19. Object ~~Oriented~~ Oriented Programming

(Using objects and classes to do programming)

19.1 Classes And Objects

using this we can structure how to write code. (style of writing code).

group of these entities.

entities in the world eg. pen

pen can have attribute like color

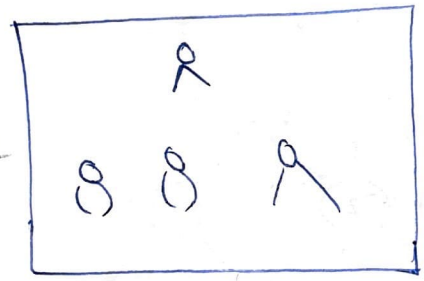
blue ~~&~~ yellow.

We can create Pen ~~object~~ class in Java.

we can have function to change color.

eg. changeColor()

- Collection of similar objects is called class.



class is a group of these entities

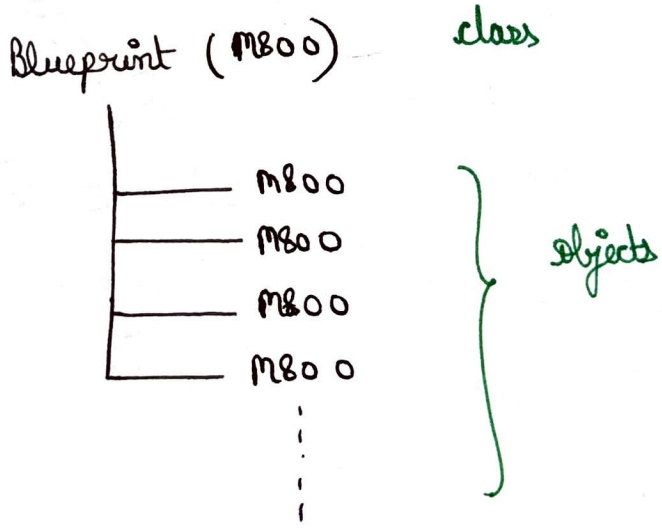
Eg. In a class all students have same uniform, grade, books, age.

Here student is an ~~object~~ object.

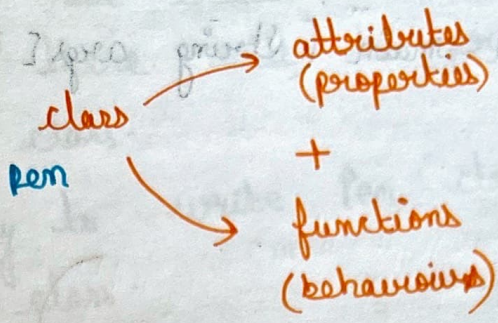
Let us say we are writing program to make Maruti800.

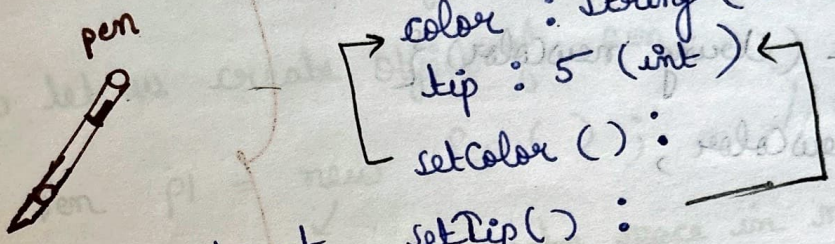
So first we give blueprint for M800.

Then it will produce various M800.



class has attributes (properties) + functions.





We can construct many pens having the above properties and functions. Let us write code.

Name of File : OOPS.java
Name of public class : OOPS.

We can construct many classes in single Java file.

> By convention class name in Java starts with capital letter.
> By convention function name in Java starts with small letter.

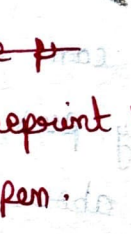
Let us create class Pen.

```
public class OOPS {

}
class Pen {
    String color;
    int tip;
}
```

Now let us create functions in the class Pen.

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

}       }

class Pen {
    String color;
    int tip;
    void setColor (String newColor) {
        color = newColor;
    }
    void setTip ( int newTip) {
        tip = newTip;
    }
}

class Student {
    String name;
    int age;
    float percentage; // cgpa
    void calcPercentage (int phy, int chem, int math) {
        percentage = (phy + chem + math) / 3;
    }
```

Blue ~~p~~ Blueprint of pen.



Another class Student is also created.

In Java if we create Pen class, Student class above public class then there will be problem in creating objects of class.
So we try to write Pen class, Student class below public ~~sto~~ class.

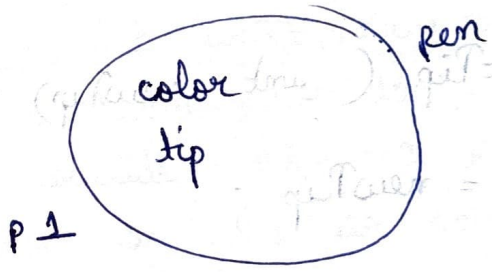
- By convention main function should be public and static.

Now let us create object of pen.

```
Pen p1 = new Pen();   // created a Pen obj. p1.
```

↙ this keyword creates space in memory which can store all attributes of pen.

This objects is created in heap.
Objects are ~~stor~~ created ~~in~~ in heap.



To set color for p1 we use:

```
p1.setColor("Blue");
```

- There should be a public class in our Java file, in a single Java file we can create multiple classes.
- In public class we write:

```
public static void main(String args[]) {
}
```

↳ Our program starts execution with main function.

- Compiler first looks for public class and then ~~sta~~ searches for for main function.
- main ~~fu~~ func^n should be public and static.

```
public class OOPS {

    public static void main(String args[]) {
        Pen p1 = new Pen();   // created a pen obj. p1
        p1.setColor("Blue");
        System.out.println(p1.color);
        p1.color = "Yellow";
        System.out.println(p1.color);
    }
}

class Pen {
    String color;
    int tip;

    void setColor(String newColor) {
        color = newColor;
    }

    void setTip(int newTip) {
        tip = newTip;
    }
}

class Student {
    String name;
    int age;
    float percentage;

    void calcPercentage(int phy, int chem, int math) {
        percentage = (phy + chem + math) / 3;
    }
}
```

this is special type of constructor

```
p1.setTip(5);
System.out.println(p1.tip);
```

why pen and student are declared below the public class.

19.2 Access 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

Access modifiers defines access for an object.

Code for maps is written in a package, code for GPay is written in a different package, code for chrome is written in another different package.

Ideally we don't want data of GPay to be accessed by maps or chrome, we don't want sensitive information to go outside from a package. For, this purpose we have access specifiers or access modifiers.

Access Modifiers are keywords in Java which tells which data is accessible to which part.

- private -
  - access is allowed only inside a class.
  - access is not allowed by another class, function outside the class within the package.
  - access is not allowed to ~~be~~ outside package subclass.
  - access is not allowed outside package.

default: By default if we don't mention any access specifiers then it is considered ~~default~~ default.

- it can be used within class.
- access by other things are allowed within package.
- access is denied to subclass outside the package.
- access is denied outside package.

protected -
- access allowed within class.
- access allowed within package.
- access allowed to subclass outside package.
- access not allowed outside package.

public -
- access allowed within class.
- access allowed within package.
- access allowed to subclass outside package.
- access allowed outside package.

```
public class OOPS{
    public static void main(String args[]){
        BankAccount myAcc = new BankAccount();
        myAcc.username = "shradhakhapra";
        myAcc.password = "abcdefghi"; //this is not allowed as password is not accessible.
    }
}
class BankAccount{
    public String username; // visible to everyone
    private String password; // not accessible outside class
}
```

Now if we create object of BankAccount :

```
BankAccount myAcc = new BankAccount();

myAcc.password = "abcdefghi"; // not allowed.
```

But we can create a public function in BankAccount.

Following is allowed :

```
public class OOPS {
    public static void main (String args[]) {
        BankAccount myAcc = new BankAccount();
        myAcc.username = "ShraddhaKhapra";
        my.Acc.setPassword ("abcdefghi");
    }
}
class Bankaccount {
    public String username;
    private String password;
    public void setPassword (String pwd) {
        password = pwd; // we are accessing private inside func
    }
}
```

Let us say we have created BankAccount class and we want to store user account info.. Let us say in a team someone has written software about upi, would someone allow permission to access user's password and change it.

So, we use the concept of Access Modifiers.

19.3 Getters and Setters

Get : to return the value

Set : to modify the value

this : this keyword is used to refer to the current object. (special)

For property associated with object like :

Pen : color
tip

} to prevent its access we used private, and to access we provided functions.

Now even if we don't have function then we need to create functions. We create get type and set type function.

```
class Pen {
private String color;
private int tip;

//getter String getColor() {
        return this.color;
    }

//setter void setColor(String newColor) {
        this.color = newColor;
    }

//getter int getTip() {
        return this.tip;
    }

//setter void setTip(int tip) {
        this.tip = tip;
    }
```

this → current object (properties of that particular object)

tip → value passed (external)

this.tip → (internal) property of pen

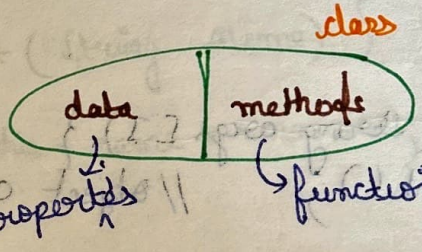
~~pppp~~

```
public class OOPS{

    public static void main (String args [ ]) {

        Pen p1 = new Pen ( );          // created a pen object p1
        p1.setColor ("Blue");
        System.out.println ( p1.getColor ( ));

        p1.setTip (5);
        System.out.println (p1.getTip( ));

        p1.setColor ("Yellow");
        System.out.println ( p1.getColor ( ));

    }
}
```

19.4 Encapsulation

There are four major pillar of OOPS:
- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

Encapsulation:



Encapsulation is defined as the wrapping up of data and methods under a single units. It also implements data hiding.

Data hiding means data ~~is~~ which is useless or sensitive for user which we hide from them by making them private or protected or default i.e. by using

access modifiers.

19.5 Constructors

Constructor is a special method which is invoked automatically at the time of object creation.

- Constructors have the same name as class or structure.
- Constructors don't have a return type. (not even void).
- Constructors are only called once, at object creation.
- Memory allocation happens when constructor is called.

It is called automatically when we create a object.

When we write:

```
Pen p1 = new Pen();
```

then immediately a constructor is called. Constructor is mainly used for initializing objects.

Then memory is given to object and then we can store color, tip, etc.

> Class name and constructor name are same.

```
public class OOPS {

    public static void main (String args[]) {
        Student s1 = new Student();    // object creation
    }
}
class Student {

    String name;
    int roll;
```

Student () { // constructor

} // value initialisation purpose

}

19.6 Types Of Constructors

There are three types of constructors in Java :

- Non-Parameterized : we don't take any parameters or arguments.
- Parameterized : we take parameters.
- Copy Constructor

```
public class OOPS {
    public static void main(String args[]) {
        Student s1 = new Student();
        Student s2 = new Student("Neha");
        Student s3 = new Student(123);
        Student s4 = new Student("Neha", 1234);
    }
}
class Student {
    String name;
    int roll;

    Student() {                          // non-parameterized constructor
        System.out.println("Constructor is called");
    }

    Student(String name) {               // parameterized constructor
        this.name = name;
    }

    Student(int roll) {                  // parameterized constructor
        this.roll = roll;
    }
}
```

no error, by default non-parameterized constructor will be called.

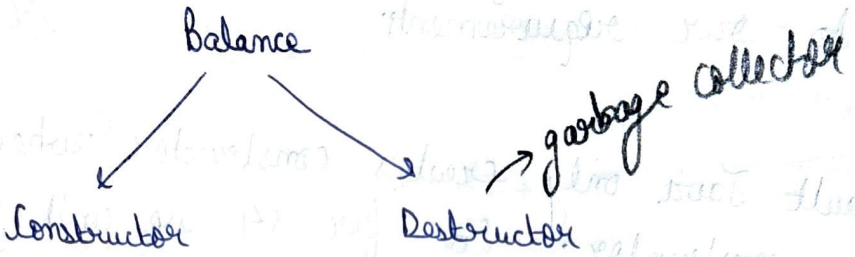
which constructor will be called depends on parameter matching

no constructor will be called, this will generate error because no constructor exists whose first parameter is String and second parameter is is roll no.

In a class we can create multiple constructors according to our requirements.

By default Java only creates constructor when there is no other constructor. for s4 we will get error.
(Here there are already other constructors, so Java cannot create constructor.)
The above concept is called Constructor Overloading. This is an example of polymorphism.

19.9 Destructors



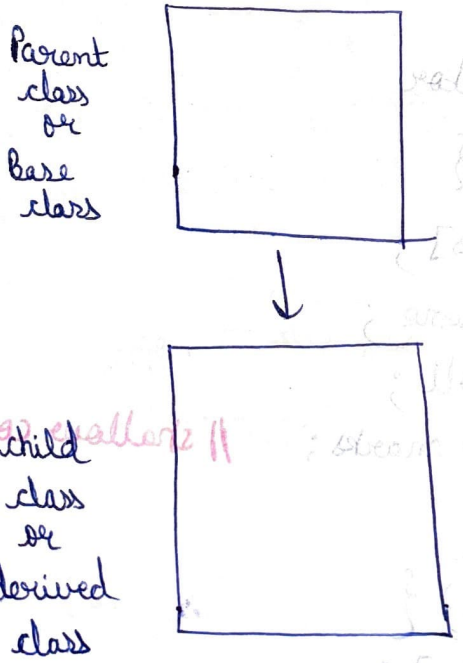
If there is a constructor then there should be destructor also.

- In Java we have ~~for~~ garbage ~~collector~~ collector as destructor.
- Garbage collector checks from time to time whether variable, object, ~~array~~ arrays are being used in the program; if they are not being used then it removes them.

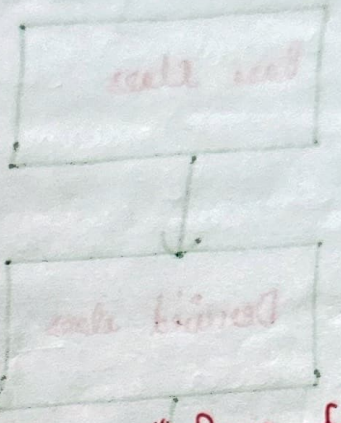
We don't write destructor program in Java, it is done automatically by ~~destructor~~ garbage collector.

19.10 Inheritance

~~Into~~ Inheritance is when properties and methods of base ~~class~~ class are passed onto a derived class.



```
public
class OOPS {
    public static void main (String args [ ]) {
        Fish shark = new Fish ();
        shark.eat ();
    }
}
```

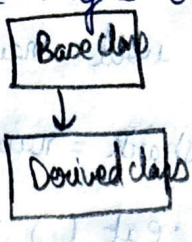


```
// Base class
class Animal {
    String color;
    void eat() {
        System.out.println ("eats");
    }
    void breathe () {
        System.out.println ("breathes");
    }
}
class Fish extends Animal {          // Derived class
    int fins;
    void swim () {
        /* fish will inherit everything
        from animal and also has
        its own properties fins and
        */
        System.out.println (" swims in water");
    }
}
```

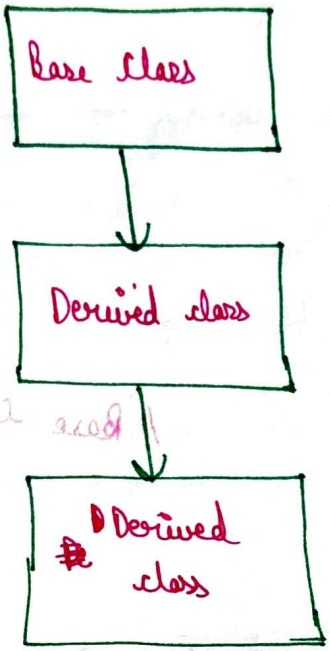
o/p: eats

19.11 Single Level Inheritance

When we have single base and single derived class.



19.12 Multi Level Inheritance



Animal (eat, breathe) color

↓

Mammal (legs)

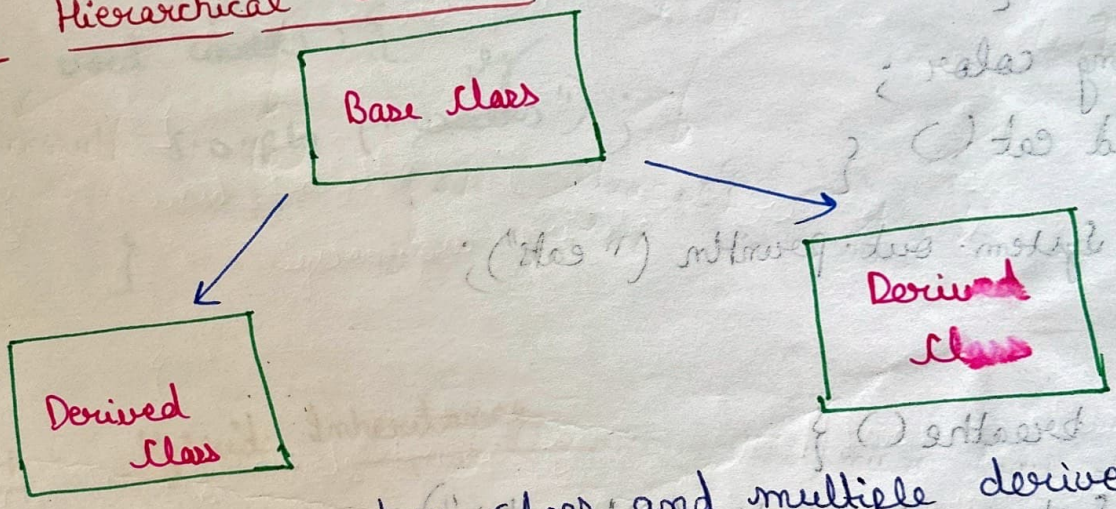
↓

Dog (breed)

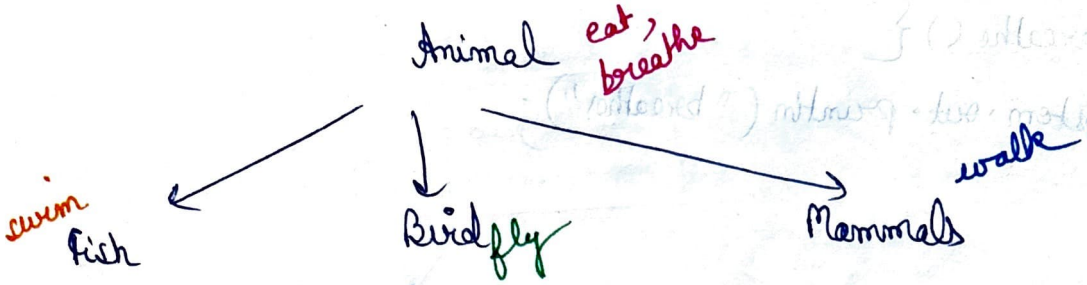
```
class OOPS {
public static void main (String args[]) {
        Dog dobby = new Dog(); /* for dobby we can define
        dobby.eat();            eat, legs, color,
        dobby.legs = 4;         breed because it inherits from Mammal,
        System.out.println(dobby.legs);   Animal. */
    }
}
class Animal {
        String color;
        void eat() {
            s.o.pln("eats");
        }
```

```
    void breathe(){
        System.out.println("breathes");
    }
}
class Mammal extends Animal {
    int legs;
}
class Dog extends Mammal{
    String breed;
}
```

19.13 Hierarchical Inheritance



Here we have single base class and multiple derived classes. All derived class will inherit property of base class.

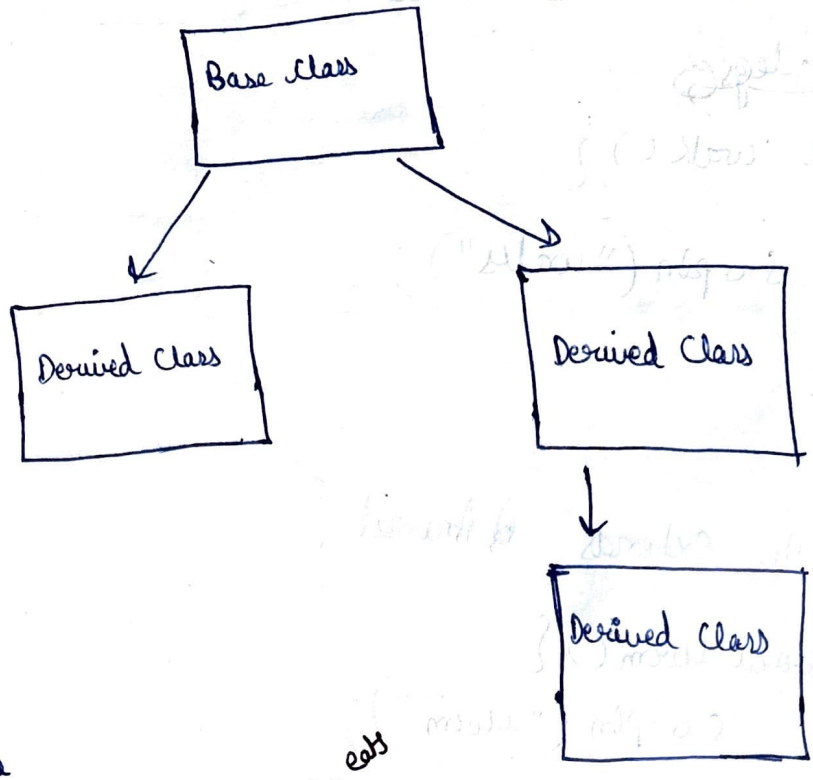


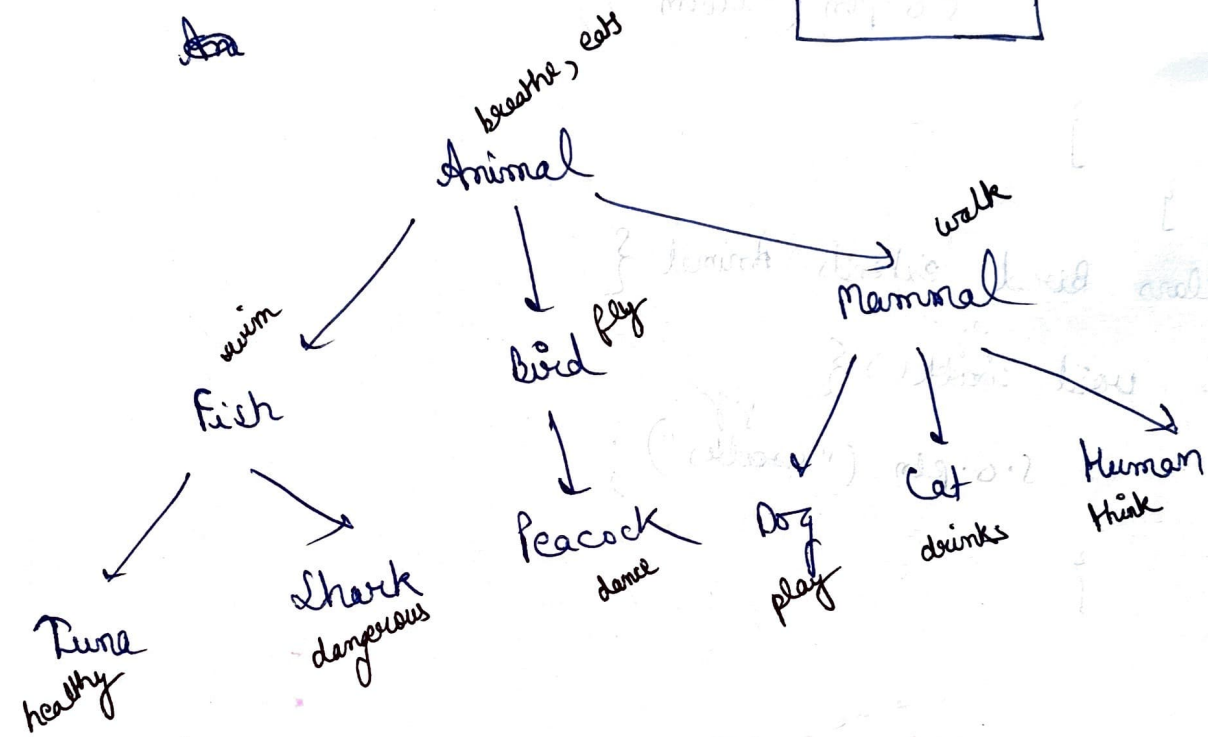
Fish, Bird, Mammals have different properties then inherit properties of Animal.

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

    ~~Dog dobby = new Dog();~~
    ~~dobby.eat();~~
    S.o.pln (dobby.legs);

    Bird eagle = new Bird();
    eagle.eat();
    S.o.pln(eagle.eat());
}
class Animal {
    String color;
    void eat() {
        System.out.println ("eats");
    }
    void breathe() {
        System.out.println ("breathes");
    }
}
```

```
class Mammal extends Animal {
    int legs;
    void walk() {
        S.o.pln("walks");
    }
}
class Fish extends Animal {
    void swim() {
        S.o.pln("swim");
    }
}
class Bird extends Animal {
    void fly() {
        S.o.pln("flies");
    }
}
```

19.14 Hybrid Inheritance

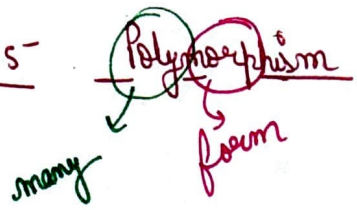





```
class OOPS {
    public static void main(String args[]) {
        Cat puchki = new Cat();
        puchki.breathe();
        puchki.drinks();
    }
}
class Fish extends Animal {
    void swim() {
        S.o.ptln("can swim");
    }
}
class Animal {
    void breathe() {
        S.o.ptln("All animals can breathe");
    }
    void eats() {
        S.o.ptln("eats");
    }
}
```

```
    void healthy () {
        s.o.ptln ("healthy to eat");
    }
}
class Shark extends Fish {
    void dangerous () {
        s.o.ptln ("sharks are dangerous");
    }
}
class Bird extends Animal {
    void fly ()
    s.o.ptln ("birds can fly");
    }
}
class Peacock extends Bird {
    void dance () {
    s.o.ptln ("peacock can dance");
    }
}
class Mammal extends Animal {
    void walk (){
    s.o.ptln ("they can walk");
    }
}
class Dog extends Mammal {
    void play() {
    s.o.ptln ("loves to play");
    }
}
class cat extends Mammal {
    void drinks() {
    s.o.ptln ("drinks milk");
    }
}
class human extends Mammal {
    void think () {
    s.o.ptln ("humans have cognitive
    ability");
    }
}
```

19.15 Polymorphism : When in multiple forms we try to do similar task then it is called polymorphism.



There are two types of polymorphism :

> Compile time Polymorphism (static) - When compiler is running our code then when it observes different forms then it is called compile time polymorphism. Method Overloading.

> Run time Polymorphism (dynamic) - When polymorphism is observed during runtime then it is called runtime polymorphism. Method Overriding.

19.16 Method Overloading -

Multiple functions with same name but different parameters. Either number of parameters should be different or types of parameters should be different.

eg. Let us say we have a Calculator class.

```
Calculator {
    sum (int a , int b) {}
    sum (float a , float b)
    sum ( int a, int b , int c)
}
```

Return type does not affect the concept of function overloading.

Code :

```
public class OOPS{
    public static void main(String args[]){
        Calculator calc = new Calculator();
        System.out.println(calc.sum(1,2));
        System.out.println(calc.sum((float)1.5,(float)2.5));
        System.out.println(calc.sum(1,2,3));
    }
}
class Calculator{
    int sum(int a, int b){
        return a+b;
    }
    float sum(float a, float b){
        return a+b;
    }
    int sum(int a, int b, int c){
        return a+b+c;
    }
}
```

↳ we typecast to float because by default Java considers 1.5 as double.

19.17 **Method Overriding** —

Parent and child classes both contain the same function with a different definition.

class A ≡ eats()

↓ both func^{n} have same name, same parameters but have different definitions

class B ≡ eats() i.e. perform different work.

eg. Animal class → void eat ()
{
"eat Anything";
}

Deer class → void eat ()
{
"eats grass";
}

Code :

```
public class OOPS {
    public static void main (String args [ ]) {
        Deer d = new Deer ();
        d.eat ();
    }
}
class Animal {
    void eat () {
        System.out.println ("eats anything");
    }
}
class Deer extends Animal {
    void eat () {
        System.out.println ("eats grass");
    }
}
```