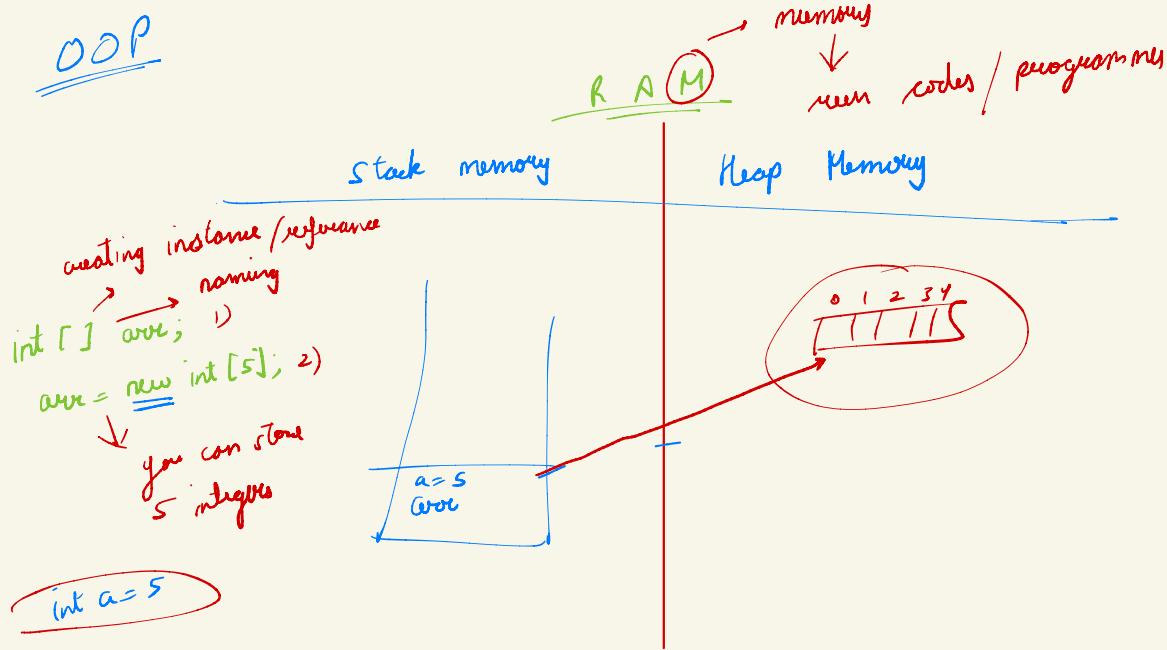



OOP



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;
b=5;)^①

a=5 ^②

b=2

```

class HelloWorld {
    space is created in stack
    for a function
    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);

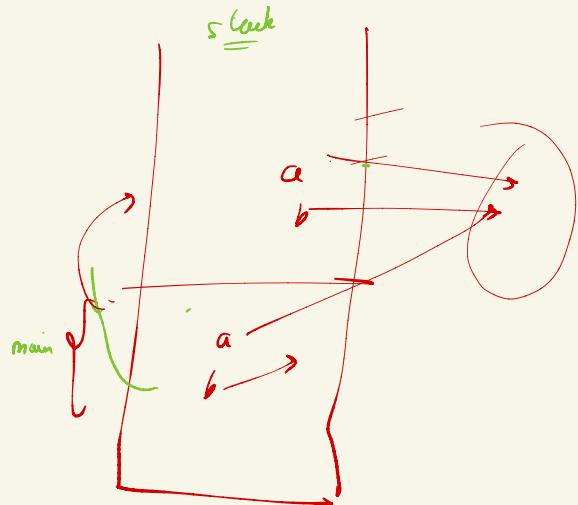
        swap(a,b);

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

```

$$\begin{array}{l} a=2 \\ b=5 \end{array}$$

$$\begin{array}{l} a=2 \\ b=5 \end{array}$$



Object Oriented Programming

class → blueprint

objects

→ cores

DRY

don't repeat yourself

leads

(how ends)

Blueprint



color?

sunwurf

```

class Car {
    int model-number;
    String color;
    int mileage;
}

```

Student name
null class
group

Teacher subject class

```

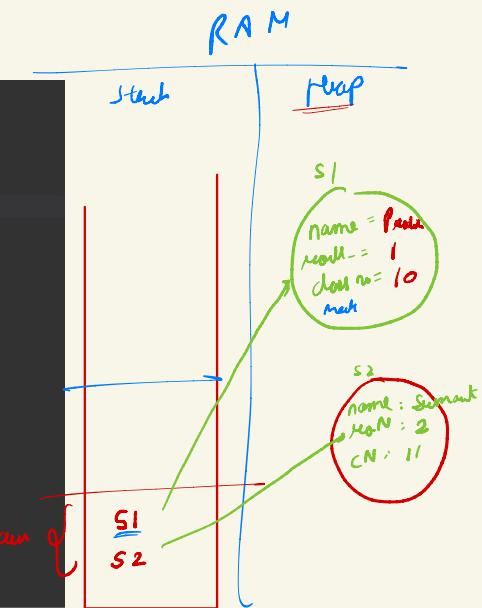
class Student {
    String name;
    String class;
    int roll-number;
}

```

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

Class → attributes / properties
methods / functions

```
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 → anyone can access, even outside the class

- 1) public
- 3) private

you can access private
members inside the
class

2) protected → only sub-classes
can access.

4) default ↓
can be accessed
within the same
package

Constructors

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

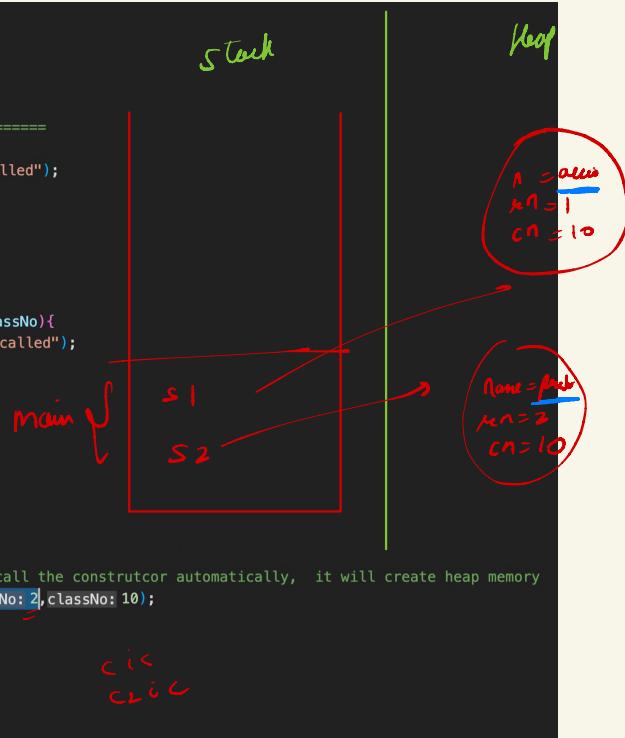
Types

1) default → constructor with no parameter

2) parametrized constructor ↳ constructors with parameters.

```
class Student {  
    String name;  
    int roll_number;  
    int class_number;  
  
    // default constructor ======  
    public Student(){  
        System.out.println("Constructor is called");  
        name="Accio";  
        roll_number=1;  
        class_number=10;  
    }  
  
    // parametrized constructor  
    public Student(String a, int rollNo, int classNo){  
        System.out.println("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 constructor 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

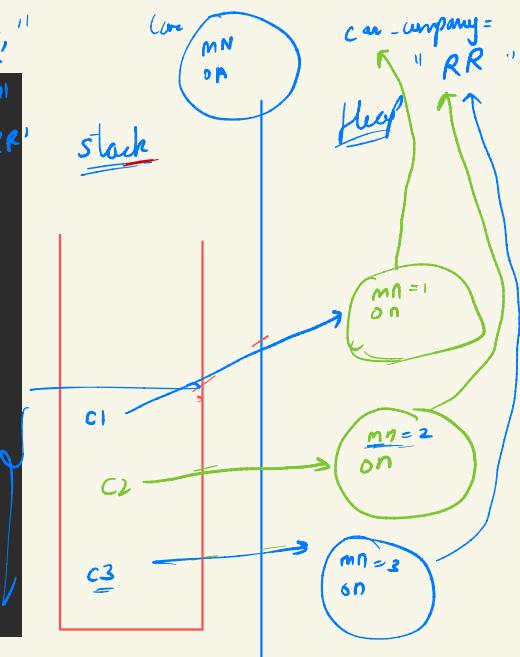
public static void main
public static int fun

new object

```
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);  
    }  
}
```

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



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

```

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

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

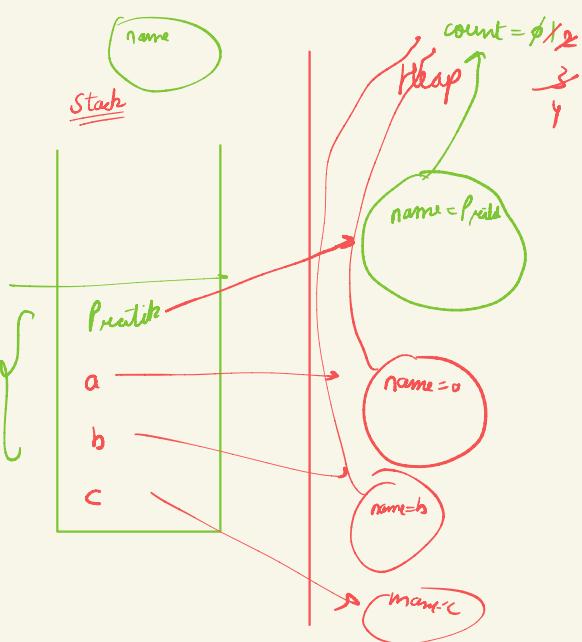
public class Main
{
    public static void main(String[] args) {
        1) Student Pratik=new Student("Pratik");
        System.out.println(Pratik.count);

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

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

class Main {
 public static void main() {
 Main.main();
 }
}

3
3

Patterns of OOPS

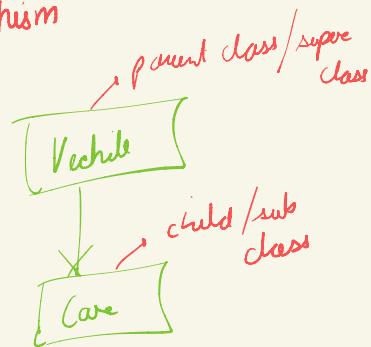
- 1) Inheritance
- 3) Abstraction

- 2) Encapsulation
- 4) Polymorphism

⇒ Inheritance

Reusability

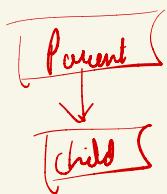
class Vehicle {
 int no-of-wheels;
 string color;



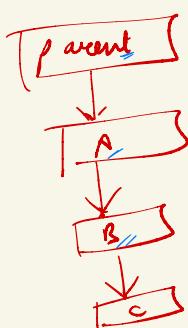
- 3
1) Parent class constructor is called first, after that child class constructor is called.

Types of Inheritance

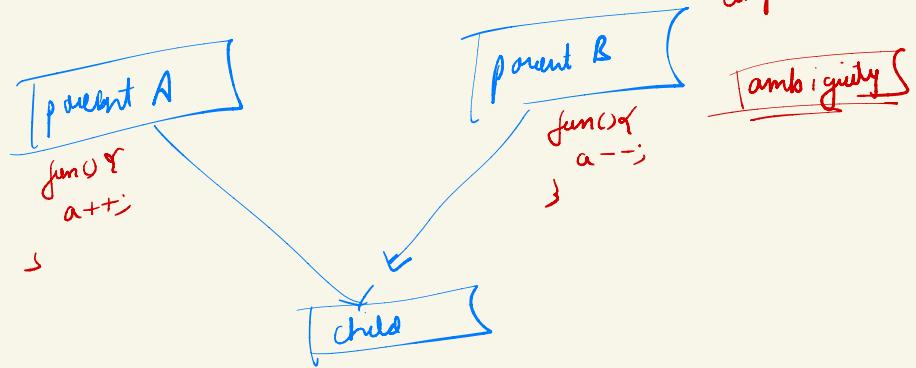
1) Single Inheritance



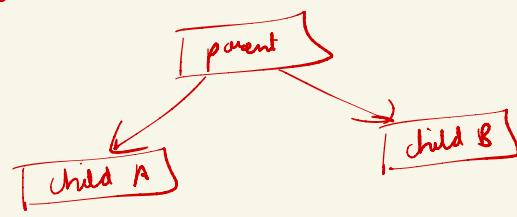
2) Multi-level Inheritance



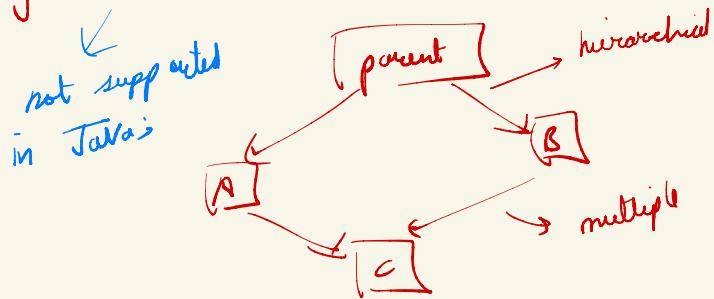
3) Multiple Inheritance [not in Java]



4) Hierarchical Inheritance



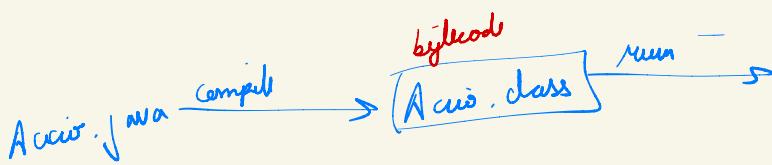
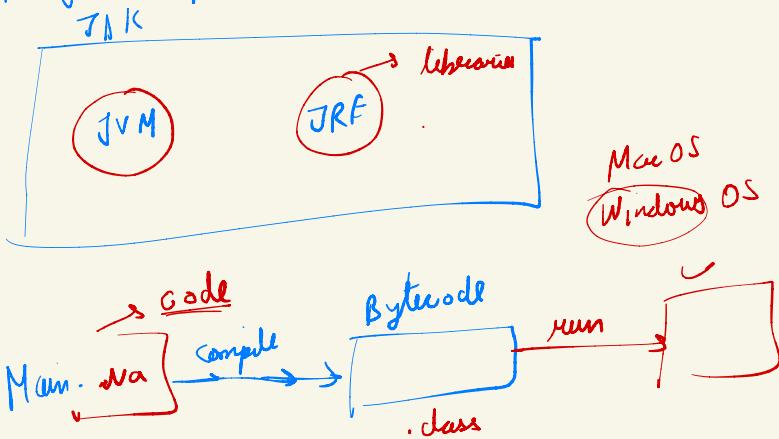
5) Hybrid Inheritance [sum of Hierarchical and multiple]



JDK → Java development kit
↓
libraries

JVM → Java virtual Machine.

Java is platform-independent



#this → this 'refers' to own instance of class.

```
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);  
        }  
    }  
}
```

name = name
name = "Accio"

Constructor Overloading

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

```
class Student {  
    String name="Pratik"; acc  
    int roll_number= 10; acc  
    int class_number= 15; acc  
  
    // parameterized constructors  
    // cons 1  
    public Student(){  
    }  
  
    public Student(String n){ // cons2  
        1 System.out.println("Constructor 2 is called");  
        2 this.name=n; acc  
    }  
  
    public Student(String n, int rn){ // cons3  
        1 this(n); acc  
        2 System.out.println("Constructor 3 is called");  
        3 this.roll_number=rn; acc  
    }  
  
    public Student(String n, int rn, int cn){ // cons4  
        1 this(n,rn); // calling 3rd constructor acc  
        2 System.out.println("Constructor 4 is called");  
        3 this.class_number=cn; acc  
    }  
}
```

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

cons 2
↑
cons 3
↑
cons 4

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

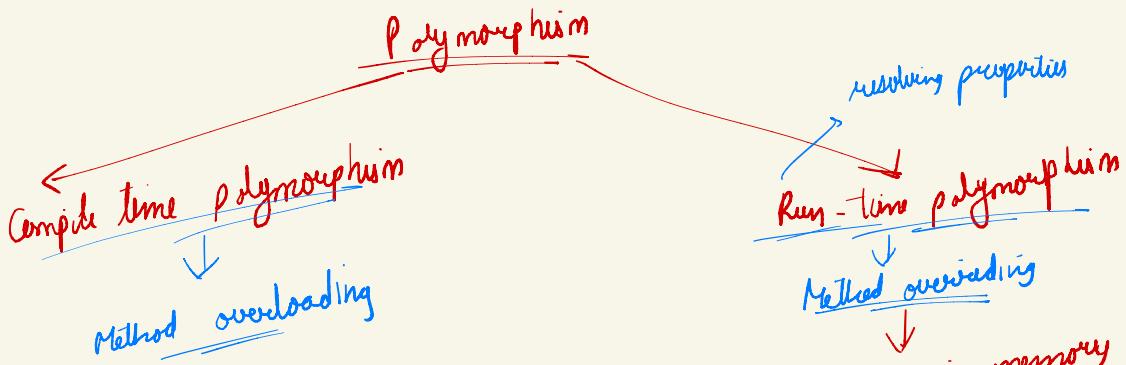
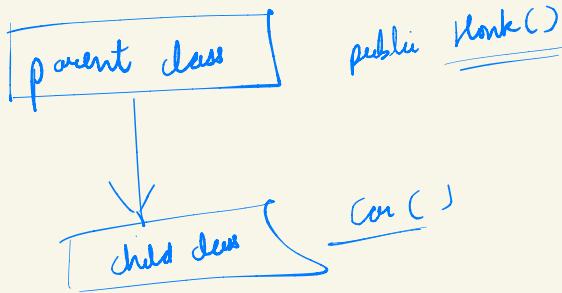
Poly morphism →

poly + *morph*
many forms

Method overloading

- 1) Change in number of parameters
- 2) Change in data type of parameters

Method overriding



```

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();
    }
}

```

$v = \underline{\text{Vechile}}$ $v.\underline{\text{honk()}}$
 at compile time $\Rightarrow v.\underline{\text{honk()}}$
 \downarrow
 Honking from Vechile class

$v = \underline{\text{Car}}$
 $v.\underline{\text{honk()}}$
 at run time $\leftarrow v.\underline{\text{honk()}}$
 \Rightarrow Honking from Car class