LLD - 3 OOP

Agenda

→ Inheritance & Constructors ⇐ super()

→ Polymorphism ⇐

↳ Compile
↳ Runtime

Constructor Chaining

→ Other Concepts

→ Interfaces ⇐

→ Abstract Classes ⇐

→ Static =

```
Class A {
    private int a;
    private int b;
};

Class B extends A {
    private int c;
};
```

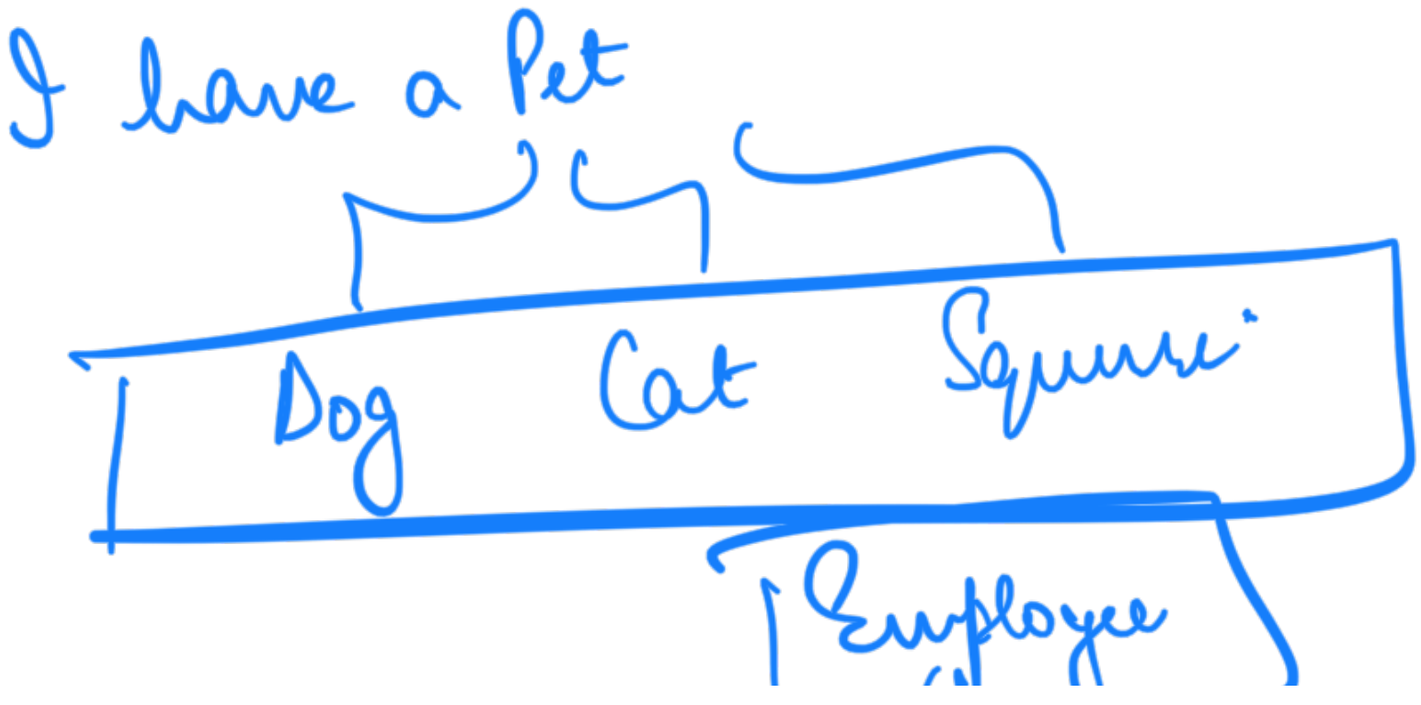
⇒ When we call constructor of a child class, before executing itself it calls the Constructor of parent class

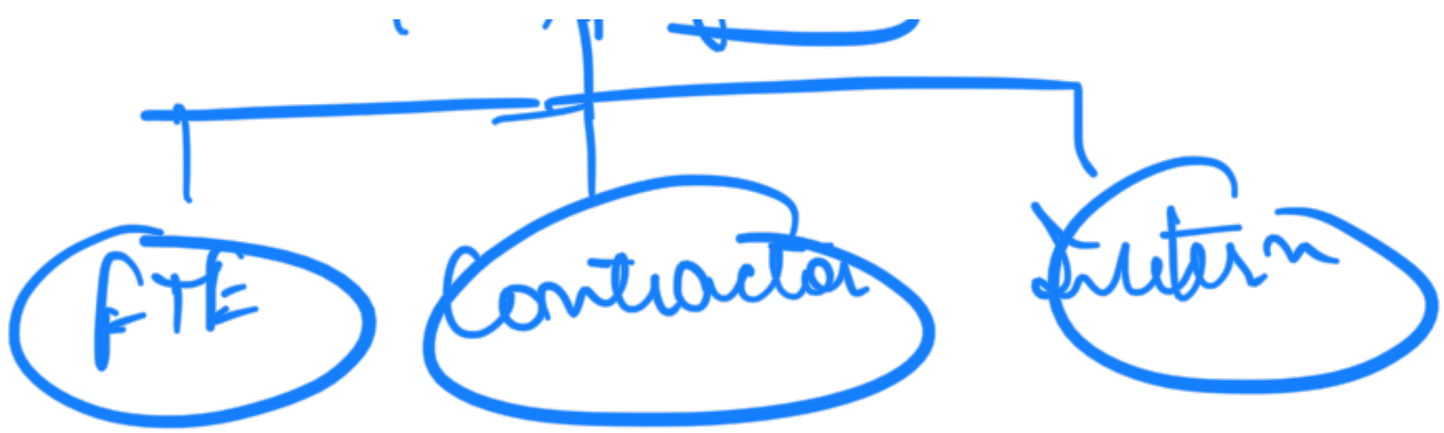
By default constructor of child class calls the constructor of parent class with NO

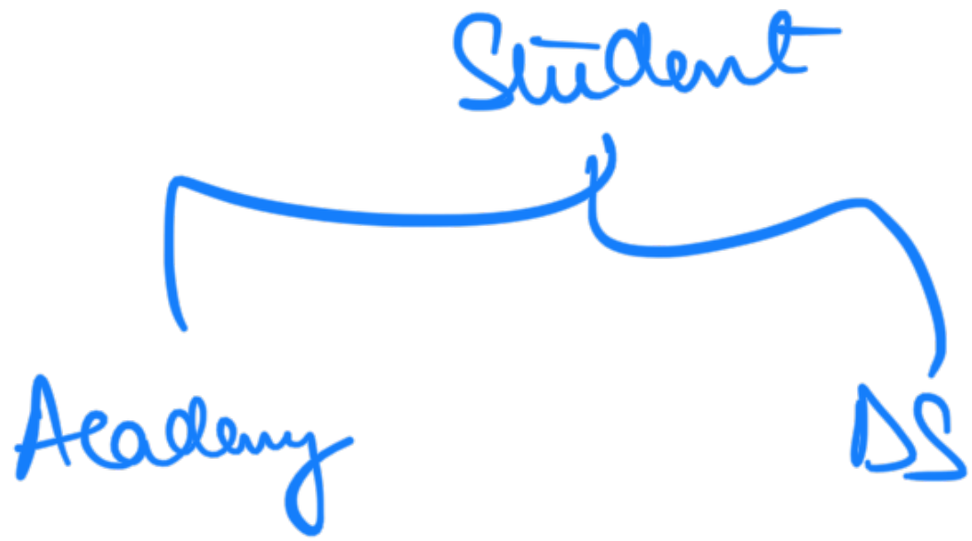
attributes.

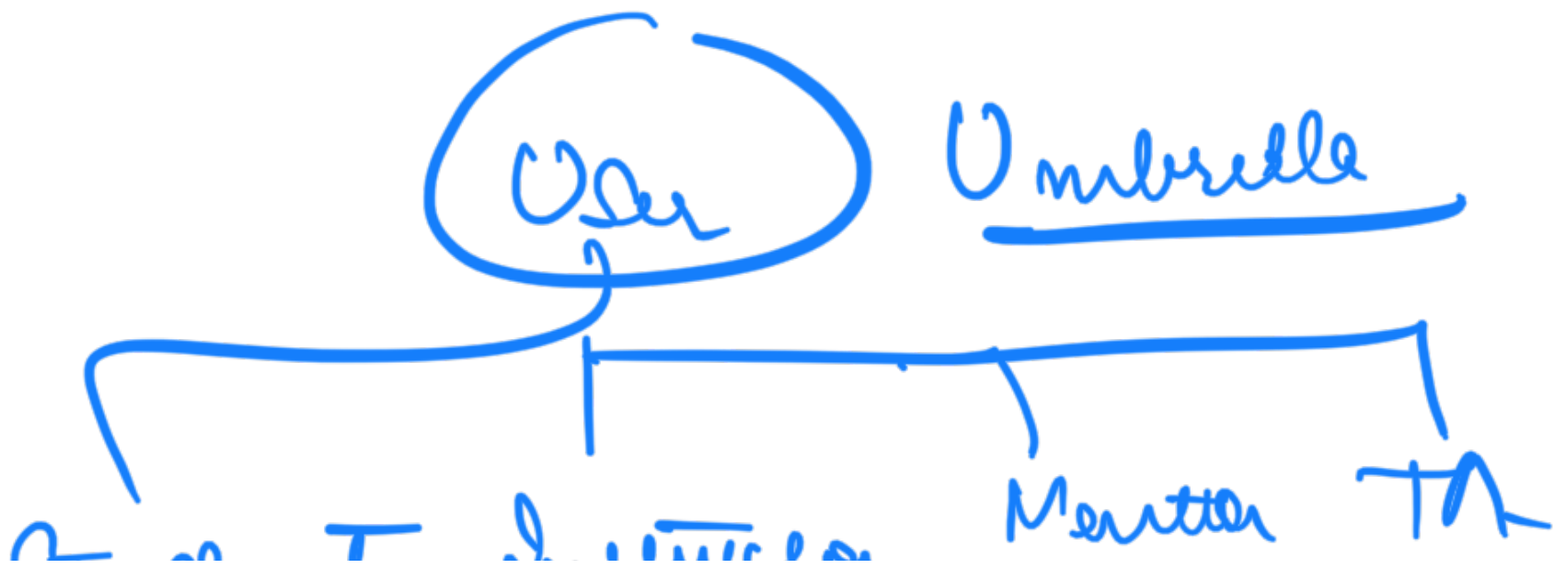
Polymorphism

We refer to objects not by their exact name




FTE
Contractor
Intern


Student
Academy
DS


User
Umbrella
Mentor

Student

Polymorphism

Many

Forms

User

Mentor

Student TA

IS-A

User u = New Student()

User u = New TA()

User u = new Mentor( )

Student s = new User( ) X

All Student are users

All user are Student X

Many form

Allows me to refer to an umbrella term rather than an exact term

User u = new Student

u. → ates of Student

Method

POLY MORPH ISM

Many Form

Class A {

print () { }

print (Hello World)

print (String name) {

print("Hello " + name);

}

print (String , int) ;

}

Overloading

Compile Time Polymorphism

)

Method Overloading happens when there are 2 or more methods with same name but diff signature

(String, int)

(String, String)

(int)

⇒ Return type doesn't play a role in deciding overloading

int print (String)

String print (String)

X

Same method Name

BUT

diff Signature

① Return type isn't a part of

Overloading

Sign: funcName(params...)

print(String)
print(int)

Method Overriding

A {

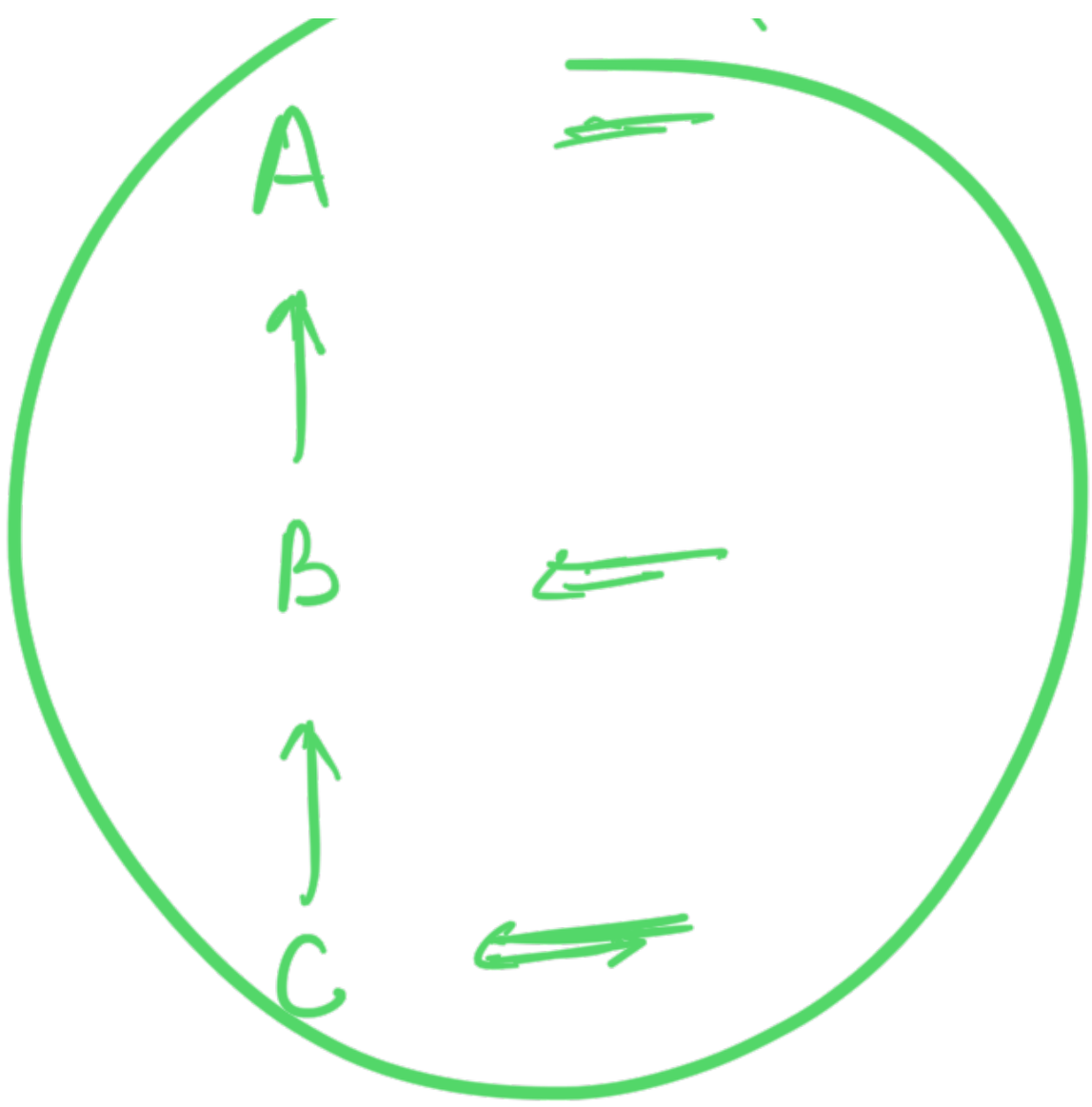
```
void doSomething() {
    cout("I am A")
}
```

```
B extends A {
    void doSomething() {
        cout("I am B")
    }
}
```

B b ⇔ new B()
b.doSomething()

⇒ When a child class has a method with exact same signature + return type, child class is said to be overriding the method of the parent class.

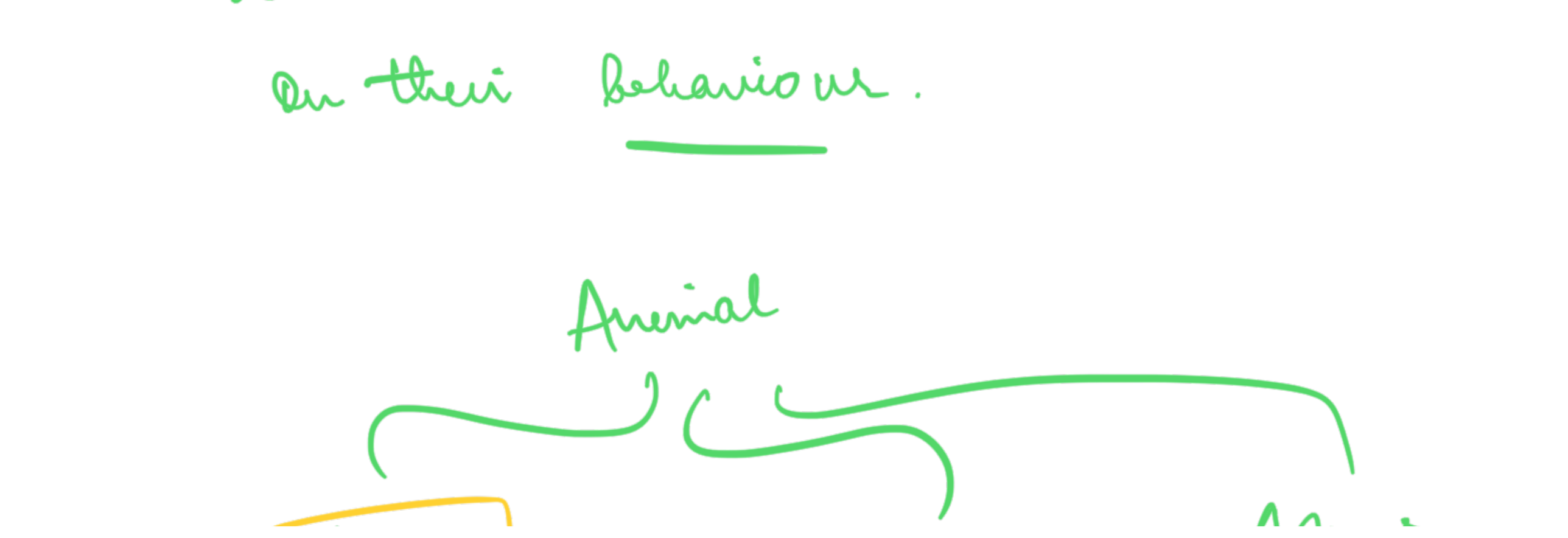
Method Overriding ⇒ RUNTIME POLYMORPHISM

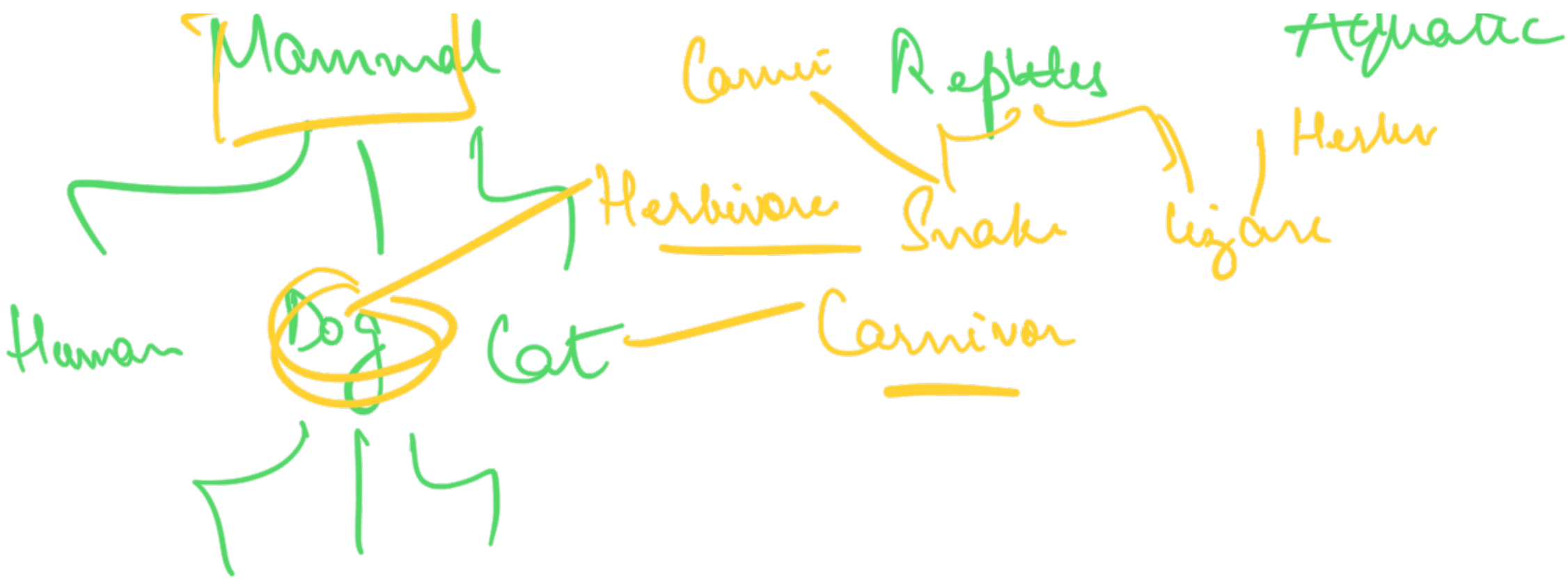


Student St = New User C

Other Concepts

Sometimes we don't classify entities based on their attributes but based on their behaviour.





Dog extends ~~Animal~~ Mammal, Herbivor{

}

Interfaces : Blueprint of behaviour

```
interface Herbivore {

    void eatPlant();

}
```

```
interface Carnivore {

    void eatAnimal();
```

class Dog extends Animal implements Herbivore {

eatPlant() {

}

}

List<Herbivore> {

Zoo {

addAnimal(Herbivore

)

→ Il class can implement ∞