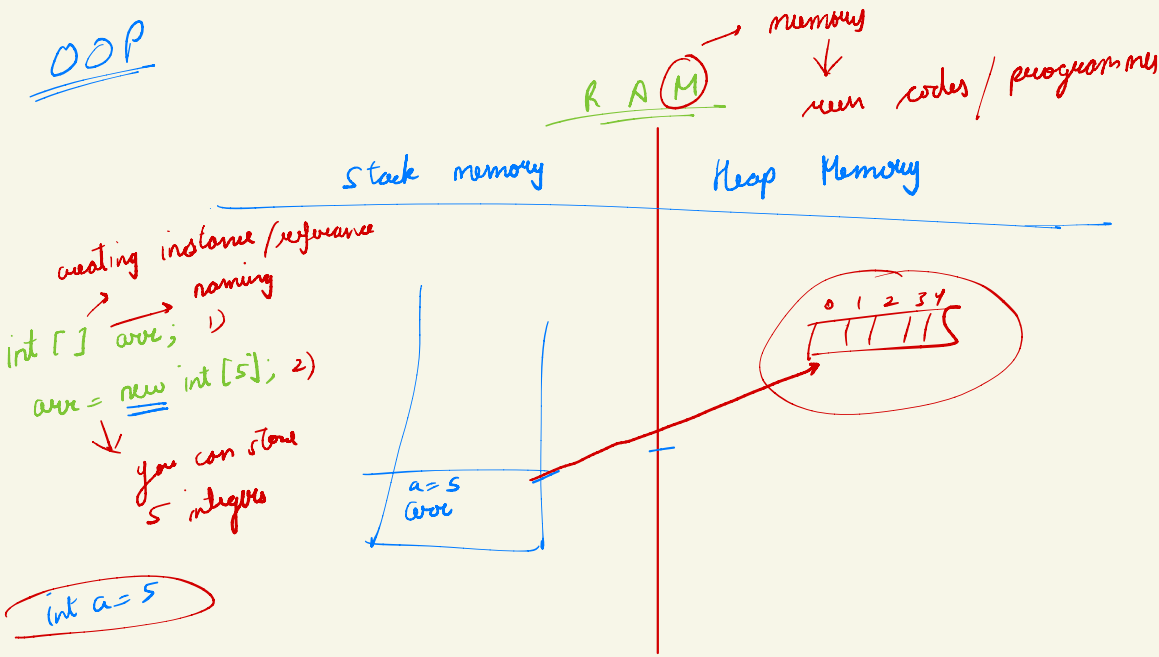
OOP




```
public void swap (int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int a = 2;
int b = 5;

swap (a, b);
```

→ a = 2; ① a = 5 ②

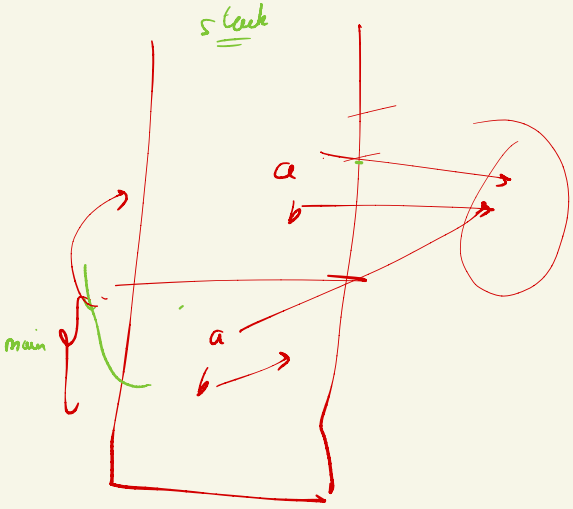
→ b = 5; b = 2

space is created in stack for a function

```java
class HelloWorld {
    public static void swap(int a, int b){
        int temp=a;
        a=b;
        b=temp;
    }

    public static void main(String[] args) {
        int a=2;
        int b=5;

        System.out.println("value of a before swapping "+a);
        System.out.println("value of b before swapping "+b);

        swap(a,b);

        System.out.println("value of a before swapping "+a);
        System.out.println("value of b before swapping "+b);
    }
}
```

a = 2
b = 5

a = 2
b = 5



Object Oriented Programming

class → blueprint

objects → copies

DRY

don't repeat yourself

Locks

Thorn ands

Blueprint

color? sunroof

Students name

roll class

group

Teachers

subject class

```
class Car {
    int model_number;
    String Color;
    int mileage;
}
```

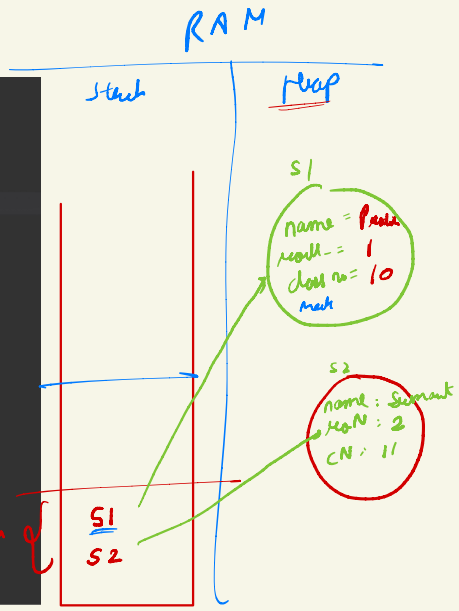
```
class Student {
    String name;
    String class;
    int roll_number;
}
```

1) Student samrat;
2) samrat = new student();

Class → attributes / properties

methods / functions

```java
public class Main
{
    // blueprint for student
    class Student {
        String name;
        int roll_number;
        int class_number;
    }

    public static void main(String[] args) {
        Student s1 = new Student();

        s1.name = "Pratik";
        s1.roll_number = 1;
        s1.class_number = 10;

        Student s2= new Student();

        s2.name="Sumant";
        s2.roll_number=2;
        s2.class_number=11;
    }
}
```



Access Specifiers

1) public → anyone can access, even outside the class

2) protected → only sub-classes can access.

3) private → you can access private members inside the class

4) default → can be accessed within the same package

Constructors

↓

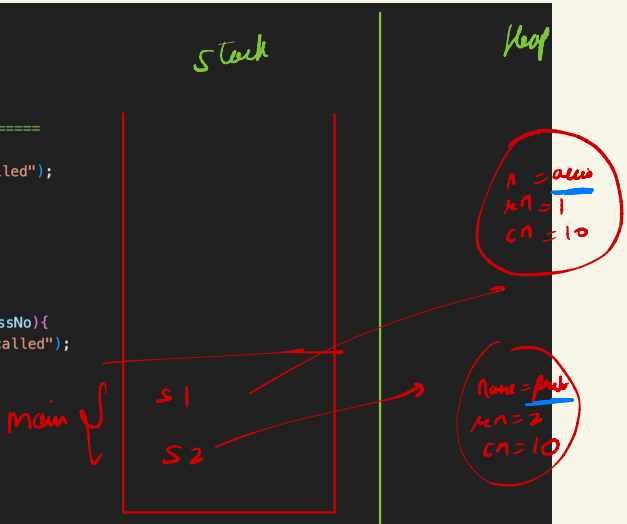
function inside your class with same name as your class and no return type

Types

1) default → constructor with no parameter

2) parametrized constructor
   ↳ constructors with parameters.

```java
class Student {
    String name;
    int roll_number;
    int class_number;

    // default constructor ========================
1)  public Student(){
        System.out.println(x: "Constructor is called");
        name="Accio";
        roll_number=1;
        class_number=10;
    }

    // parametrized constructor
2)  public Student(String a, int rollNo, int classNo){
        System.out.println(x: "constructor 2 is called");
        name=a;
        roll_number=rollNo;
        class_number=classNo;
    }
}


public class OOP {
    Run | Debug
    public static void main(String[] args) {
        Student s1 = new Student(); // it will call the construtcor automatically,  it will create heap memory
        Student s2= new Student(a: "pratik",rollNo: 2,classNo: 10);

        System.out.println(s1.name);
        System.out.println(s2.name);
    }
}
```



this
↳ it is used to access properties/attributes of current class.

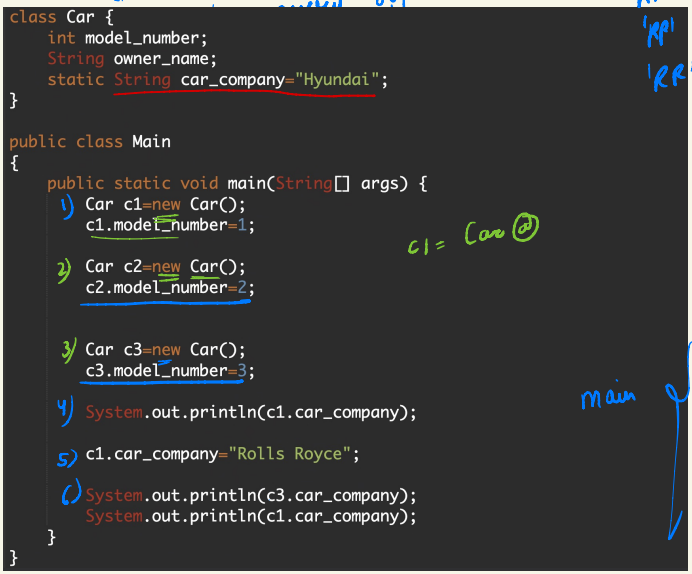
---

static keyword

public static void main

public static int fun

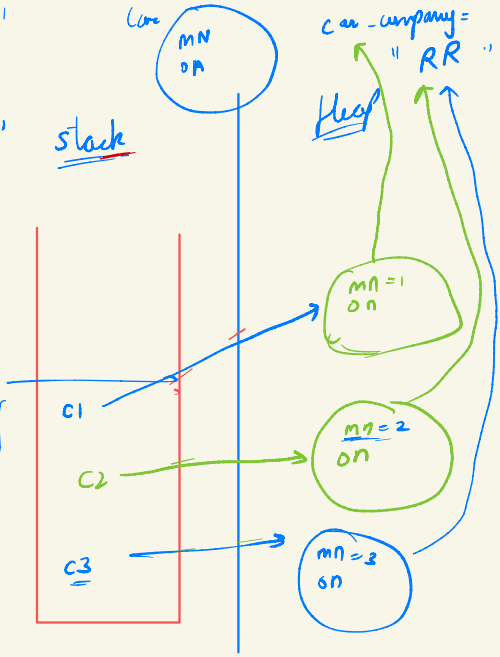
class Student {
String name;
int roll_no;
int module_no;
static String organisation = "accio";
}



```
class Car {
    int model_number;
    String owner_name;
    static String car_company="Hyundai";
}

public class Main
{
    public static void main(String[] args) {
     1) Car c1=new Car();
        c1.model_number=1;

     2) Car c2=new Car();
        c2.model_number=2;


     3) Car c3=new Car();
        c3.model_number=3;

     4) System.out.println(c1.car_company);

     5) c1.car_company="Rolls Royce";

     6) System.out.println(c3.car_company);
        System.out.println(c1.car_company);
    }
}
```

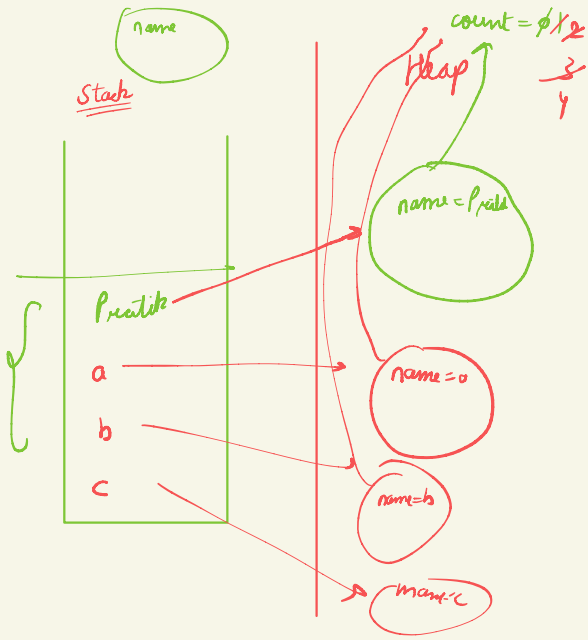
"Hyundai" 'RR' 'RR' 'RR'

c1 = Car@



Ques Create a student class. Create some student objects and keep a count to store number of student object.

```java
class Student {
    String name;
    static int count=0;

    public Student(String n){
        name=n;
        count=count+1;
    }
}

public class Main
{
    public static void main(String[] args) {
     1) Student Pratik=new Student("Pratik");
        System.out.println(Pratik.count);

     2) Student a=new Student("a");
        System.out.println(Pratik.count);

     3) Student b=new Student("b");
        System.out.println(Pratik.count);

     4) Student c=new Student("c");
        System.out.println(c.count);
    }
}
```



public static void main

1) static functions belong to classes
2) static functions can only change static variables.
3) We can call static functions without creating object.

Main.main();

class Main {

public static void main( ) {

}

}

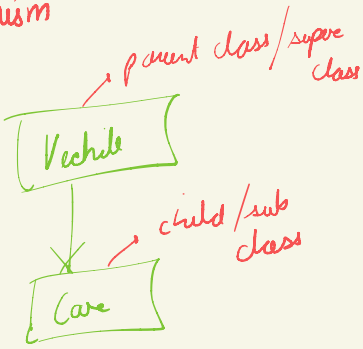
Pillars of OOPs

1) Inheritance
2) Encapsulation
3) Abstraction
4) Polymorphism

=> Inheritance

Reusability

```
class Vechile {
    int no_of_wheels;
    String color;
}

Class Car extends Vechile {
}
```



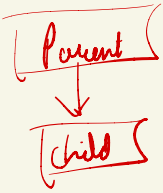
Bike

Truck

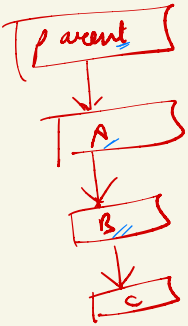
1) Parent class constructor is called first, after that child class constructor is called.

Types of Inheritance

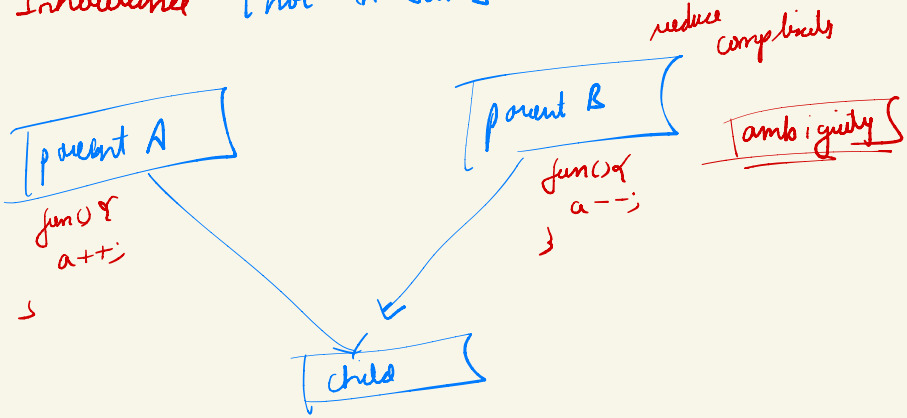
1) Single Inheritance



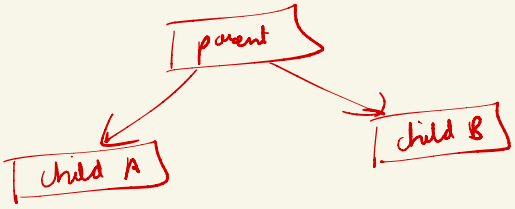
2) Multi level Inheritance



3) Multiple Inheritance [not in Java]

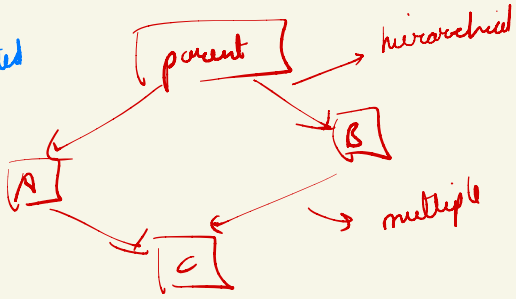


4) Hierarchical Inheritance



5) Hybrid Inheritance [sum of Hierarchical and multiple]

not supported in Java



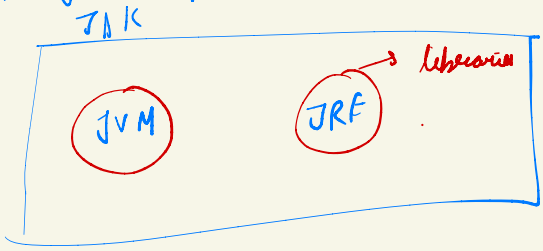
JDK → Java development Kit

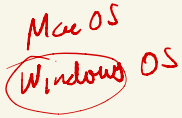
↳ libraries

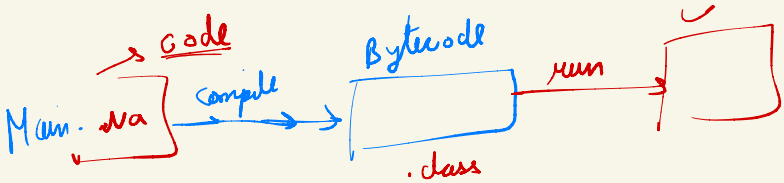
↓

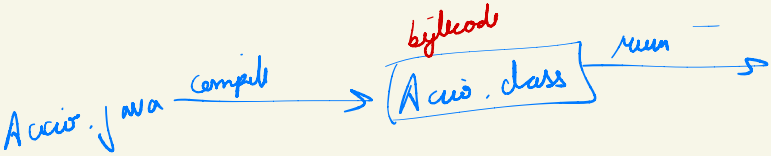
JVM → Java virtual Machine.

1) Java is platform independent









---

this

↳ this 'refers' to our instance of class.

```java
class Student {
    String name="Pratik";
    int roll_number=1;
    int class_number=12;

    public Student(String name, int roll_number, int class_number){
        name=name;
        roll_number=roll_number;
        class_number=class_number;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1=new Student("Accio",15,25);

        System.out.println(s1.name);
        System.out.println(s1.roll_number);
        System.out.println(s1.class_number);
    }
}
```

Accio 15 25

name = name

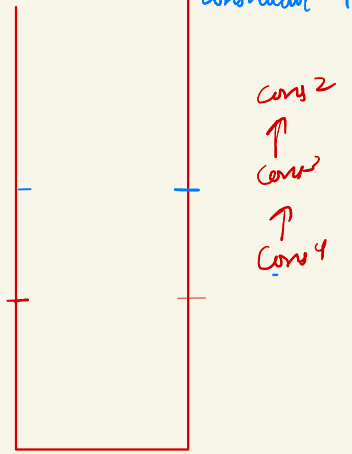
name = "accio"

Constructor Chaining

✓ Student s1 = new Student("Acxio", 10, 15);

✓ Constructor 2 is called
✓ Constructor 3 is called
✓ Constructor 4 is called

```
class Student {
    String name="Pratik";
    int roll_number=1;
    int class_number=12;

    // parametrized constructors
    //cons 1
    public Student(){

    }

    public Student(String n){ // cons2
        System.out.println("Constructor 2 is called");
        this.name=n;
    }

    public Student(String n, int rn){ // cons3
        this(n);
        System.out.println("Constructor 3 is called");
        this.roll_number=rn;
    }

    public Student(String n, int rn, int cn){ // cons4
        this(n,rn); // calling 3rd constructor
        System.out.println("Constructor 4 is called");
        this.class_number=cn;
    }
}
```



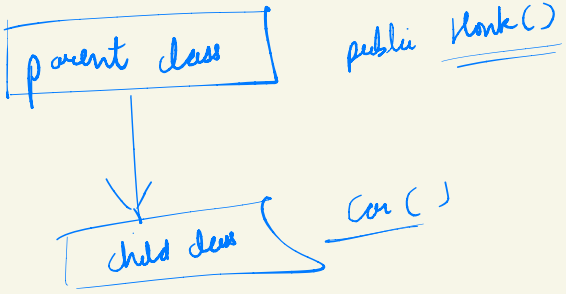
1) constructor calling should be first line.
2) order doesn't matter.
3) there should be atleast 1 constructor with no constructor call.

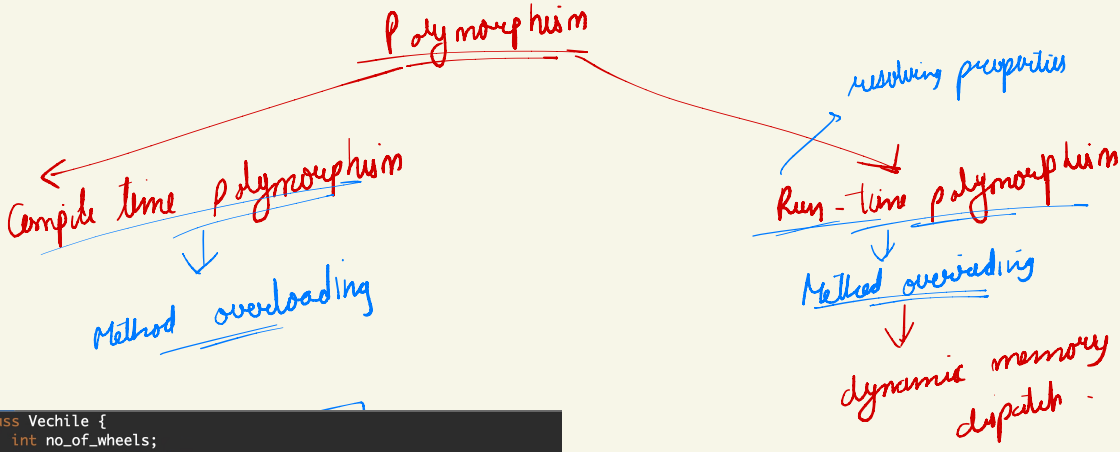
Polymorphism →

poly + morph
↓ ↓
many forms

1) Method overloading
   1) Change in number of parameters.
   2) change in data type of parameters.

2) Method overriding






```java
class Vechile {
    int no_of_wheels;
    String color="Blue";

    public void honk(){
        System.out.println("Honking from the Vechile class");
    }
}

class Car extends Vechile {
    int model_number;

    @Override
    public void honk(){
        System.out.println("Honking from Car  class");
    }
}

class Bike extends Vechile {
    public void honk(){
        System.out.println("Honking from bike");
    }
}

public class Main {
    public static void main(String[] args) {
        // upcasting
        Vechile v= new Car();
        v.honk();
    }
}
```

v = Vechile

v.honk()

at compile time ⇒ v.honk() ⇒ honking from Vechile class

v = Car

v.honk()

at run time = v.honk() ⇒ honking from Car class

compile

b

5

late binding

dynamic binding

```
public static void main(String[] args) {
    Bank b; // at compile time "b" is of Bank type
    // upcasting
    b=new SBI(); // at run time, when i allot space in Heap, "b" is ac
    System.out.println("Rate of interest is "+b.rateOfInterest());

    b=new Icici();
    System.out.println("Rate of interest is "+b.rateOfInterest());
}
```

Heap

roi → 3

Icici

roi → 4

b = Bank()

Compile → Bytecode → run

b → 3

b → 4

Access Specifiers

1) private → accessible only within the class.

2) default → within class

within package → folder of your class

→ group of same type of classes

3) protected → within class

→ within package

→ outside package but by sub class (inherit and)

X not accessible outside package

4) public → can be accessed anywhere / every where

Encapsulation

↳ wrapping code / hiding code inside a single unit

⇓

we can achieve encapsulation by using "private" keyword

Abstraction

↳ hiding details / hiding implementation

showing only functions.

⇓

abstract classes / Interface

Abstract classes ⇒ classes which can have abstract methods
normal methods

abstract methods ⇒ no method body

Interface

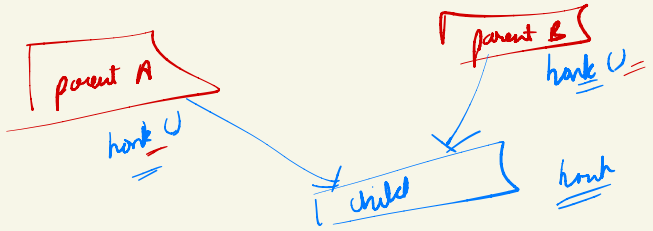
⇒ you can have only abstract methods

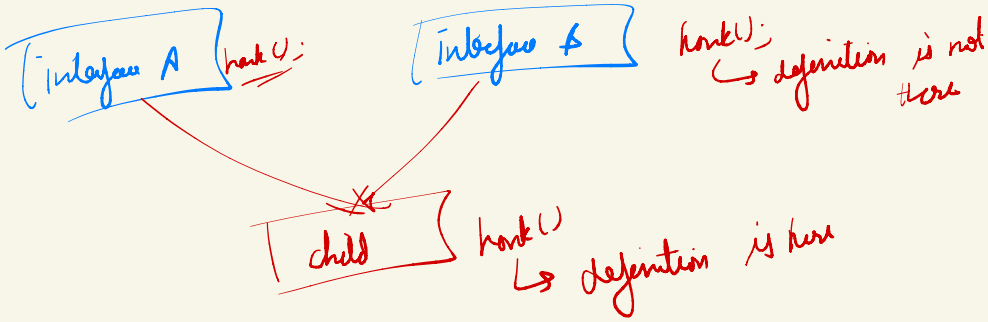
to achieve 100% abstraction

Is multiple Inheritance possible in Java?

↳ possible using Interface

to achieve multiple inheritance





---

final

1) OOPs is important.
2) OOPs is easy.

We have put in 12 hours to OOPs [create Notes and revise]

and we had fun

skills matter.

1) You didn't come this far to come this far.