15/4/25

OOPS in JAVA
(Lec-27)

C → functional Programming language

C++ → (C language with classes, that named object oriented.)
(C++, Object oriented language)

JAVA → "completely Pure object oriented language"
(It doesnot support only functional Programming)

#Diff b/w JAVA and C++

- JAVA: Completely Pure object oriented
- C++: classes & objects are a part of language / feature of C++

Classes & object

- It is a blueprint
- (An entity that follows that blueprint)

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

    public addSubject()
    {
    }
}
```

attributes / properties

methods / functions

⇒ [classes has properties (variables) & functions as blueprint]

→ Student s1 = new Student()

object

This object has by default (name & rollno)

Eg. ArrayList<Integer> al = new ArrayList<>();

object

stores in heap memory.

```
class Student{
    String name;
    int roll_no;
    public int sum(int a, int b)
    {
        return a+b;
    }
    public String toString(){
        return "name of student"+name+"RollNo"+roll_no;
    }

    p s void main(String[]args)
    {
        Student s1 = new Student();
        sysO(s1.name);
        sysO(s1.roll_no);
    }
```

special function that calls automatically when we print object. (diff from constructor)

null

0

→ if we haven't set any value to object s1, it will print its default value.

```
Student s2 = new Student();
s2.name = "Ram"
s2.roll_no = 34
sysO(s2.name);
sysO(s2.roll_no);
```

Ram

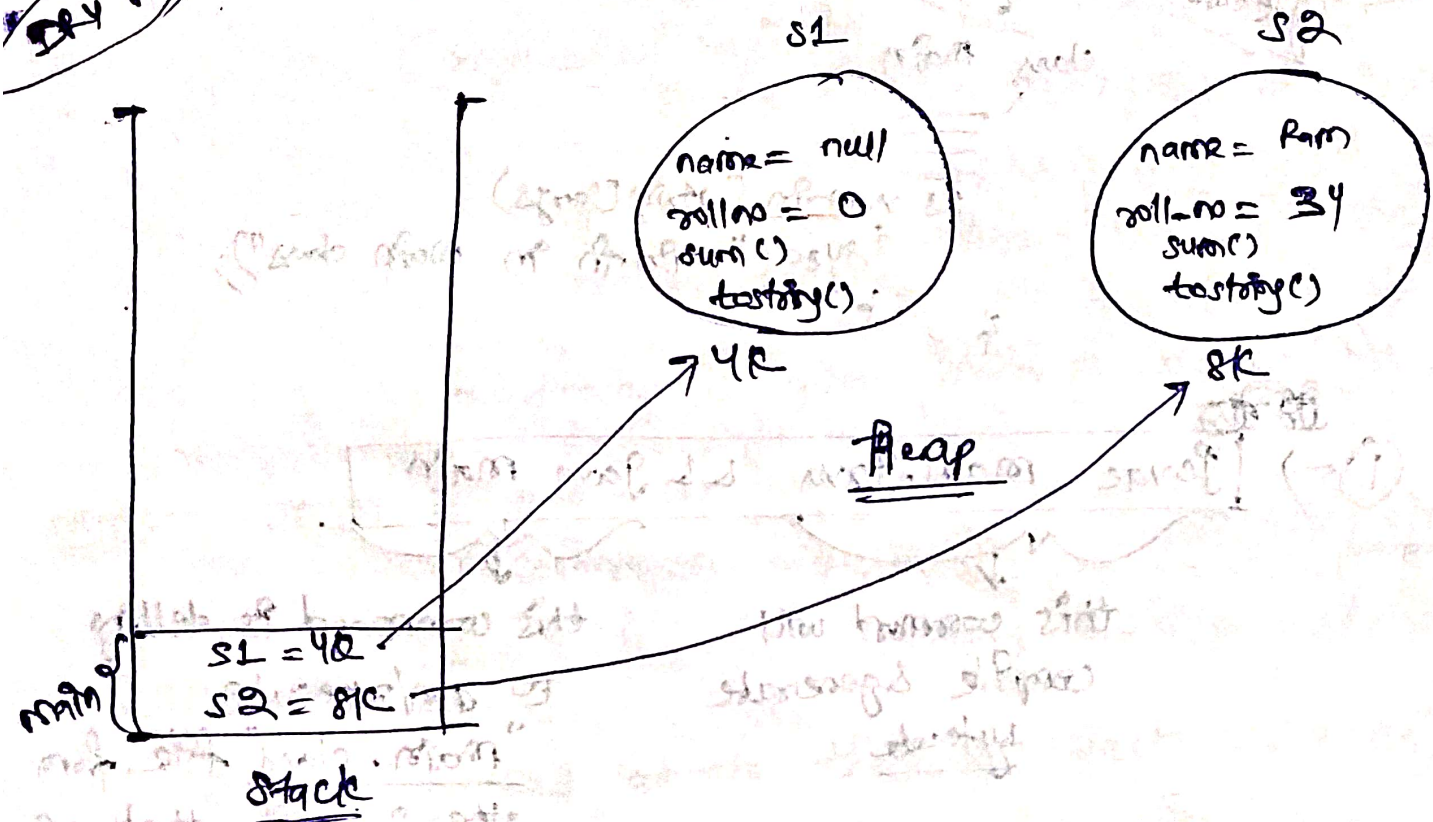
34

sysо(s1); → Name of student null Roll no 0.
sysо(s2); → Name of student Ram Roll no 34.

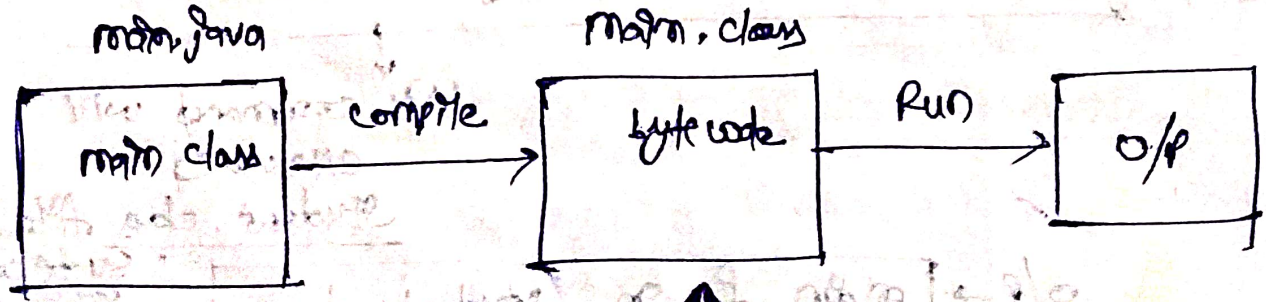
(If toString() method was not written we will get address of objects)

}

Dry Run



Note:- Objects are alloted space exactly like arrays in Heap memory.



(How a program is executed in JAVA)

Main.java ✗

```
class Student
{
    ps v main(String[] args)
    {
        sysо("main fn in student class")
    }
}
class main
{
    ps v main(String[] args)
    {
        sysо("main fn in main class");
    }
}
```

main.class
Student.class
bytecode

① → javac main.java && java main

this command will compile & generate bytecode

this command is telling to run/execute "main.class" file from the 2 files that are generated.

o/p → main fn in main class

② → javac main.java && java Student

this command will executed Student.class file (bytecode).

o/p → main fn in student class

Errors

Compile time Error (which occurs at compile time only)

int a = "Pushan";

Assigning a string value to integer (Also called Logical Error)

Runtime Error (which occurs at run time)

int [] arr = new int [5];

arr[56] = 4;

Assigning at a index which is way beyond the assigned space to the Array.

& it will happen only at runtime because Array is given memory at runtime.

Note:- JAVA is platform independent.

(can run on any machine ie MAC/win/Linux).

CONSTRUCTORS

(function invoked at the time of object creation)

- no return type
- Name same as class Name.

Types

Default (No Arguments)

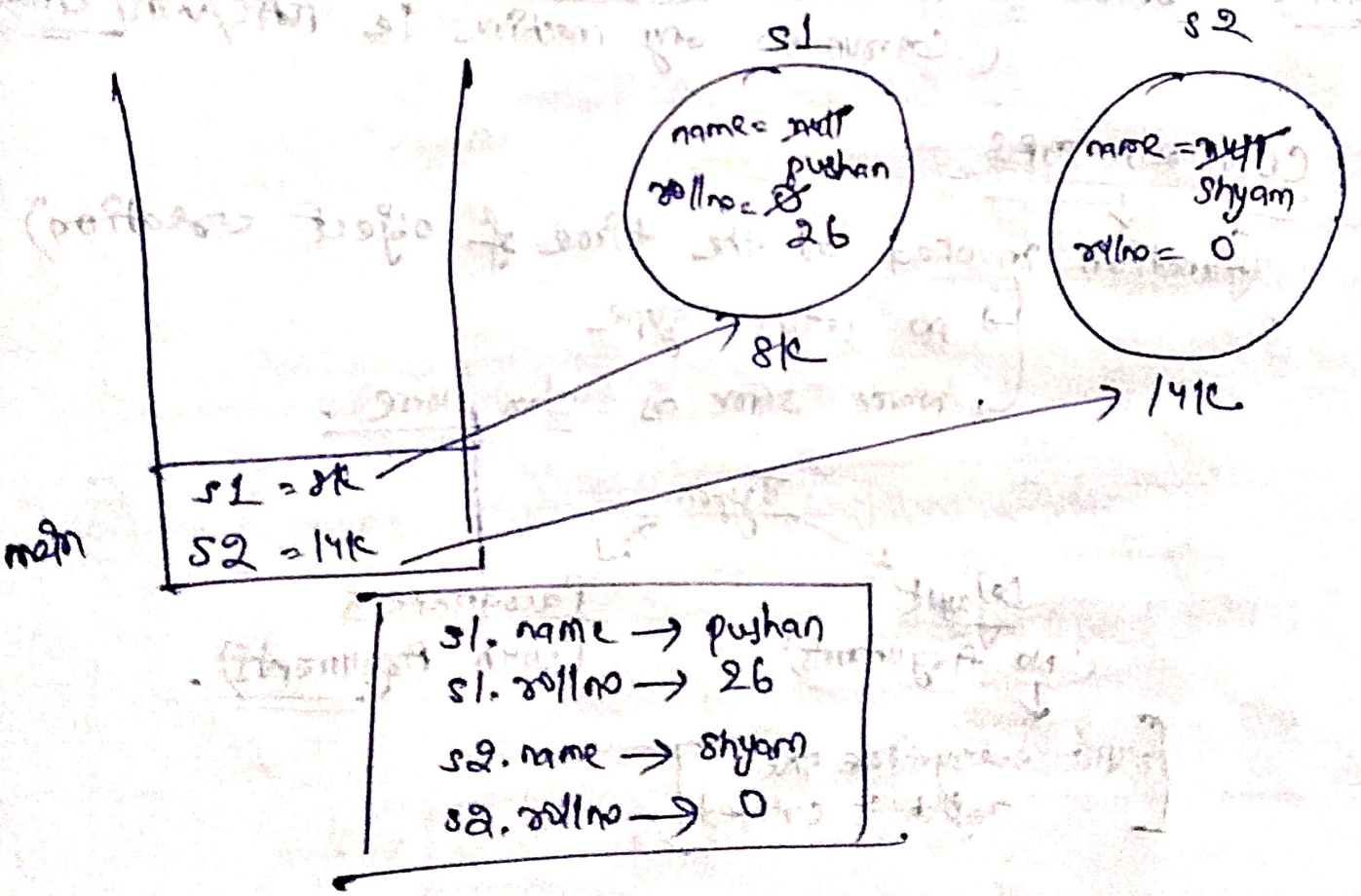
[runs everytime the object is created]

Parameters (with Arguments).

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

    public Student()
    {
        name = "pushan";
        rollno = 26;
    }

    public Student(String currentName)
    {
        name = currentName;
    }

    public static void main(String[] args)
    {
        Student s1 = new Student();
        Student s2 = new Student("shyam");
    }
}
```

global variables (scope is inside whole class)

local variable (only available inside brackets).



this keyword :→ (was introduced to avoid confusion the global variables & local variables).

```
class Student
{
    String name = "default"
    int roll_no = -1;

    public Student(String name)
    {
        sysо( this.name + "," + name);
        this.name = name;
    }

    p s v main (String[] args)
    {
        Student s1 = new Student("pushan");
    }
```

refers to function argument

refers to class global attribute

O/P →

```
default , pushan
```

Constructor Chaining :→

⇒ (Calling another constructor from one constructor.)

Rules →
- 1st line should be constructor call.
- There should be atleast one constructor with no further calls.
- Order won't matter.

Why it is necessary

↓

Ans to follow "DRY"

(Don't repeat yourself).

eg:

```
class Vehicle
{
    String color;
    int mileage;
    int seats;

    public Vehicle (String color)
    {
        this.color = color;
        sysO("Const with color param");
    }
    public Vehicle (String color, int mileage)
    {
        this(color);
        this.mileage = mileage;
        sysO("Const with color, mileage param");
    }
    public Vehicle (String color, int mileage, int seats)
    {
        this(color, mileage);
        this.seats = seats;
        sysO("Const with color, mileage, seats of param");
    }
```

Calling constructor already made which has color as param

Constructor Chaining

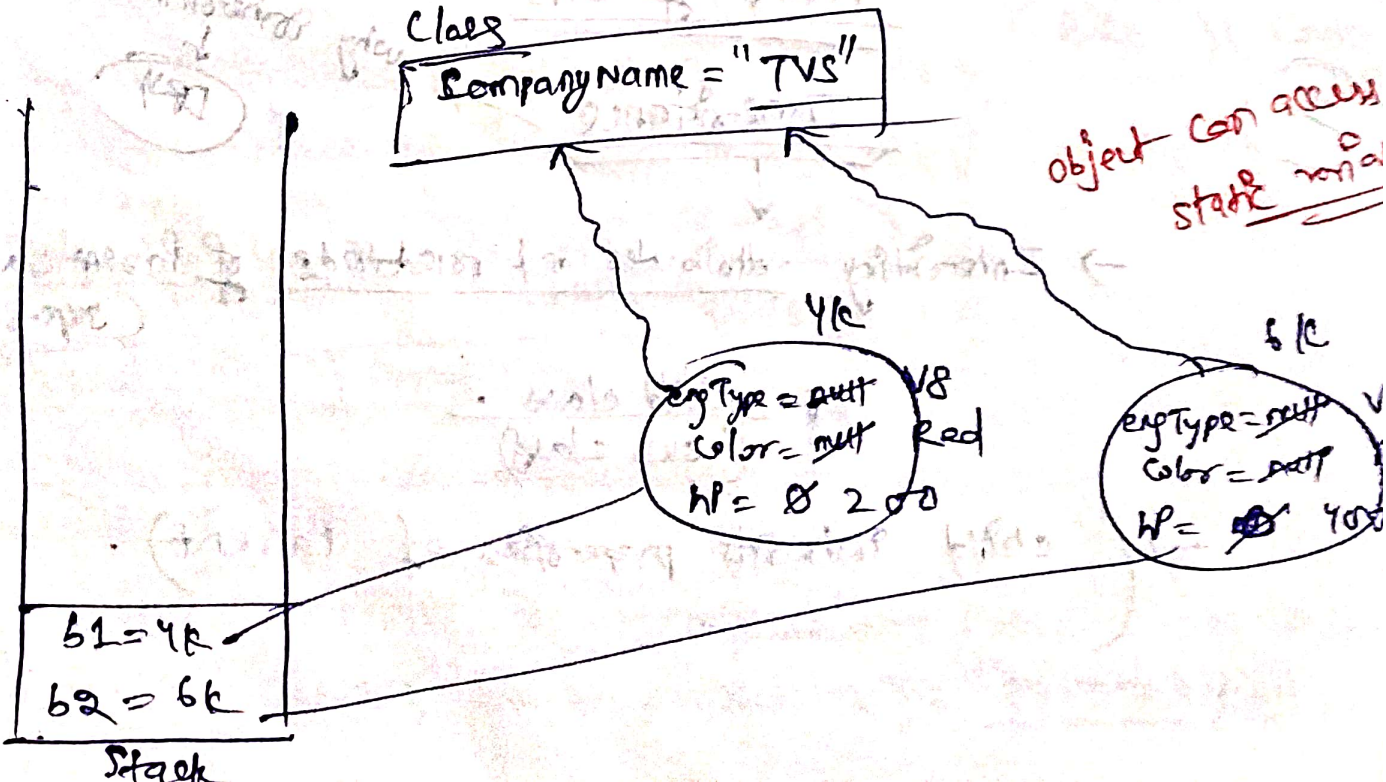
Q: Make a class "Bike"

- attributes → engineType, color, horsePower
- methods → colorChange(), double horsePower()

static keyword :→

↳ (static is the property of class & not of the objects)

eg:
```
class Bike
{
    String engineType;
    String color;
    int horsePower;
    static String companyName = "TVS";

    //constructors
    Bike()
    {
    }
}
main()
{
    Bike b1 = new Bike("V8", "Red", 200)
    Bike b2 = new Bike("V4", "Black", 400)
    syso(b1.companyName); → TVS
    syso(b2.companyName); → TVS

    b1.companyName = "SUZUKI";

    syso(b1.companyName); → SUZUKI
    syso(b2.companyName); → SUZUKI
```



→ Both static attributes & methods exists

Q. Why is static written before main fn?

A. bcz static functions can be called without creating objects. compilers directly run these

Main.main()
Questions.main()
Palindrome.main() etc.

Rules :→ 1) You can call static methods directly without creating objects.

2) Static functions cannot change non static values and static functions cannot call Non static functions.

3) "Non static" function can call "static" function and "Non static" method can change "static" values.

19/4/25

Lec-28

Pillar of OOPS

I

Inheritance → why inheritance? DRY

→ Inheriting attributes and methods of Parent class (super class) to child class. (sub class)

→ (child inherits properties of Parent).

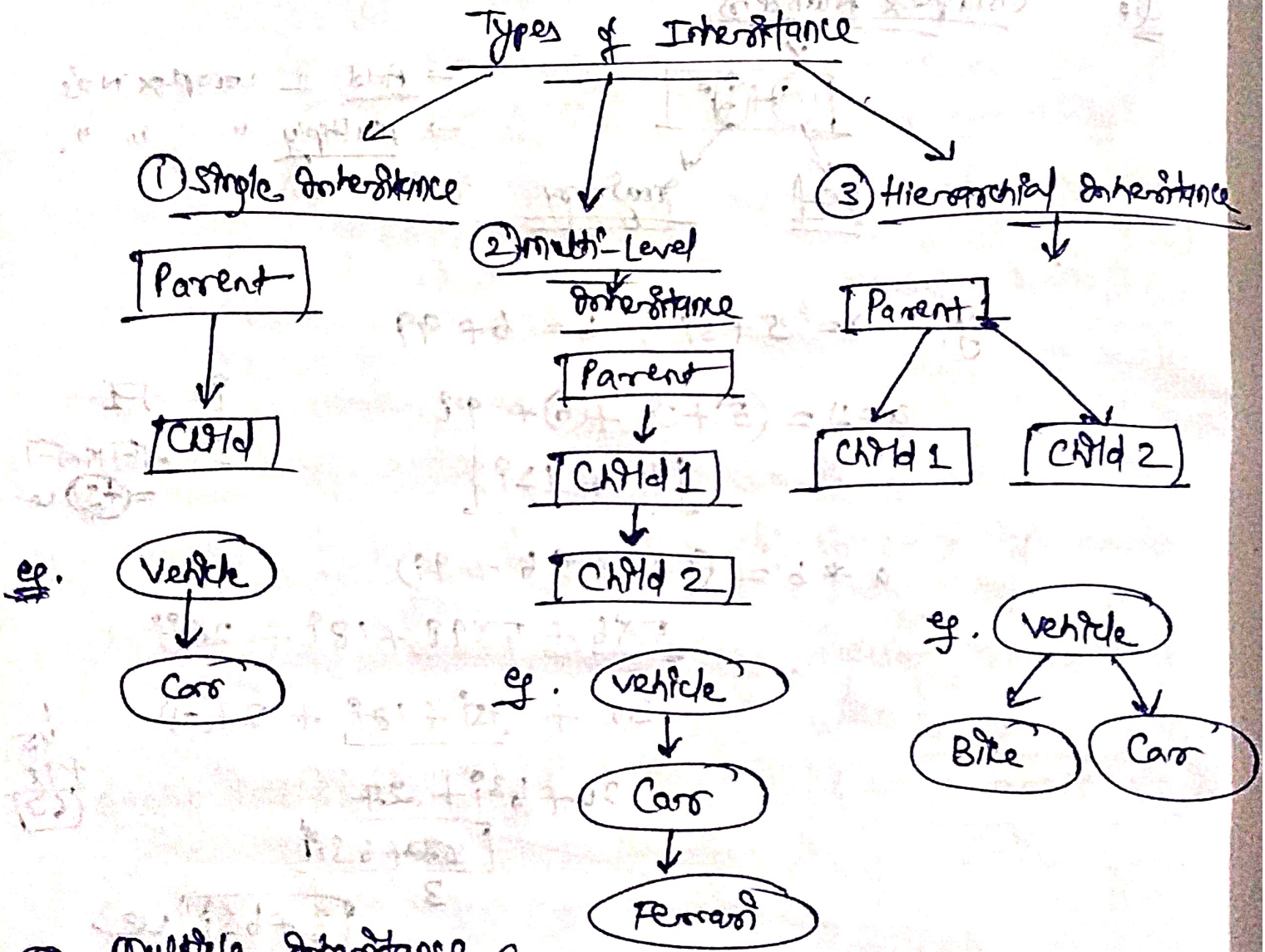
Note:- Constructor of Parent class is called before constructor of child class.

'super' keyword

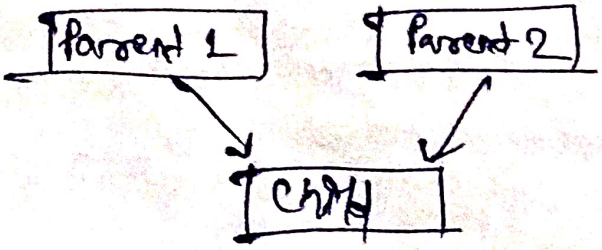
↓

super will call parameterized constructor of Parent class from child class (similar to how this calls within a particular class).

Types of Inheritance



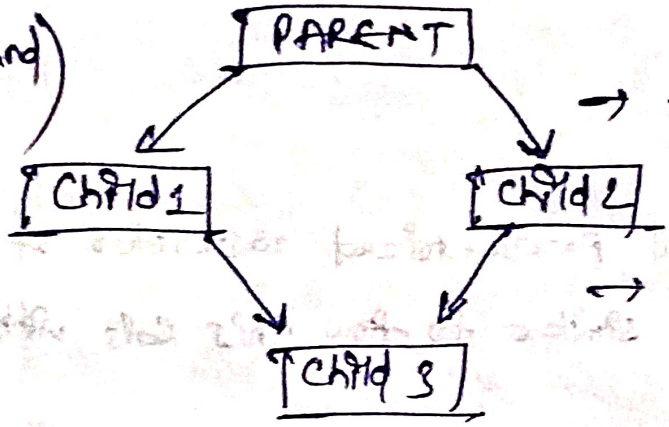
4) Multiple Inheritance (NOT POSSIBLE)



(NOTE: MULTIPLE INHERITANCE IS NOT POSSIBLE IN JAVA "WITH CLASSES", can be possible using "INTERFACES").

⑤ Hybrid Inheritance (NOT POSSIBLE)

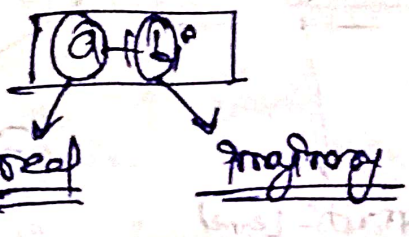
Combination of (Hierarchical and multiple)



→ Hierarchical

→ multiple

Q. Complex numbers.



→ Add 2 complex no's

→ multiply " " ",

eg. $a = 5+3i$, $b = 6+9i$

$(a+b) = (5)+3i+(6)+9i$

$11 + 12i$

$i = \sqrt{-1}$

$i^2 = \sqrt{-1} \times \sqrt{-1} = -1$

$a * b = (5+3i)(6+9i)$

$5\times6 + 5\times9i + 18i + 27i^2$

$30 + 45i + 18i + 27(-1)$

$30 + 63i + 27$

$3 + 63i$

45
18
63

=

II

Polymorphism

Poly (many) — morph (forms)

Types

Compile Time Polymorphism

can be achieved using "method overloading"

also called as [Early binding / Static binding]

Run time Polymorphism

can be achieved using "method overriding"

also called as [Late binding / Dynamic binding]

III

Encapsulation

Capsule

Capsulating data

hiding some information showing only necessary information which is done through

ACESS MODIFIERS

Access modifiers

public
(can be accessed anywhere)

private
(can be accessed within class)
value cannot be set & cannot get but possible only with help of [getters & setters]

protected
(can be accessed within same package).

default
(can be accessed within class, child class, same package.)

Q. Student

```
{
    int roll_no;
    private int marks;
}
```

getters, setters

IV

Abstraction → abstract keyword

(Hiding of implementation)

→ writing declaration of fn in "Parent class"

but defining in "child class".

Abstract classes

```
abstract class Shape
{
    int numberofsides;
    abstract public double getArea();
}
class Rectangle extends Shape
{
    int length;
    int width;

    public Rectangle(int length, int width)
    {
        super.numberofsides = 4;
        this.length = length;
        this.width = width;
    }
    public double getArea()
    {
        return this.length * this.width;
    }
}
```

Constructors can be made in Abstract class

not declare in parent class rather define in child class

NO objects can be made out of Abstract class

Note:- In Abstract, there can be abstract classes and non-abstract classes aswell.

Interface → (it implements) not extends.

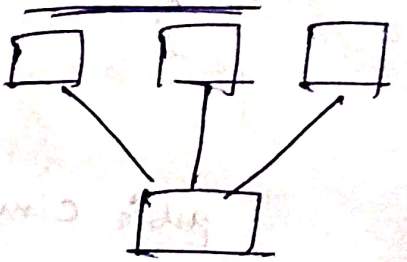
→ (100% abstraction)

↳ (All the methods are abstract).

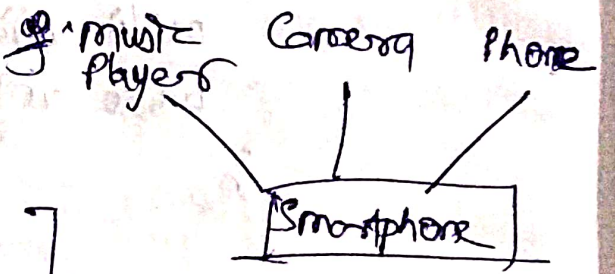
Note: Abstract classes are different than Interface

→ main work of Interface is

- Abstraction
- multiple inheritance



Interface doesn't constructors



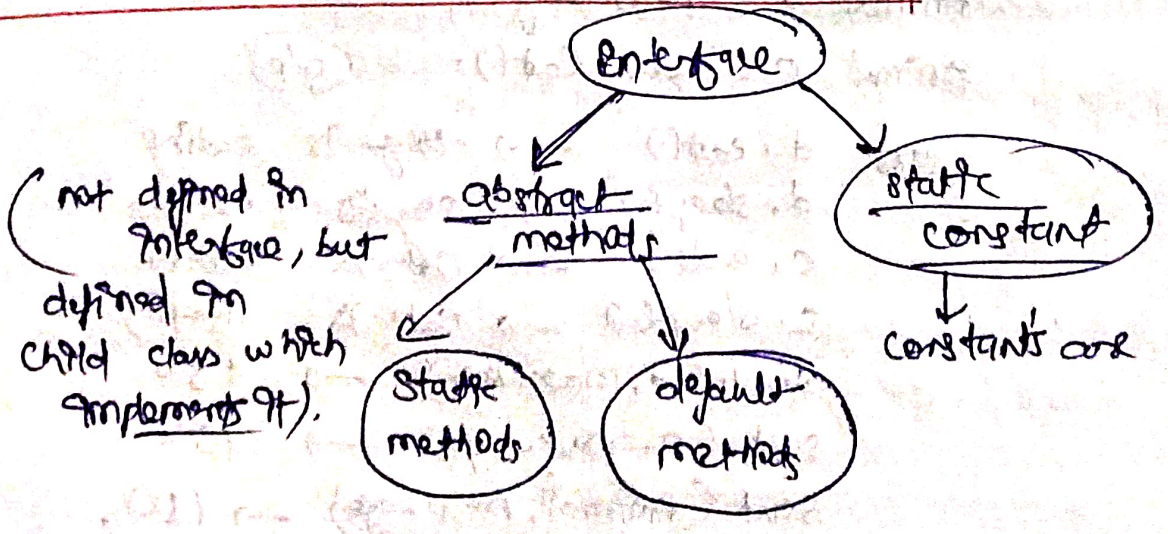
class → blueprint to make objects

Interface → blueprint to make classes

No objects can be made of interface.

Interface

- abstract methods
  - (not defined in Interface, but defined in child class which implements it).
  - Static methods
  - default methods
- static constant
  - constants are public static final } by default



Interface Animal { [Animal.java]

```
public static final int max-age=150;
public abstract void eat();
public abstract void sleep();
}
```

[Dog.java]

```
public class Dog implements Animal {
    public void eat() { sout("Dog is eating"); }
    public void sleep() { sout("Cat is ~~eating~~ sleeping"); }
}
```

[Cat.java]

```
public class Cat implements Animal {
    public void eat() { sout("Cat is eating"); }
    public void sleep() { sout("Cat is sleeping"); }
}
```

[Test.java]

```
public class Test {
    psvm(String... args){
        Animal d = new Dog();
        Animal c = new Cat();
```

O/P

```
        d.eat()  → Dog is eating
        d.sleep() → Dog is sleeping
        c.eat()  → Cat is eating
        c.sleep() → Cat is sleeping
        sout(d.max-age) → 150
        sout(c.max-age) → 150
        sout(Animal.max-age) → 150.
    }
```

Here we can write whole definition unlike Abstraction where we only declare the method

In JAVA 8

Inside Interface we can make

static methods (Can be accessed by Parent class)

- Animal.info() ✓
- dog.info() ✗
- cat.info() ✗

default methods (Can be accessed by its child class)

- d.run() ✓
- c.run() ✓
- Animal.run() ✗

Inside Animal Interface,

static method :-

```
public static void info(){
    sout("Animal information");
    }
```

default method :-

```
public default void run(String typeofAnimal)
    {
    sout(typeofAnimal + " is running");
    }
```

Test.java

Animal.info() → Animal information

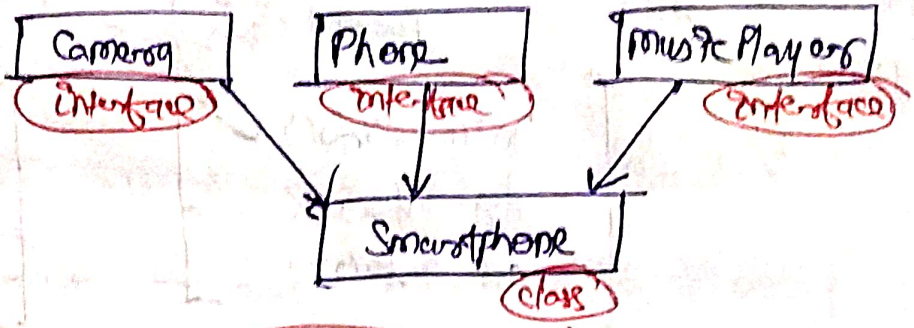
d.run("dog") → Dog is running

c.run("Cat") → Cat is running.

Multiple Inheritance with help of Interface

✗ class smartphone extends Camera, Phone, Music Players

↓ "Cannot happen"



Interface smartphone implements Camera, Phone, Music Players

↓ "CAN HAPPEN"

Lec - 29

Java Wrapper Class

↳ Wrapper classes provides us a way to use our primitive datatype as objects.

Primitive Data type	Wrapper Class
int	Integer
float	Float
double	Double
boolean	Boolean
char	Character

Wrapper classes provides alot of inbuilt pre built functions (utility methods).

int → (default value = 0)

Integer → (default value = null)

→ Wrapper classes are used in collection API.

try Catch (Exception handling)

final
↳ (fixed → cannot be changed)
↳ Can be applied to → (variable, methods & classes).

Qa

Pizza
↳ base price → veg → 300
→ Non veg → 400

↳ addExtraToppings() → veg → 70
→ Non veg → 100

↳ addExtraCheese() → 80

↳ takeaway() → 20.

DeluxePizza
↳ extraToppings()
↳ Cheese()

Note: → Every request/addOns must be added only once.