04 Inheritance And Polymorphism

28 May 2024 21:02

Agenda :

1. **Inheritance**
2. **Constructor Chaining**
3. **Polymorphism**
4. **Method Overloading**
5. **Method Overriding**

1. Inheritance

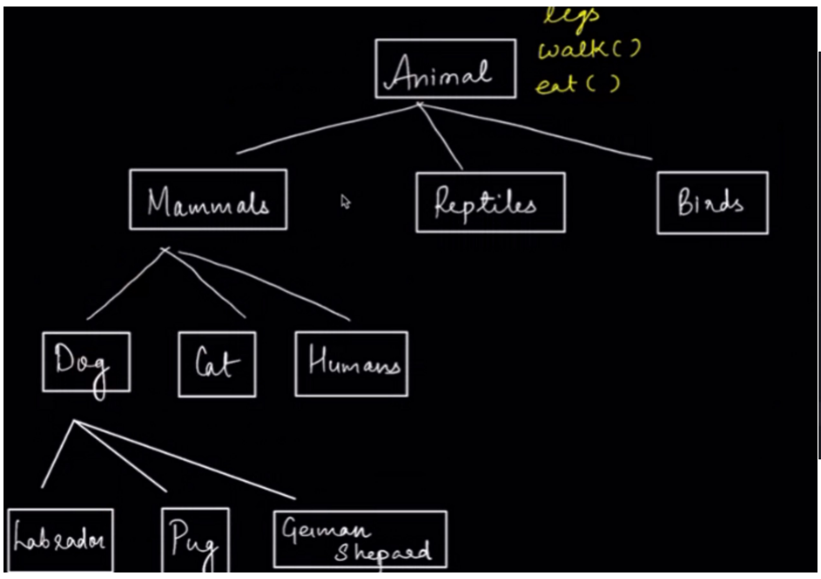
- A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass.* This mechanism is known as **Inheritance.**

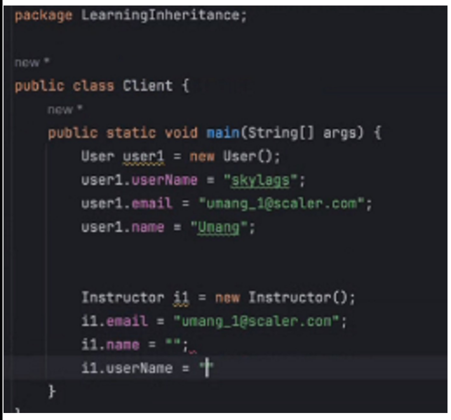
- A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

What we can do in a subclass ?

- **A subclass inherits all of the *public* and *protected* members of its parent**, no matter what package the subclass is in.

- **If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent.**
  - The inherited fields can be used directly.
  - You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
  - You can declare new fields in the subclass that are not in the superclass.
  - The inherited methods can be used directly as they are.
  - You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
  - You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
  - You can declare new methods in the subclass that are not in the superclass.
  - You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.
  - A nested class has access to all the private members of its enclosing class—both fields and methods.
  - **Final** methods cannot be redefined in child class. (cannot be overridden)



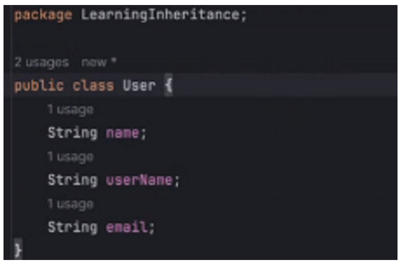


```
package LearningInheritance;

new *
public class Client {
    new *
    public static void main(String[] args) {
        User user1 = new User();
        user1.userName = "skylags";
        user1.email = "umang_1@scaler.com";
        user1.name = "Umang";


        Instructor i1 = new Instructor();
        i1.email = "umang_1@scaler.com";
        i1.name = "";
        i1.userName = "
    }
```

- **Animal** is the highest level of abstraction.
- **Inheritance** is this kind of representation of hierarchy among classes.
- With this kind of oops feature we don't need to define common features among all classes like walk(), eat(), etc. So inheritance allows us to share attributes and behaviour.
- **Animal** is a parent class/ super class and **Mammals** is a child class / sub class. All the properties and behaviours of parent can be shared with child.
- A child class inherits all the members of the parent class and may add their own members.



```
package LearningInheritance;

2 usages  new *
public class User {
    1 usage
    String name;
    1 usage
    String userName;
    1 usage
    String email;
}
```

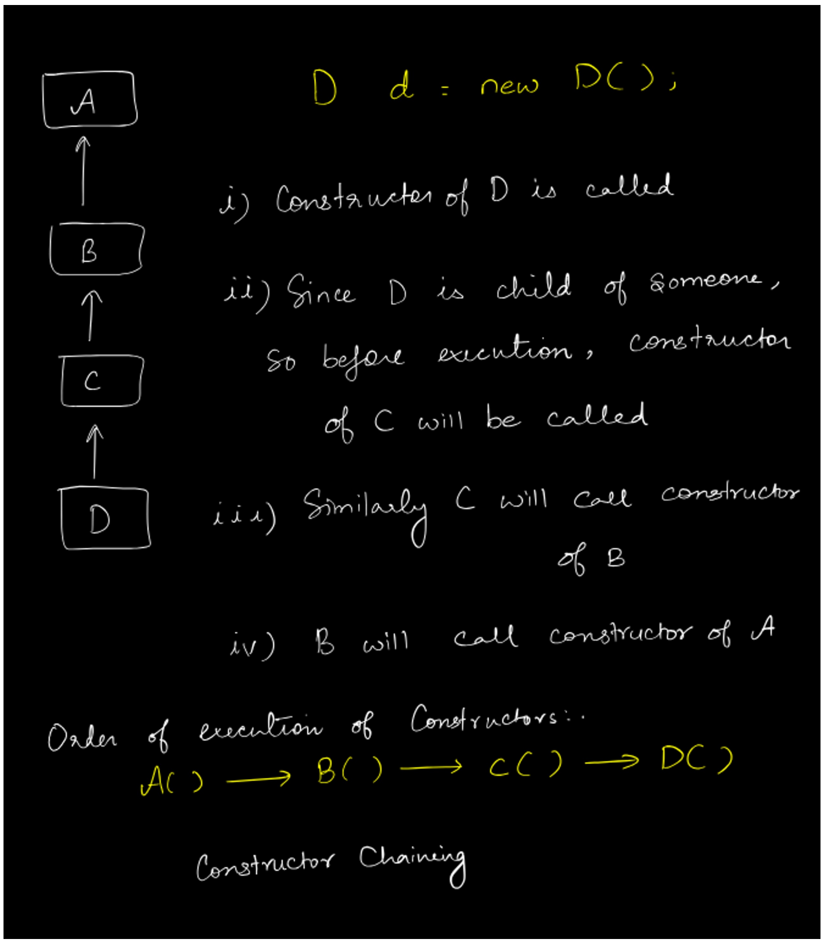
```
Public class Instructor extends User{
        String batchName;
        Double avgRating;

        Instructor(){
                System.out.println("This is a constructor of Instructor class!!")
        }
}

o/p -> User's constructor is called
         This is a constructor of Instructor class!!
```

- **Constructor** of **User** class should be one to initialize User class.
- When we create a constructor of Instructor, we need to call the constructor of User class (Parent class) first. This is called **Constructor chaining.**

2. Constructor Chaining


A
B
C
D
D d = new DC);
i) Constructor of D is called
ii) Since D is child of someone, So before execution, constructor of C will be called
iii) Similarly C will call constructor of B
iv) B will call constructor of A
Order of execution of Constructors :-
A( ) ⟶ B( ) ⟶ C( ) ⟶ D( )
Constructor Chaining

Sequence of constructor call :

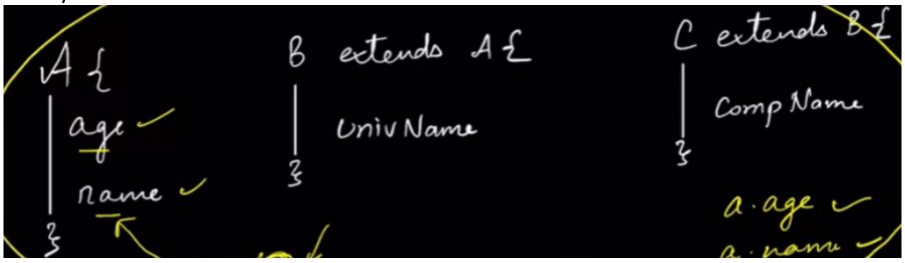
A() -> B() -> C() -> D()

- If there is a default constructors in the parents. Then from top to bottom constructor is called in sequence, this is called as constructor chaining.
- If there is no default constructor in the parent and other parameterized constructor is present then chain will be broken, as no default constructor is present.
  In this case we need to add a **super("....params")** in order to maintain the chaining. Number of parameters inside constructor calls the respective parent constructor.

https://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html - object class in java.

3. Polymorphism

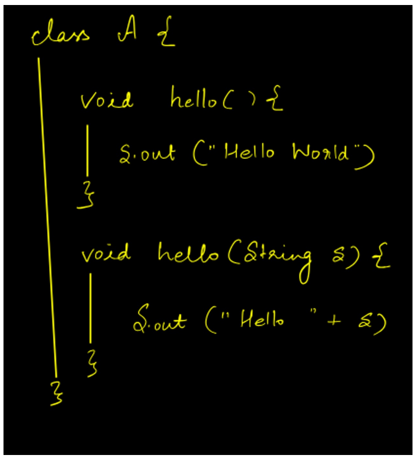
- Polymorphism is a way in which single behaviour can be represented in the multiple forms. Thus handling different use case for same functionality.



- We can put and object of child that takes parent class.
  Ex : A a = new B();
  But vice versa is not correct.
- Objects are created at runtime. **So A a = new C(); is not possible** as C does not directly inherit A, we will get runtime error.

- Types of Polymorphism :
  - Run time Polymorphism / Method Overriding
  - Compile time Polymorphism / Method Overloading

4. Method Overloading

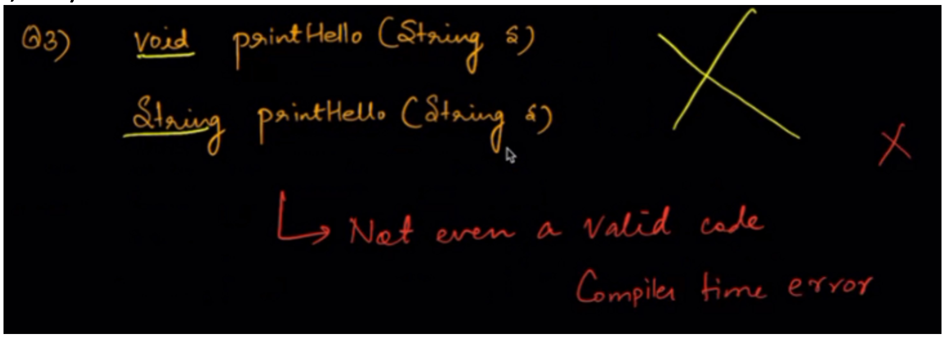


```
class A {
    void hello() {
        S.out("Hello World")
    }
    void hello(String s) {
        S.out("Hello " + s)
    }
}
```

- Different methods with **same signature, but different set of parameters. Return type doesn't matter to the compiler.** This scenario is called as Method Overloading.

- Final form that is going to execute is mapped by compiler. So called **compile time polymorphism.**
  ex : hello(), hello(str) ...

- **Order of parameters also matter.** Below one is also sample of overloading.
  Int c(int x, String c)
  int c(String c, int x)



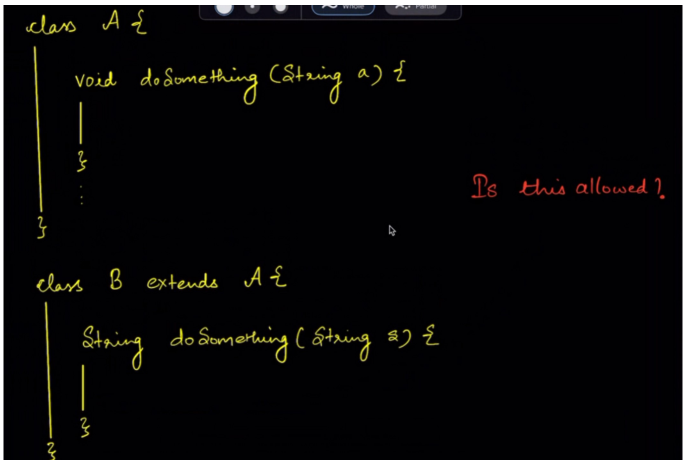
```
Q3) void printHello(String s)
    String printHello(String s)
```

Not even a valid code

Compiler time error

5. Method Overriding

- When parent and child have **same name, signature and return type is also same**. This scenerio is known as method overriding.
- **Method of which class to called is determined at run time on the basis of object mapping / object creation.** So it is called as run time polymorphism.



```
class A {
    void doSomething(String a) {
    }
}

class B extends A {
    String doSomething(String s) {
    }
}
```

Is this allowed?

- Above is not allowed as Methods have same name and parameters. **But return type is different.**

06 June 2024 16:14