
    }

    // // in cpp:
    // // int main()
    // {
    //     Student S1;
    //     S1.study(); //assumes taht study() is a declared func inside class
Student
    // }
}
```

//8th aug new program:

Object as a parameter

```
//write a prog ram to compute subtraction of a complex number
//use the terminology of object as a parameter and return an object
import java.util.*;
class Complex {
    int real, imag;
    static Scanner sc = new Scanner(System.in);

    Complex(int r, int i) {
        real = r;
        imag = i;
    }

    static Complex subtract(Complex c, Complex d) {
        int r1 = c.real - d.real;
        int i1 = c.imag - d.imag;
        Complex ob = new Complex(r1, i1);
        return ob;
```

```
    }
    void display()
    {
        System.out.println(real + " + " + imag + "i");
    }
        public static void main(String[] args) {
        System.out.println("Enter two complex numbers in the format a b if
complex number= a+ib");
        int i = sc.nextInt();
        int j = sc.nextInt();
        Complex c1 = new Complex(i,j);
        i = sc.nextInt();
        j = sc.nextInt();
        Complex c2 = new Complex(i,j);
        Complex result = subtract(c1,c2);
        result.display();
    }
}

// define a class student  with data members = name, marks1, marks2
// a method that takes student s as an object so that cumulative marks are
displayed for all students
// compare bw two students, returns topper object


import java.util.*;
class Student
{
    String name;
    int marks1, marks2;

    Student(String s,int m,int n)
    {
        name = s;
        marks1 = m;
        marks2 = n;
    }

    static Student compare(Student Student1, Student Student2)
```

```
    {
        if(marks(Student1)>marks(Student2)) return Student1;
        else return Student2;
    }

    static int marks(Student S1)
    {
        int total = S1.marks1 + S1.marks2;
        return total;
    }

    public static void main(String[] args) {
        System.out.println("Enter the name of two students and their marks in 2
subjects");
        Scanner sc = new Scanner(System.in);
        String s = sc.next();
        int m = sc.nextInt();
        int n = sc.nextInt();
        Student S1 = new Student(s,m, n);
        s = sc.next();
        m = sc.nextInt();
        n = sc.nextInt();
        Student S2 = new Student(s,m, n);
        Student Topper = compare(S1,S2);
        System.out.println(Topper.name);
    }
}

// define a player class with runs name and matches
// method better average (player P) accepts an obj as parameter
// and returns player with better average
// second method getaverage(player P) that accepts an obj as parameter
// and returns the average runs per match of player p
```

Call by reference vs. Call by value

```
void modify(int x)
{
```

```
    x = 10;
}

int main()
{
    int a = 5;
    modify(a);
cout << a << endl;
}
```

//here, a does not change to 10 inside main(). when we print 'a', we get 5 only. Passed by value.

```
void modify(int &x)
{
    x = 10;
}

int main()
{
    int a = 5;
    modify(a);
cout << a << endl;
}
```

BUT when we accept the address of a variable "&" by using ampersand, the actual value is changed at the memory location. This prints 10. Passed by reference.

Recursion

Factorial of a number upto size terms using recursion

```
#include <bits/stdc++.h>
using namespace std;

int factorial(int a, int b, int size)
{
```

```
    if(size---==0)
    {
        return b;
    }
    cout << b << " ";
    return factorial(b,b+a,size);
}

int main()
{
    factorial(0,1,10);
}
```

Wrapper classes in java

```java
// wap to translate girafi language : a e i o u are replaced with z

import java.util.Scanner;
public class translator {
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter string : ");
        String s = sc.nextLine();
        String s1 = "";
        s = s.toLowerCase();
        for(int i=0;i<s.length();i++)
        {
            char ch = s.charAt(i);
            if(ch=='a' || ch=='e' || ch=='i' || ch=='o' || ch=='u')
            {
                s1 = s1 + 'z';
            }
            else{
                s1 = s1 + ch;
            }
        }
        System.out.println("Translated string:" + s1);
    }
}
```

Output:
Enter string :
ashwera hasan
Translated string:zshwzrz hzszn

Theory

Vectors cannot handle primitive data types like float int etc. Primitive data types may be converted into object types by using the wrapper classes contained in the java.lang packages.

Each primitive type can be converted to its object. Syntax mein only difference is capitalization.

primitive type: int
Object: Integer and so on

JDK 5.0 provides autoboxing and auto-unboxing of primitive data types.

Autoboxing:

The automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double, etc.

```
char ch = ‘a’;
Character a = ch;  //autoboxing
```

Unboxing

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

```
Character a = ‘A’;
char ch = a; //unboxing
```

```
int a = list.get(0); //gets the 0th element in list, converts object to primitive data type.
```

Examples:

```
list.add(13); // with autoboxing
list.add(new Integer(25)); // without autoboxing
int tmp = list.get(0); // with autounboxing
int tmp2 = list.get(0).intValue() //without
```

All yield the same result, the only difference is declaring explicitly by user or implementing implicitly by computer.

Java advanced Sorting Techniques

Comparator and Comparable

The comparator and comparable interfaces allow you to specify any rule you want to employ in sorting elements.
This eliminates the need for manual sorting algorithms like bubble sort etc.
Being able to specify your own rule for sorting lets you have absolute control over how the elements are sorted.

The compare() method is a method that decides which object among two should go first in the list. It returns a number.

- Negative: If the first object should go first
- Positive: If the second object should go first
- Zero: Order doesn't matter

Comparable

Comparable is used when a class wants to define its own natural sorting order.
public int compareTo(T o)
Collections.sort() or Arrays.sort() will automatically use compareTo() when the objects implement Comparable.
One class can have only one natural ordering because you can define only one compareTo method.

Usage:

```
class Student implements Comparable<Student> {
    int marks;
    public int compareTo(Student s) {
        return this.marks - s.marks; // sort by marks
    }
}
```

Comparator

Comparator is used when you want multiple sorting methods or when the class cannot or should not implement Comparable.
public int compare(T o1, T o2)

```
class SortByName implements Comparator<Student> {
    public int compare(Student a, Student b) {
        return a.name.compareTo(b.name);
```

```
    }
}
```

All theory questions

Module 1

1. What is the difference between unstructured, structured, and modular programming? Give one example of each.
2. Define encapsulation. Explain how it enforces abstraction with an example.
3. Explain data abstraction with a real-world analogy.
4. What are access specifiers? Explain public, private, and protected with examples.
5. What is inheritance? Explain superclass, subclass, and reusability.
6. What is polymorphism? Give one example each of compile-time and runtime polymorphism.
7. Define upcasting. Why is it also called dynamic method dispatch?
8. Explain method overriding. Mention any two rules.
9. Explain high cohesion and low cohesion with examples.
10. What is loose coupling? How can it be achieved?
11. What is an exception? Why is exception handling needed?
12. Explain try–catch–finally with syntax.
13. What is the difference between primitive and reference data types?
14. Explain type conversion and type compatibility with examples.
15. What is the purpose of the new operator? Explain with code.
16. What is a package? How is import used in Java?
17. Explain procedural programming and object-oriented programming in detail. Compare both in a table.
18. Discuss all **Object Oriented Features** (encapsulation, inheritance, polymorphism) with suitable examples.
19. Explain data hiding and information hiding in Java. Why is it important?
20. Describe cohesion and coupling. Explain all types of cohesion and coupling with examples.
21. Explain runtime polymorphism in Java. Cover overriding, upcasting, dynamic binding.
22. Explain exception handling in Java. Discuss checked and unchecked exceptions, hierarchy, and examples.
23. Explain the Java memory model: primitive default values, object initialization, null, arrays, and garbage collection.

Module 2

q1. Define a class and an object. Explain state, behavior, and identity.
q2. What is the `instanceof` operator? Give two examples.
q3. What is an Iterator? Explain its functions briefly.
q4. Explain selector, selector invocation, and receiving object with a simple example.
q5. What are access specifiers? Explain public, private, and protected.
q6. Explain non-access modifiers: transient, synchronized, and volatile.
q7. Write the difference between `super()` and `super.method()`.
q8. Explain the use of `this` keyword inside a constructor.
q9. What is garbage collection? Why is it needed in Java?
q10. Differentiate between reference counting and tracing garbage collection.
q11. How can an object become unreferenced? Give all methods.
q12. What is the purpose of `finalize()`? When is it executed?
q13. What does `System.gc()` do? Does it guarantee garbage collection? *NOTHING EVER GUARANTEES GARBAGE COLLECTION*
q14. Explain the final keyword for variables, methods, and classes.
q15. What is a static method? How do you call it without creating an object?
q16. What is a constructor? Give an example.
q17. What is constructor overloading? Why is it useful?
q18. What is a copy constructor? Explain with an example.
q19. What is a destructor in Java? Why do we say Java doesn’t need destructors?
q20. Define method overloading. Why can’t it be done by changing only return type?
q21. Define method overriding. What rules must be followed?
q22. Explain variable shadowing with an example.
q23. What is recursion? Define base case and recursive call.
q24. Trace the working of a recursive factorial function for n = 3.
q25. Explain object construction and method calling using an example: `Student s1 = new Student(); s1.marks; s1.display();`.

Module 3

q1. What is the main advantage of using arrays when storing related data?
q2. Why is accessing elements beyond array bounds dangerous in Java?
q3. How does memory layout differ between a normal 2D array and a jagged array?
q4. Why are jagged arrays preferred when each row has a different number of elements?
q5. Explain how an array of objects works. Why is memory allocation two-step in this case?
q6. What limitation of arrays does ArrayList overcome?
q7. Why can ArrayList store only objects and not primitive types?
q8. What happens internally when an ArrayList grows in size?
q9. Why do iterators throw ConcurrentModificationException? Explain fail-fast behavior.
q10. Compare Iterator and Enumeration in terms of functionality and safety.
q11. When is a Vector preferred over an ArrayList? Give two real reasons.
q12. What is the difference between Vector's size and capacity? Why does Vector maintain both?
q13. Why is String immutable in Java? Give two benefits.
q14. Why is StringBuffer considered mutable? How does this affect performance?
q15. Explain the purpose of String's compareTo() method and how lexicographic comparison works.
q16. Why is == not used to compare strings? What problem does it cause?
q17. What is the practical use of the split() function? Give one realistic scenario.
q18. What is the benefit of overriding toString() in a class?
q19. Why are arrays and strings both zero-indexed in Java?
q20. Give one situation each where:

- an array is better than ArrayList
- an ArrayList is better than an array

Module 4

q1.Explain how inheritance improves code reusability.
q2.Why do fields not override while methods do?
q3.What happens when parent and child have a variable with same name? Explain with super.
q4.In multilevel inheritance, why can a child always be categorized under its parent?
q5.Why does Java not support multiple inheritance using classes?
q6.Why do parent-class variables appear when printed through child objects in hierarchical inheritance?
q7.How do access modifiers affect inheritance behaviour?
q8.How can private members still be accessed indirectly?
q9.Why is the parent constructor always executed before the child constructor?
q10.What must a child class do if the parent has no default constructor?
q11.Difference between super.method() and calling child method directly.
q12.Why can’t this() and super() be used in the same constructor?
q13.What is method overriding and why is it runtime polymorphism?
q14.Explain dynamic method dispatch with an example.
q15.Why are variables not polymorphic but methods are?
q16.Why must overridden methods have the same signature?
q17.What is the effect of marking a parent method as final?
q18.Why is a final class not inheritable? Give one real use.
q19.Why can abstract classes not be instantiated?
q20.Can abstract classes contain non-abstract methods? Why?
q21.What happens if a subclass does not implement all abstract methods?
q22.Why can abstract constructors not exist in Java?
q23.What problem do interfaces solve that abstract classes cannot?
q24.How do interfaces achieve a safe form of multiple inheritance?
q25.Why is interface-based programming more flexible?
q26.When a class implements two interfaces, how does runtime select method implementation?
q27.Explain the “what to do vs how to do” principle of interfaces.
q28.How does runtime polymorphism increase flexibility in programs?
q29.Why is method overloading compile-time polymorphism?
q30.For `Cat a = new Tiger();` explain which variable and which method run, and why.

Module 5

q1.Explain why exceptions are handled.
q2.Define checked and unchecked exceptions.
q3.When must a checked exception be declared?
q4.What does throw do?
q5.What does throws in a method signature mean?
q6.What is Throwable class?
q7.Why extend Exception for custom checked exceptions?
q8.Why extend RuntimeException for custom unchecked exceptions?
q9.What is finally used for?
q10.How does nested try-catch work?
q11.What is the difference between Error and Exception?
q12.Give two benefits of user-defined exceptions.
q13.What causes ArrayIndexOutOfBoundsException?
q14.Explain the purpose of System.gc() and finalize().
q15.Define a package and its advantages.
q16.How do packages avoid name collisions?
q17.What is the difference: import pkg.Class vs import pkg.*?
q18.Why use static import for methods?
q19.What is a Java thread?
q20.Differentiate concurrency and parallelism.
q21.Name two ways to create threads.
q22.What does start() do versus run()?
q23.What causes a thread to become blocked?
q24.What is thread synchronization used for?
q25.Define race condition and solution.
q26.What is deadlock? Give one prevention tip.
q27.Explain daemon versus user thread.
q28.What is thread priority and scheduling?
q29.Why prefer Runnable over extending Thread?
q30.Explain wait(), notify(), notifyAll().