M T W T F S S

Page No.: YOUVA

Date:

OOPS for preparation

Paradisms & Basic constructs

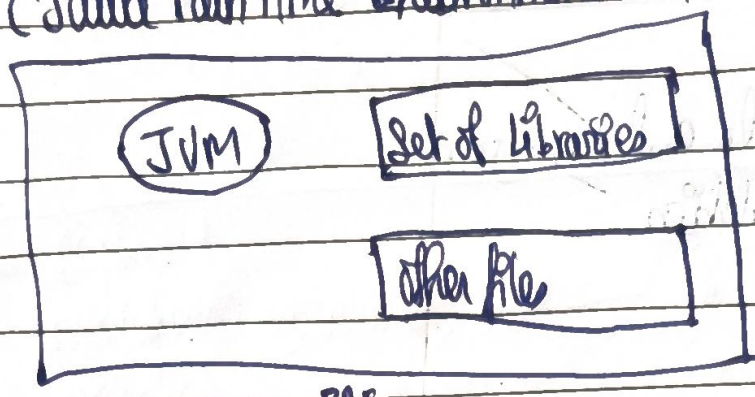
*) OOPS concepts
*) Objects, class, methods and Messages
*) abstraction and encapsulation
*) Inheritance
*) Abstract classes
*) Polymorphism
*) Objects and classes in Java
*) Defining classes, Methods
*) Access specifiers
*) Static members
*) Constructors
*) Finalize method

OOPS concept

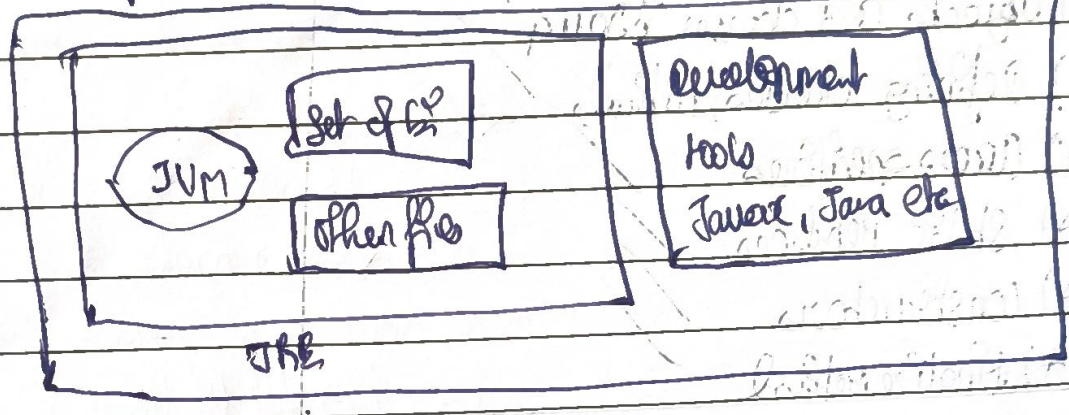
o) Classes: A collection of objects are a class, objects are like instances of the class
o) Object: Its a collection of code used to perform particular tasks
o) Inheritance: Its a way of expanding the classes with parent and child connection
o) polymorphism: Its a way a sharing same message in different forms
o) Method overloading: Method overloading is a feature that allows a class to have two or more methods having same name but ~~created~~ executes differently based on the arguments passed
o) Method overriding: Subclass provide the specific implementation of the method that has been provided by one of its parent class
o) Abstraction: The process of hiding the implementation
o) Encapsulation: Process of wrapping the code

Overview of Java

*) It is secured since it has its own Runtime environment
*) JRE (Java RunTime Environment) it is the implementation of JVM



*) JDK (Java development kit), It is the software development environment



*) JVM (Java virtual machine)
They are those that provides runtime environment,
- Load code
- Verify code
- Check error
- Check location

*) Java has two main components
1) Runtime environment
2) Application programming interface

Difference between procedure oriented programming and object oriented programming

*) POP has functions, oops has methods
*) Top to down approach in POP, In OOPS it is bottom to top approach
*) pop dose not have access specifiers, OOPs has access specifiers
*) POP is not easy, oops are easy
*) POP is old, oops are advanced

Basic programs

Class & Objects

code:

```
class Dog {                    → Class
    String name;
    int age;
    void bark() {
        System.out.println(name + " is barking!!");
    }
}
public class DogDemo {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        MyDog.name = "Jully";
        MyDog.age = 3;
        MyDog.bark();
    }
}
```

output:

Jully is barking

Abstraction and Encapsulation

Abstraction: not caring about implementation
Encapsulation: hiding certain data for security

Code:

```
class person {
    private String name;        ⟶ setting private for encapsulation
    private int age;

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

    public void setAge(int age) {
        this.age = age;
    }

    public void getAge(int age) {
        return age
    }

// write basic main function code {
    person = new person();
    person.setName("Jeeva");
    person.setAge(18);
    System.out.println(person.setName + person.getAge);
```

O/P

Jeeva 18

M T W T F S S

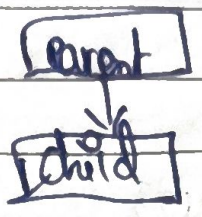
Page No.: YOUVA

Date:

Inheritance

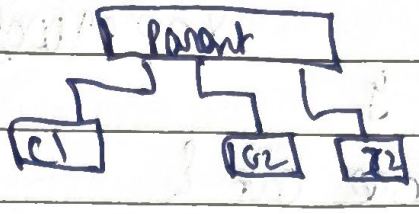
Types

1) Single



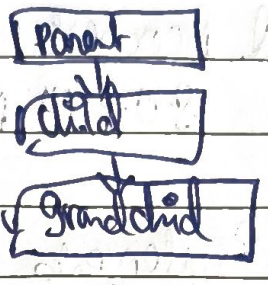
100% is more one to one

2) Hierarchical



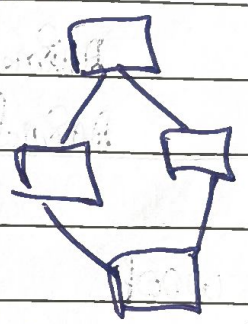
one parent Many child one-many

3) Multilevel



one parent one child, a child has one child one to one to one...

4) Hybrid : Mixture of two inheritance
Ex: Multi and Hierarchical



4) Multiple Inheritance : one child can have more than one parent

Code:

```
class Animal {
    void eat() {
        System.out.println("This animal eats food");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks");
    }
}

// Main class
    Dog mydog = new Dog();
    mydog.eat();
    mydog.bark();
```

Multiple Inheritance Code

S.O.P (System.out.println);

Code:

```
interface dog{
    void bark(){
        S.O.P("woff woff");
    }
interface cat {
    void meow(){
        S.O.P("meowwww");
    }
}
public class Animal implements dog, cat{
    public static void main(String[]args){
    Animal Animal1 = new Animal();
    Animal1.dog();
    Animal1.cat();
```

output:

woof woof

meowwww

like wise write code based on the type of Inheritance

Abstract classes

#1 Abstraction, only has interface and implementation is seperate
Just like switches in fan, moon off is the only things displayed
others are implementation

Code:

```
abstract class Animal {                    → Mentioning abstract instead of private
    abstract void Sound();                 → Interface, not implementation
    void sleep() {                         → normal method
        System.out.println("Sleeping");
    }
}

class dog extends Animal {
    void Sound() {
        S.O.P("woof woof");
    }
}

// Main class code blah blah blah

Dog MyDog = new Dog();
MyDog.Sound();
MyDog.sleep();
```

output

Sleeping
woof woof

This code can be used for interface and implementation too but don't use the word abstract just create class and object and extend to implement

- for that don't use extend use implementation

```
class Dog implements Animal
```

Constructors: → Methods

constructors initates objects in class

There are two types:

i) default (no arguments)

ii) parameterized constructor (has parameters)

Code:

```
class Box {
    double width;
    double height;
    double depth;
    Box() {                    → default Constructor
        SO.P ("constructing box");
        width = 10;
        height = 10;
        depth = 10;
    }
    Box (double w, double h, double d) {   → type 2
        width = w;
        depth = d;
        height = h;
    }
    double volume() {
        return width * height * depth;
    }
// write main class codes
    Box mybox1 = new box (10, 20, 25);   ] method overloading
    Box mybox2 = new box
```

Method overloading

Access Modifiers

1) Private, if variable is declared private, then it can only be accessed within the method or submethods

```
class test{
    private int x=0;
    void show(){
        sop(x);
    }
}
```

2) Default:

*) It is accessed only within the package (one Java program file) It allows only for the packages and subpackages

```
int x=10;
```

3) Protected:

*) Same as default

```
protected int x=10;
```

rarely used!

4) Public: Can be used anywhere, even from another packages

```
public int y=200;
```

Static methods (members, variable, blocks)

*) Static members are variables or methods of a class that belongs to the class itself, not the objects

*) They are accessed without creating objects

Code:

```
class demo{
    static int x=10;
    static void display(){
        SOP(x);
    }
}
// Main class code
```

Output:

10

Finalize():

*) In Java finalize() is a method that belongs to the object class and it is called by garbage collector (GC) before the object is destroyed.

Syntax: protected void finalize() throws Throwable

Code:

```
Public class car{
    public car(){
        System.O.P("car is created");
    }
    protected void finalize() throws Throwable{
        try{
            S.O.P("clean up.");
        }finally{
            super.finalize();
        }
    }
}
```

```
public static void main(String[] args){
        Car myCar = new Car();
        myCar = null
        System.gc();          -> Garbage collector
    try{
        Thread.sleep(100);
    } catch (InterruptedException e){
        Thread.currentThread().interrupt();
    }
    System.out.println("End of main method");
}
```

Output:

Car is created
Requesting Garbage collection
Car is been destroyed
End of main method

Polymorphism

*) Its a way of giving informations in different forms
*) There are two types of polymorphism
    1) Compile time polymorphism (method overloading)
        Many methods have same name but different parameters according to the users parameters the method is choosen in compile time
    2) Run time (method overriding)
        *) Based on the users input from runtime the method is choosen called Runtime polymorphism

Compile Time Polymorphism

Code:

```
class Calculator {
    void area(double size){          -> square
        System.out.println(size*size);
    void area(double length, double breadth){ -> rectangle
        System.out.println(length*breadth);
    void area(double radius){        -> circle
        System.out.println(3.14*radius*radius);
    }
}
public class Main {
    public static void main(String[] args){
        Calculator calc = new Calculator();
        calc.area(5.8);
        calc.area(6,7);
        calc.area(7)
```

Output

11.6 -> Square
42 -> Rectangle
153.9 -> Circle

Runtime Polymorphism

Code:

```
class Animal {
    void sound(){
        S.O.println("Animal makes sound");
    }
}
```

```
class Dog Extends Animal {
        void Sound() {
            S.O.P ("Dog Barks");
    }
}
class Cat extends Animal {
        void Sound() {
            S.O.P ("Cat meows");
    }
}
Public class Main {
        Public Static void Main (String [] args) {
            Animal d = new dog()
            Animal c = new Cat()
            d.Sound();
            c.Sound();
```

output

Dog barks
Cat meows