Object-oriented programming:

- The programming paradigm centers around entities
- Every entity has data and actions that it can perform
- Action and data are grouped together
- More natural to how humans make sense of real-world

1. Principal of oops
    a. Abstraction
2. Pillars of Oops
   It helps to implement abstraction
    a. Encapsulation
    b. Inheritance
    c. Polymorphism

Abstraction:

- The principal on which oops is based.
- It represents a system as multiple ideas that mean making something abstract i.e. ideas.
- In other words, we can say attraction represents a complex software system in terms of different ideas
- Ideas are anything in the software system that has attributes(info or data about it) or can cause behavior
- User/Client/other don't need to know all the internal workings of an idea That means attraction hides the internal implementation of an idea
- For example, a driver doesn't care about the internals of working of a steering wheel they only care about how to use the steering wheel of a car so the car is an abstraction over the internal implementation for a driver

How oops is brought into practice:

1. Encapsulation:
    - It helps to store attributes and behavior together of an entity. (Hold everything with respect to an entity together)

- It protects the data/behavior of an entity from direct/unwanted access

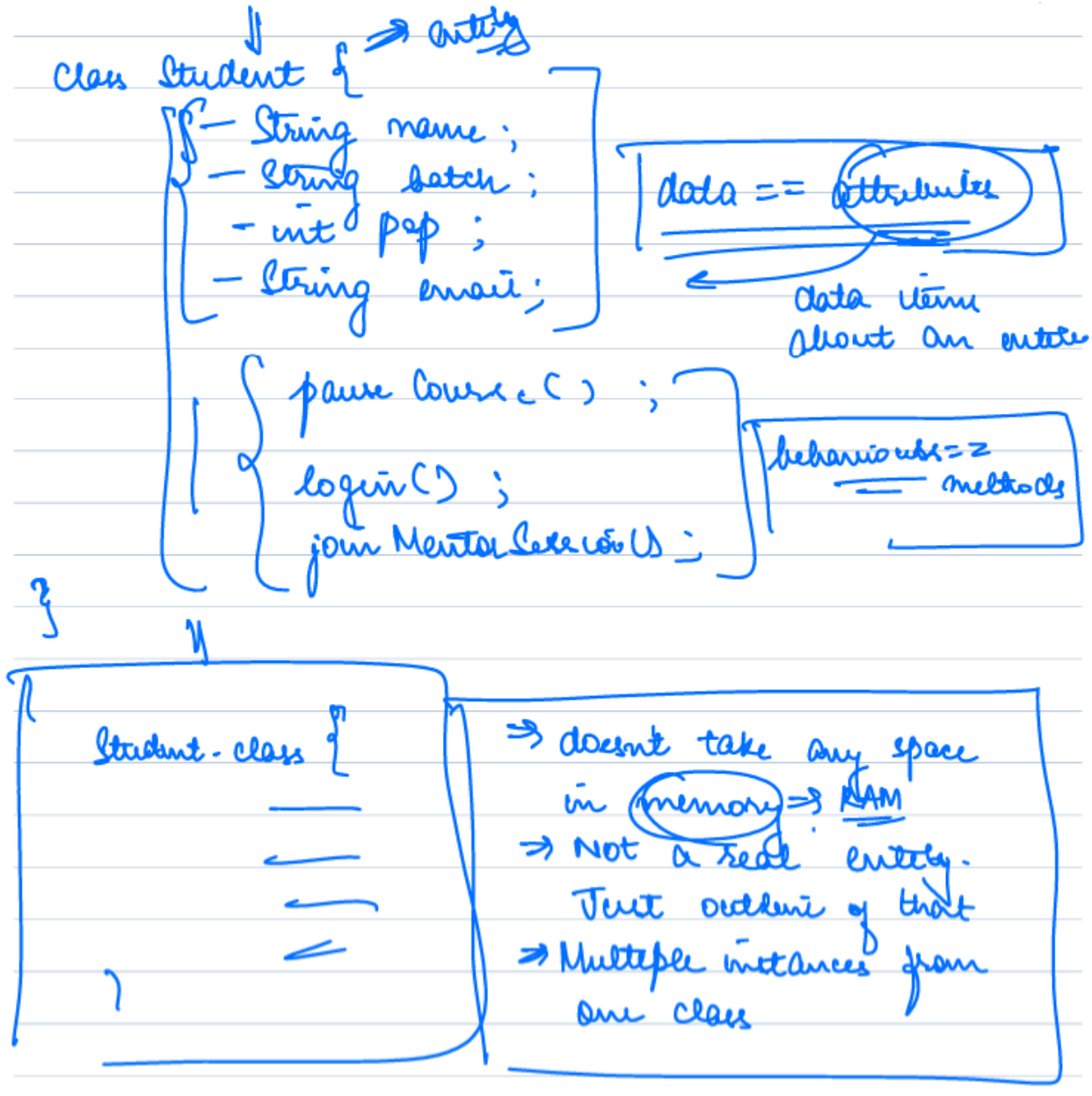
Class:

- A class is a blueprint of ideas or a template for creating objects.
- It defines the properties and behaviors that an object of that class will have.
- A class can contain fields (variables) to store data, methods to perform operations on that data, and constructors to initialize the object.

For example, consider a class named "Car". It may have fields such as "make", "model", "year", and "price", as well as methods like "start", "stop", "accelerate", and "brake".

To create an object of the "Car" class, you would use the "new" keyword and specify the class name, like this:

```
Car myCar = new Car();
```



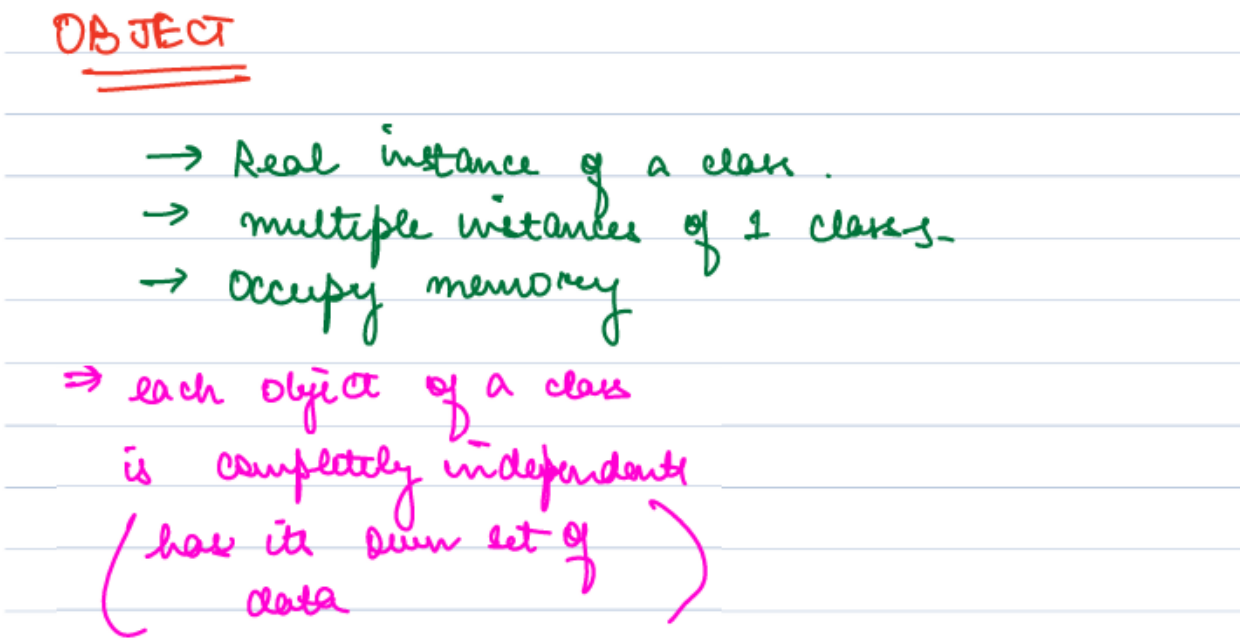
Object:

- Objects are instances of a class.
- Each object has its own set of values for the fields defined in its class and can perform operations using the methods defined in its class.

Using the "Car" class example, you could create multiple objects of the "Car" class, each with different values for the fields:

```
Car car1 = new Car();
car1.make = "Toyota";
car1.model = "Corolla";
car1.year = 2020;
car1.price = 20000;

Car car2 = new Car();
car2.make = "Honda";
car2.model = "Civic";
car2.year = 2021;
car2.price = 22000;
```

Here, "car1" and "car2" are both objects of the "Car" class, with different values for their fields.



Access Modifier:

Access modifiers are keywords that are used to set the accessibility or visibility of classes, variables, methods, and constructors. There are four types of access modifiers in Java: public, private, protected, and package-private (default).

Public: A public access modifier makes the class, variable, method, or constructor visible to all classes in all packages. Any class can access a public member of another class.

Private: A private access modifier restricts the visibility of the class, variable, method, or constructor to only within the same class. Private members cannot be accessed from outside the class, even from a subclass.

Protected: A protected access modifier makes the class, variable, method, or constructor visible to the same class, subclasses, and classes in the same package. Protected members can be accessed by subclasses or classes in the same package, but not by classes in other packages.

Package-private (default): When no access modifier is specified, the member has package-private access. Package-private members are accessible only within the same package. This means that they cannot be accessed from a class in a different package, even if it is a subclass.

Access modifiers are important for encapsulation and information hiding in object-oriented programming. By controlling the accessibility of members, developers can limit the exposure of implementation details and protect the integrity of their code.

A table that summarizes the accessibility of each access modifier in different scopes in Java:

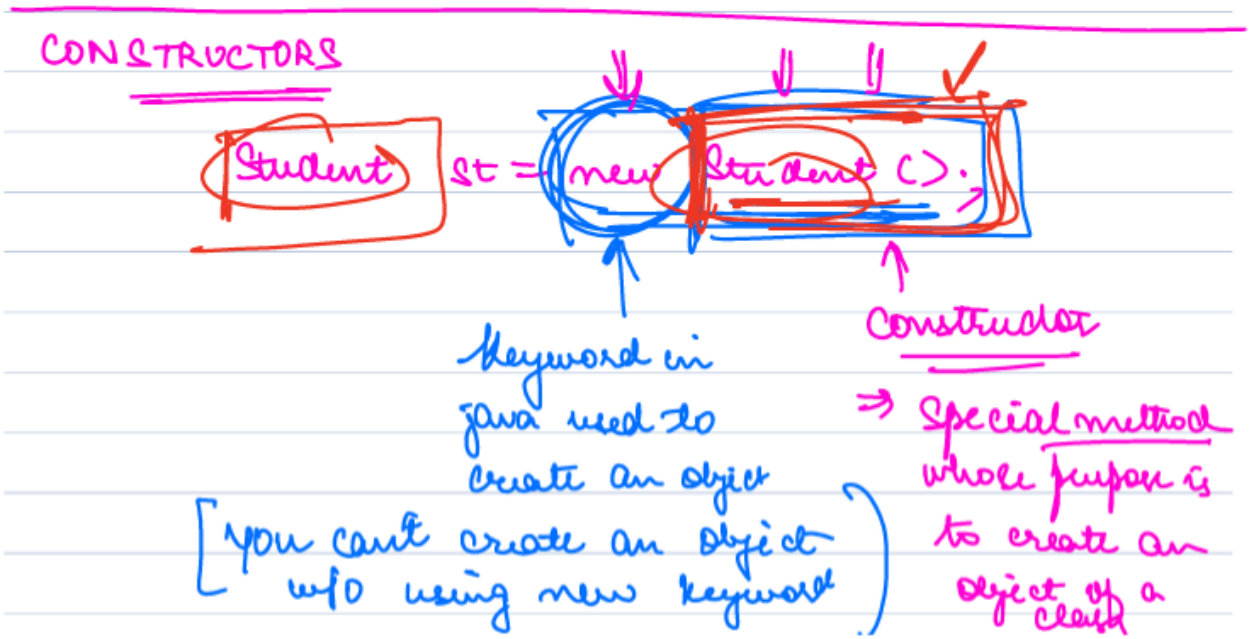
Access Modifier	Class	Package	Subclass	Other packages
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
package-private (default)	Yes	Yes	No	No
private	Yes	No	No	No

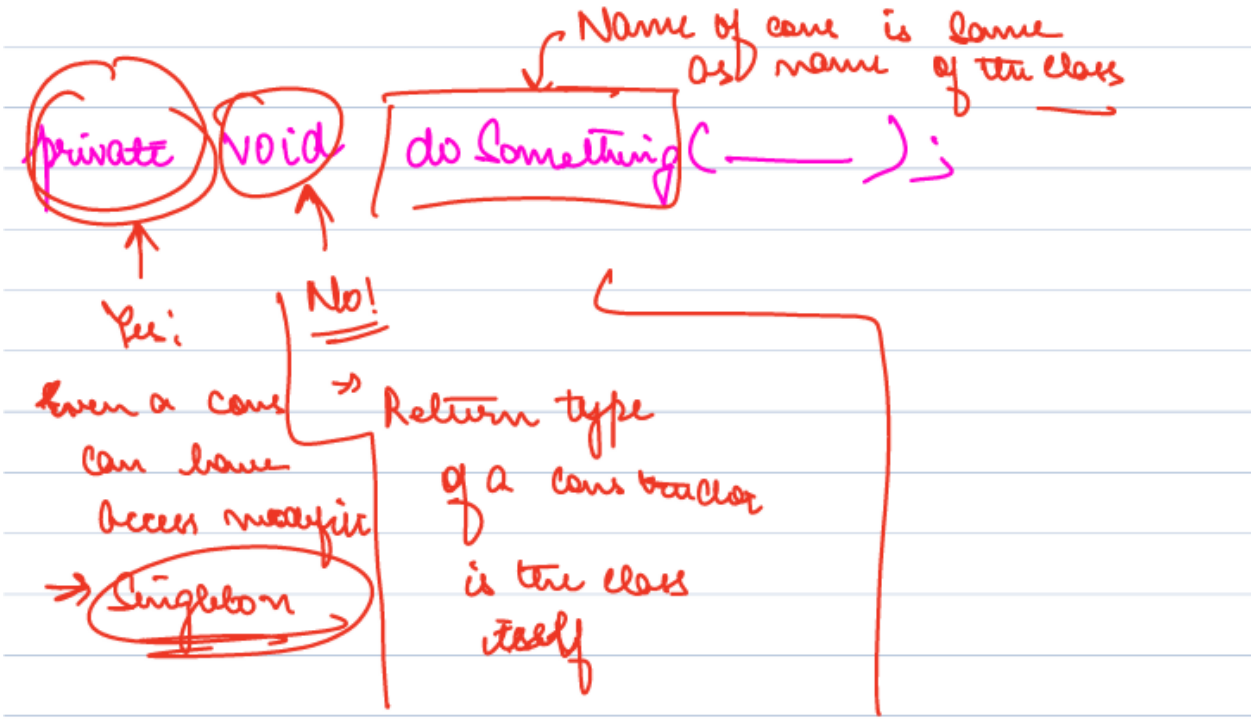
Note that "Yes" means the member is accessible in that scope, and "No" means the member is not accessible in that scope.

Here's a brief explanation of what each scope means:

- Class: Refers to the class where the member is defined.
- Package: Refers to all classes within the same package as the class where the member is defined.
- Subclass: Refers to all subclasses of the class where the member is defined.
- Other packages: Refers to all classes outside of the package where the class where the member is defined is located.

Constructor:





A constructor is a special type of method that is used to create and initialize objects of a class. It has the same name as the class and does not have a return type. The purpose of a constructor is to ensure that an object is properly initialized before it is used.

When an object is created using the "new" keyword, the constructor of that class is called automatically. The constructor initializes the object by setting values for its instance variables, allocating memory for the object, and performing any other necessary initialization tasks.

Constructors can be overloaded, which means that a class can have multiple constructors with different parameter lists. This allows objects to be created with different initial values depending on the needs of the program.

For example, consider a class called "Person" that has a constructor with three parameters: name, age, and gender:

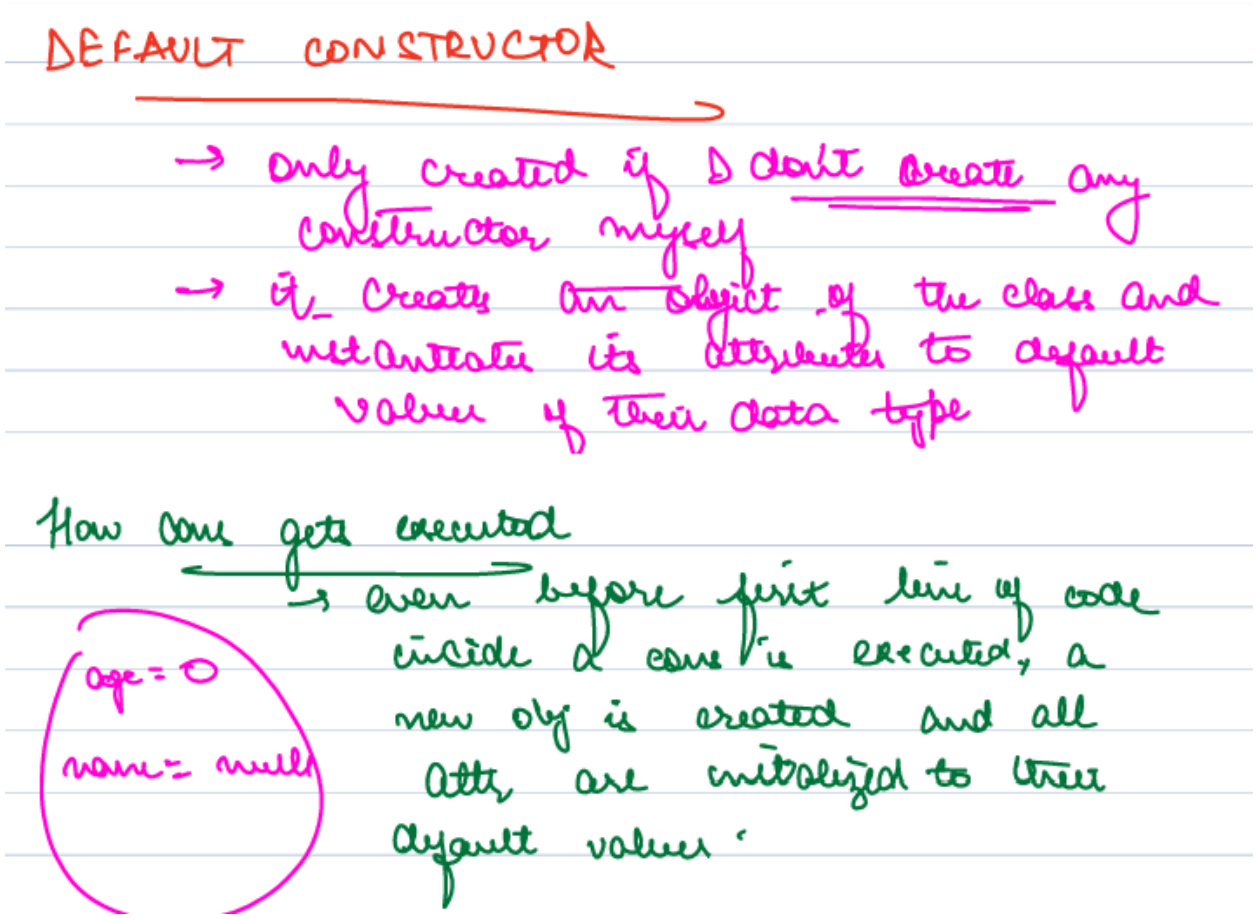
```
public class Person {
    String name;
    int age;
    String gender;

    public Person(String name, int age, String gender) {
        this.name = name;
```

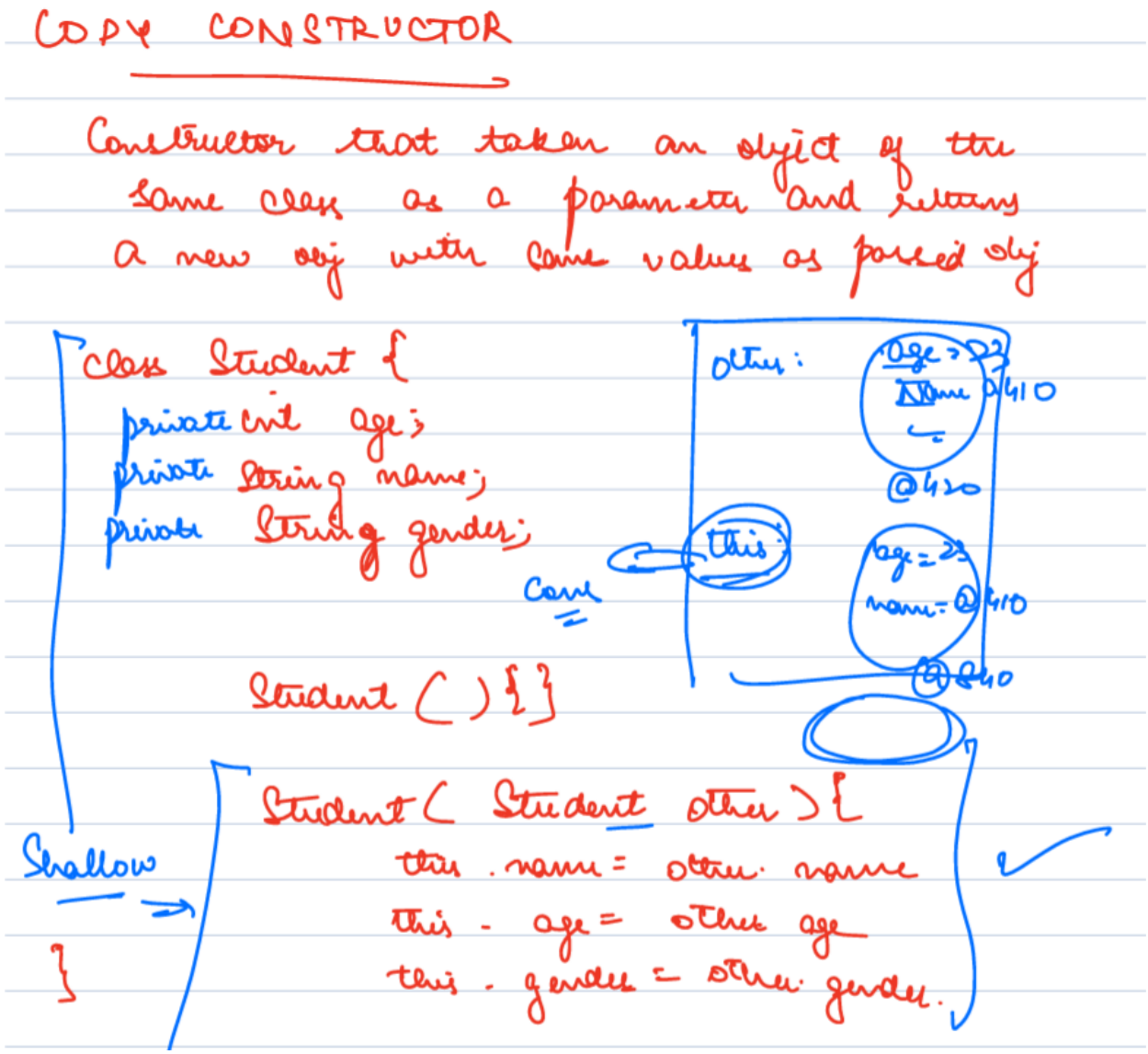
```
        this.age = age;
        this.gender = gender;
    }
}
```

This constructor takes three parameters and initializes the instance variables of the Person object with those values. To create a new Person object, you can call the constructor and pass in the necessary arguments:

```
Person person1 = new Person("John", 30, "Male");
```

This creates a new Person object named "person1" with the name "John", age 30, and gender "Male". The constructor is called automatically when the object is created, and the object is properly initialized before it can be used.





In Java, copy constructors can be used to create new objects that are copies of existing objects. There are two main types of copy constructors: shallow copy constructors and deep copy constructors.

A shallow copy constructor creates a new object that is a copy of the original object but with the same references to objects as the original object. This means that any changes made to the referenced objects will affect both the original object and the copied object. Shallow copy constructors are often used when the referenced objects are immutable or should be shared between the original and copied objects.

Here is an example of a shallow copy constructor in Java:

```
public class Person {
    private String name;
```

```
    private Address address;

    public Person(Person other) {
        this.name = other.name;
        this.address = other.address;
    }

    // getters and setters omitted for brevity
}

public class Address {
    private String street;
    private String city;

    // getters and setters omitted for brevity
}

// Example usage
Address address = new Address("123 Main St", "Anytown");
Person person1 = new Person("John Doe", address);
Person person2 = new Person(person1); // Shallow copy constructor
address.setStreet("456 High St");
System.out.println(person1.getAddress().getStreet()); // Output: 456 High St
System.out.println(person2.getAddress().getStreet()); // Output: 456 High St
```

In this example, a shallow copy constructor is used to create a new "Person" object ("person2") that is a copy of the original "Person" object ("person1"). However, both "person1" and "person2" reference the same "Address" object ("address"). When the "setStreet" method is called on "address", it changes the value of "street" for both "person1" and "person2".

In contrast, a deep copy constructor creates a new object that is a copy of the original object, but with new references to objects. This means that changes made to the referenced objects will only affect the copied object, not the original object. Deep copy constructors are often used when the referenced objects are mutable or should not be shared between the original and copied objects.

Here is an example of a deep copy constructor in Java:

```
public class Person {
    private String name;
    private Address address;

    public Person(Person other) {
        this.name = other.name;
        this.address = new Address(other.address);
    }

    // getters and setters omitted for brevity
}

public class Address {
    private String street;
    private String city;

    public Address(Address other) {
        this.street = other.street;
        this.city = other.city;
    }

    // getters and setters omitted for brevity
}

// Example usage
Address address = new Address("123 Main St", "Anytown");
Person person1 = new Person("John Doe", address);
Person person2 = new Person(person1); // Deep copy constructor
address.setStreet("456 High St");
System.out.println(person1.getAddress().getStreet()); // Output: 456 High St
System.out.println(person2.getAddress().getStreet()); // Output: 123 Main St
```

In this example, a deep copy constructor is used to create a new "Person" object ("person2") that is a copy of the original "Person" object ("person1"). However, the "Address" object is also copied

using a deep copy constructor. This means that "person2" references a new "Address" object that is a copy of the original "Address" object. When the "setStreet" method is called on "address", it changes the value of "street" for "person1", but not for "person2.

Inheritance:

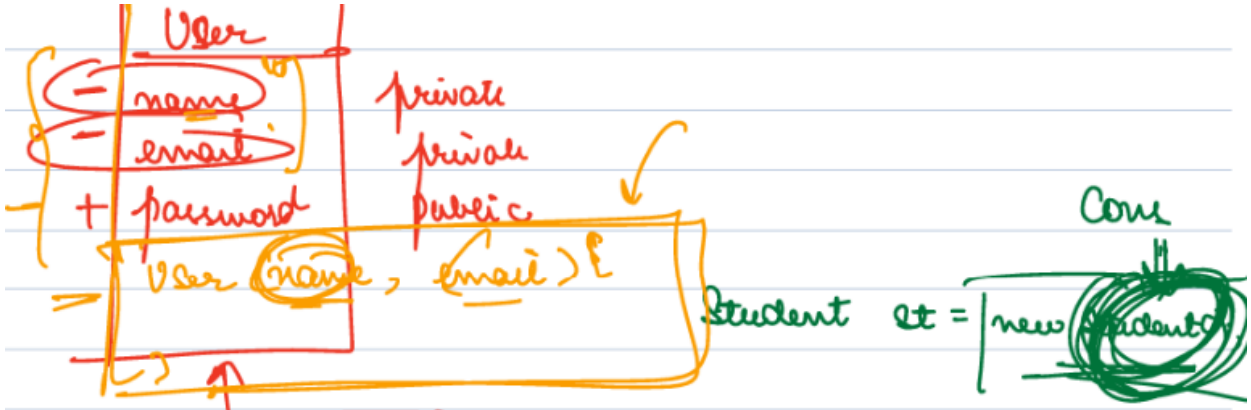
In Java, inheritance is a mechanism that allows one class to inherit the properties and behaviors of another class. The class that inherits properties and behaviors is called the subclass, while the class that provides these properties and behaviors is called the superclass.

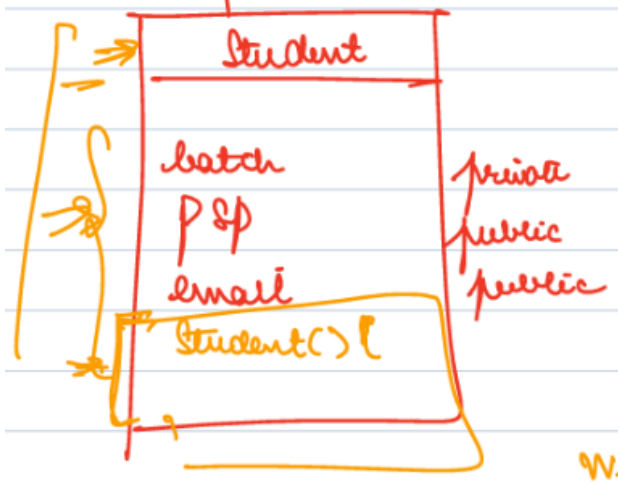
To implement inheritance in Java, you can use the "extends" keyword to create a subclass that inherits from a superclass. The syntax for creating a subclass is as follows:

```
class Subclass extends Superclass {
    // subclass members
}
```

The subclass can access the public and protected members of the superclass, including fields, methods, and constructors. **However, it cannot access private members of the superclass but private members will be present in the subclass.**

Inheritance allows for code reusability and can help to organize complex code into a hierarchy of related classes. However, it can also lead to a tightly coupled design if used improperly, so it's essential to carefully consider the relationship between the classes before implementing inheritance.





(1) even the child class has a const'

(2) all of the attr (incl the attributes of parent) must be initialized

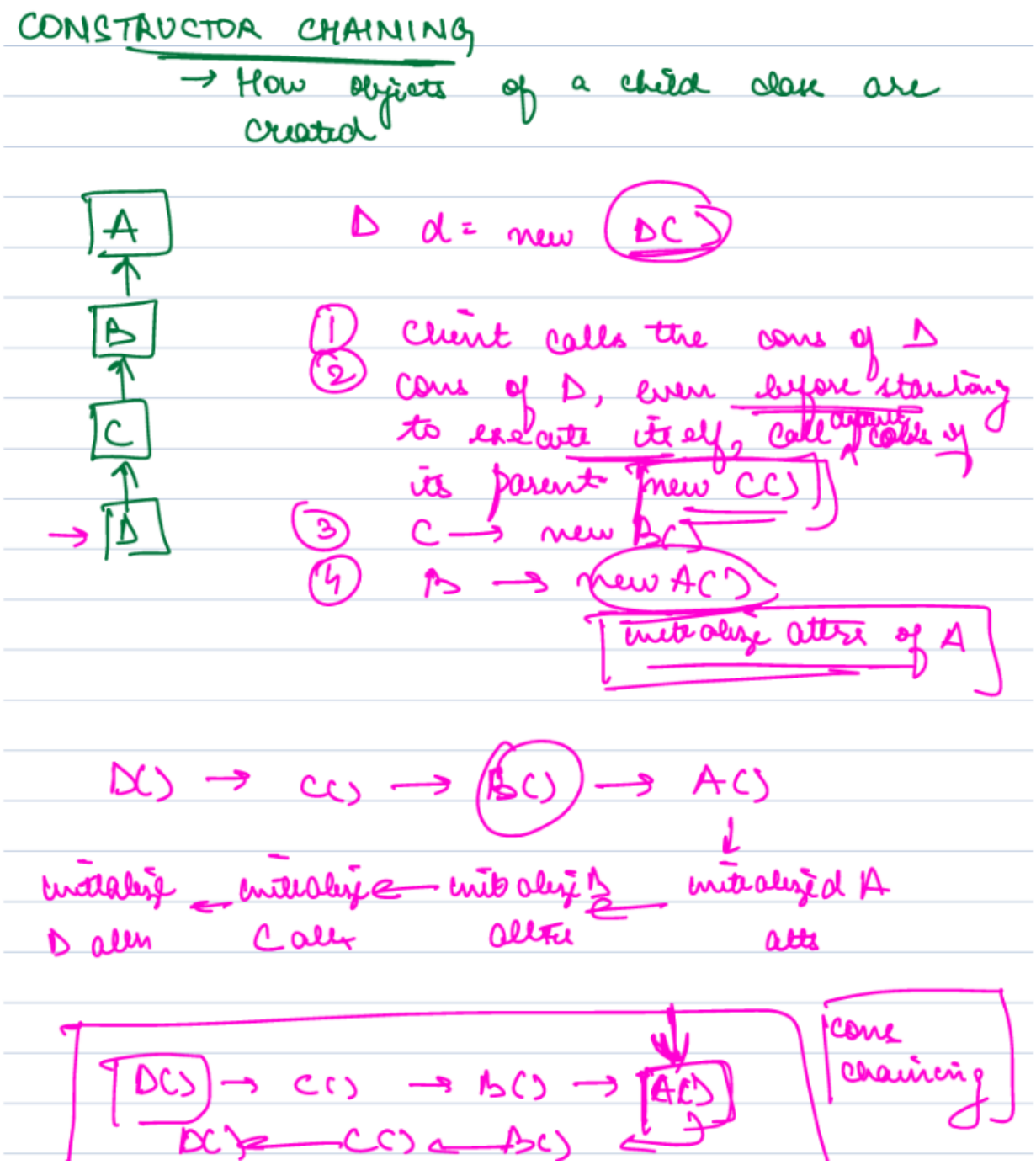
Who will initialize all attrs.

a.) Can the cons of child initialize all attrs of parent → NO ∵ it may not be able to access its private attrs

If all are public → Yes, cons of child can do that

But is that good → NO

Constructor chaining:



Constructor chaining is a mechanism in Java that allows one constructor to call another constructor of the same class or of its superclass. This is useful when you want to avoid duplicating code in multiple constructors and simplify the construction process.

Constructor chaining is achieved by using the "this" keyword to call another constructor within the same class, or the "super" keyword to call a constructor of the superclass. When a constructor is called using either "this" or "super", it must be the first statement in the constructor body.

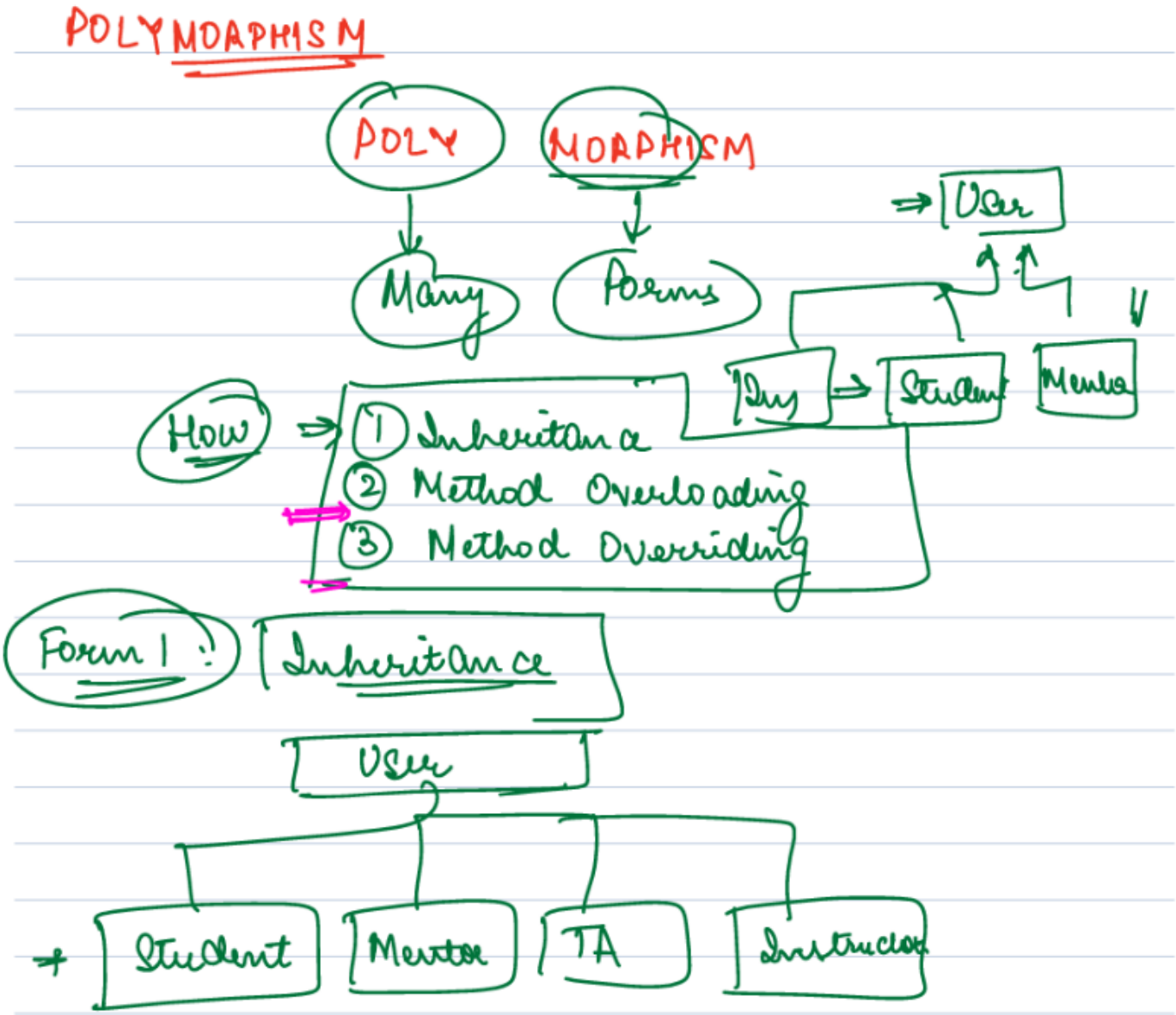
Here is an example of constructor chaining using the "this" keyword:

```
public class Person {
    private String name;
    private int age;

    public Person() {
        this("John Doe", 0);
    }

    public Person(String name) {
        this(name, 0);
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

In this example, the first constructor with no arguments calls the second constructor with a default age of 0 using the "this" keyword. Similarly, the second constructor with one argument calls the third constructor with a default age of 0 using the "this" keyword. Finally, the third constructor initializes the name and age attributes of the Person object.

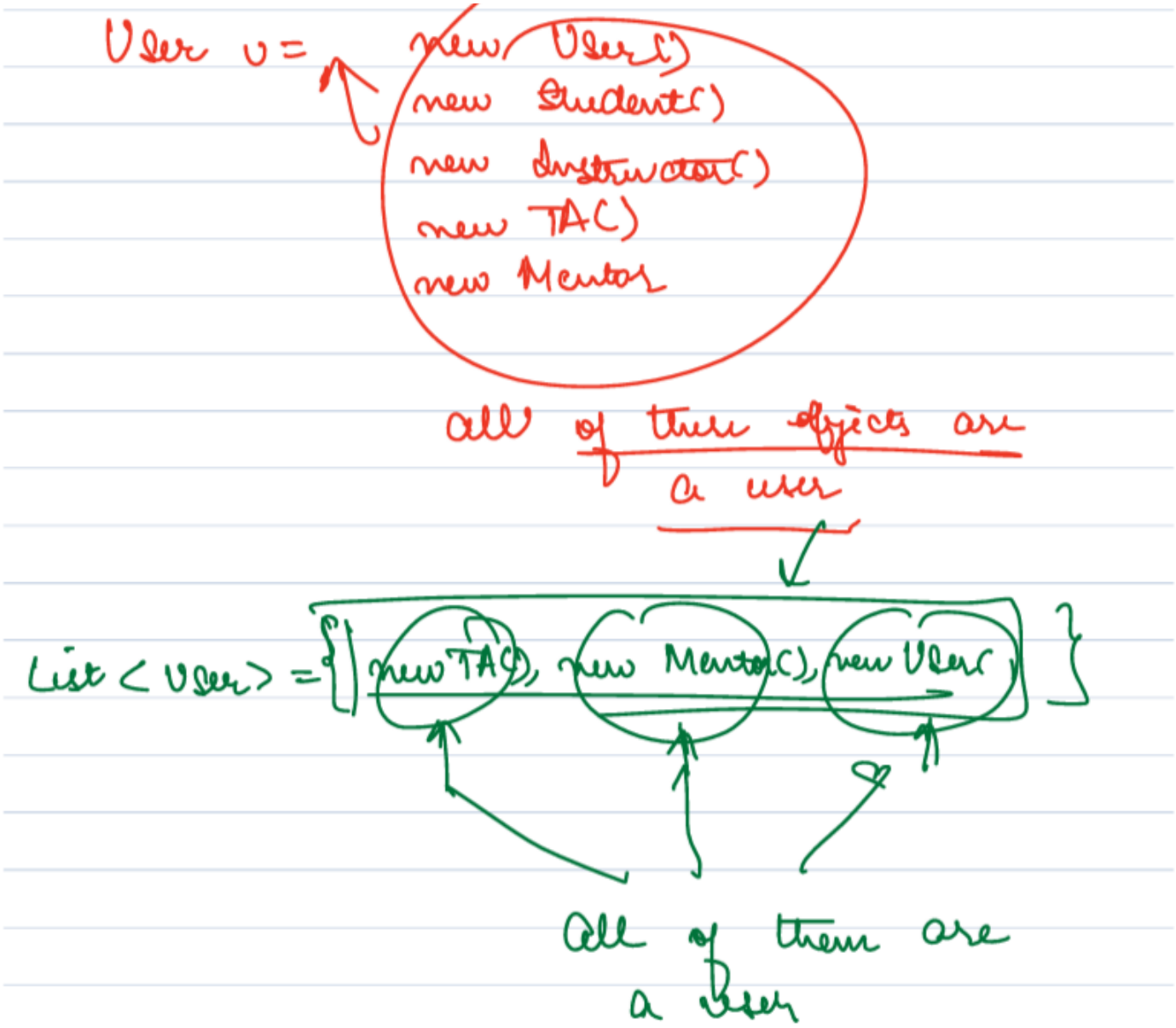
Constructor chaining using the "super" keyword works similarly, except that it calls a constructor of the superclass instead of the same class. This can be useful when you want to initialize the attributes of the superclass before initializing the attributes of the subclass.

Polymorphism:



⇒ A variable of parent class can store objects of child classes as well. because all properties of parent are going to be present in the child.

⇒ But you will be allowed to call only those methods that are declared in the data type (not in child)



Polymorphism is a concept in object-oriented programming that allows objects of different classes to be treated as if they were objects of a common superclass. It is the ability of an object to take on many forms, meaning that a single object can be used to represent different things in different contexts.

There are two main types of polymorphism: **compile-time** (or **static**) and **runtime** (or **dynamic**) polymorphism. Compile-time polymorphism is achieved through **method overloading**, which allows multiple methods with the same name but different parameters to coexist in a class. Runtime polymorphism is achieved through **method overriding**, which allows a subclass to provide its own implementation of a method that is already defined in its superclass.

Compile-time (or static) Polymorphism / Method Overloading:

METHOD OVERLOADING

→ compile time polymorphism

→ A class having multiple methods with the same name.
(with diff set of params)

2 type of polymorphism:
1. compile time poly
2. runtime poly

Method Overloading : More than one method with same name but diff signature.

Method Signature

```
void hello (int i) { }
```

Signature: hello (int)

1. Return type is not a part of method signature
2. Param. name is not a part of signature

Overloading Example:

```
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }

    public double add(double x, double y) {
        return x + y;
    }

    public int add(int x, int y, int z) {
        return x + y + z;
    }
}


public class OverloadingExample {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        int result1 = calculator.add(1, 2);
        double result2 = calculator.add(1.5, 2.5);
        int result3 = calculator.add(1, 2, 3);

        System.out.println(result1);
        System.out.println(result2);
        System.out.println(result3);
    }
}
```

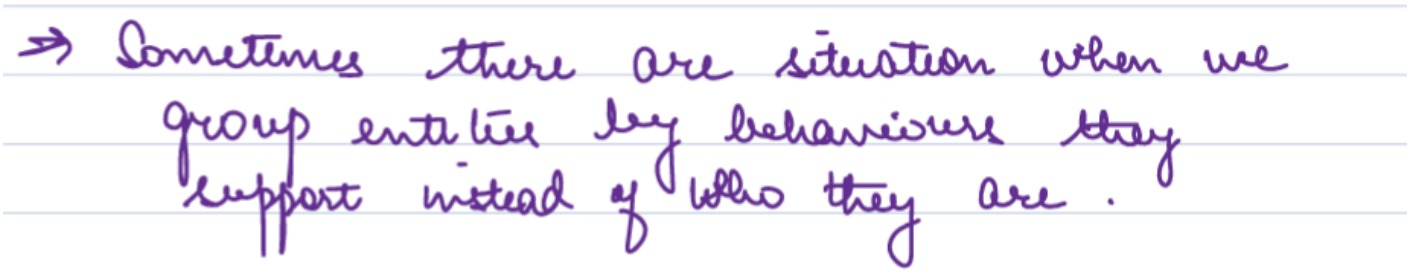
Runtime (or dynamic) polymorphism / Method Overriding:

Overriding ⇒ A method with same signature and same return type present in the parent class as well as in the child class. Method of child class overrides the method of parent class.

Overriding Example:

```
public class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound");
    }
}


public class Dog extends Animal {
    public void makeSound() {
        System.out.println("The dog barks");
    }
}

public class Cat extends Animal {
    public void makeSound() {
        System.out.println("The cat meows");
    }
}
```

Interface:



An interface is a collection of abstract methods and constants (i.e., fields with static and final modifiers) that define a contract between a class and the outside world. It specifies a set of methods that a class that implements the interface must implement, but it does not provide any implementation for those methods. The implementation is left to the classes that implement the interface.

```
public interface Shape {
    double getArea();
    double getPerimeter();
}

public class Rectangle implements Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
```

```
        this.length = length;
        this.width = width;
    }

    public double getArea() {
        return length * width;
    }

    public double getPerimeter() {
        return 2 * (length + width);
    }
}

public class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }

    public double getPerimeter() {
        return 2 * Math.PI * radius;
    }
}

public class Square implements Shape {
    private double side;

    public Square(double side) {
        this.side = side;
    }

    public double getArea() {
        return side * side;
    }

    public double getPerimeter() {
        return 4 * side;
    }
}
```

```
public class InterfaceExample {
    public static void main(String[] args) {
        Shape rectangle = new Rectangle(3, 4);
        Shape circle = new Circle(2);
        Shape square = new Square(5);

        System.out.println("Rectangle area: " + rectangle.getArea());
        System.out.println("Rectangle perimeter: " +
rectangle.getPerimeter());

        System.out.println("Circle area: " + circle.getArea());
        System.out.println("Circle perimeter: " + circle.getPerimeter());

        System.out.println("Square area: " + square.getArea());
        System.out.println("Square perimeter: " + square.getPerimeter());
    }
}
```

Abstract Classes

↳ to rep an entity

```
abstract Animal
  - name
  - age
  - gender
  - makeSound() { }
abstract play()
```

```
Animal a = ____
  → a.play()
```

⇒ As the entity is not completely known, it shouldn't be allowed to create object of that

→ For such an entity, the methods that are not defined are declared abstract

→ As well as, the class is also declared abstract

→ The child classes of that class must implement all abstract methods or themselves should be declared abstract