第07天 面向对象

今日内容介绍

* 方法的概述
* 方法的练习
* 方法的重载
* 方法参数类型详解

# 继承

## 继承的概述

### 继承概述

多个类中存在相同的属性和行为时，将这些内容抽取到单独一个类中，那么多个类无需在定义这些属性和行为，只要继承那个类即可

单独的这个类称为父类，基类或者叫超类，多个类可以称为子类或者派生类

有了继承以后，我们定义一个类的时候，可以在一个已经存在的类的基础上，还可以定义自己的新成员

### 实现继承的方式

通过extends关键字可以实现类与类的继承

格式：

public class 子类名 extends 父类名{}

### 案例代码一

package com.itheima\_02;

/\*

\* 标准人类

\*

\* 继承的格式：

\* public class 子类名 extends 父类名 {}

\*

\* 继承的好处：

\* A:提高了代码的复用性

\* B:提高了代码的维护性

\* C:让类与类之间产生了关系，是多态的前提

\*

\* 继承的弊端：

\* 让类与类之间产生了关系，也就让类的耦合性增强了。

\*

\* 开发原则：高内聚，低耦合。

\* 内聚：就是自己完成某件事情的能力

\* 耦合：类与类的关系

\*/

public class Person {

private String name;

private int age;

public Person() {}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

package com.itheima\_02;

public class Student extends Person {

public void study() {

System.out.println("学生要好好学习");

}

}

package com.itheima\_02;

public class Teacher extends Person {

public void teach() {

System.out.println("老师要好好讲课");

}

}

## 继承的好处和弊端

### 继承的好处

* 提高了代码的复用性
  + 多个类相同的成员可以放到同一个类中
* 提高了代码的维护性
  + 如果功能的代码需要修改，修改一处即可
* 让类与类之间产生了关系，是多态的前提

### 继承的弊端

* 好处的第三点同时也是继承的弊端
  + 类与类之间产生了关系，让类的耦合性增强了
  + 设计原则：高内聚低耦合

### 示例代码

package com.itheima\_02;

/\*

\* 标准人类

\*

\* 继承的格式：

\* public class 子类名 extends 父类名 {}

\*

\* 继承的好处：

\* A:提高了代码的复用性

\* B:提高了代码的维护性

\* C:让类与类之间产生了关系，是多态的前提

\*

\* 继承的弊端：

\* 让类与类之间产生了关系，也就让类的耦合性增强了。

\*

\* 开发原则：高内聚，低耦合。

\* 内聚：就是自己完成某件事情的能力

\* 耦合：类与类的关系

\*/

public class Person {

private String name;

private int age;

public Person() {}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

## Java 中继承的特点

### Java中单继承以及多级继承

* Java中只支持单继承，不支持多继承
  + 一个类只能有一个父类，不可以有多个父类
  + public class Son extends Father{} // ok
  + public class Son extends Father,GrandFather // Error
* Java中类支持多层继承(继承体系)
  + Public class GrandFather{}
  + Public class Father extends GrandFather{}
  + Public class Son extends Father{}

### 示例代码

package com.itheima\_03;

public class Father extends GrandFather {

public void fatherSay() {

System.out.println("爸爸都是从儿子走过来的");

}

}

package com.itheima\_03;

public class GrandFather {

public void grandFatherSay() {

System.out.println("爷爷都是从孙子熬过来的");

}

}

package com.itheima\_03;

/\*

public class Son extends GrandFather {

public static void main(String[] args) {

Son s = new Son();

s.grandFatherSay();

}

}

\*/

/\*

public class Son extends Father {

public static void main(String[] args) {

Son s = new Son();

s.fatherSay();

}

}

\*/

//Java中类只支持单继承，不支持多继承

/\*

public class Son extends Father,GrandFather {

public static void main(String[] args) {

Son s = new Son();

s.fatherSay();

}

}

\*/

//Java中类支持多层继承

public class Son extends Father {

public static void main(String[] args) {

Son s = new Son();

s.fatherSay();

s.grandFatherSay();

}

}

## Java继承中成员变量的特点

### Java继承中成员变量的特点

* 成员变量名称不一样，使用的时候非常简单
* 成员变量名称一样的情况：
  + 在子类中访问变量：（就近原则）
    - 在方法的局部范围找，如果有就使用
    - 在子类的成员范围找，如果有就使用
    - 在负累的成员范围找，如果有就使用
    - 如果还找不到 就报错

### 示例代码

package com.itheima\_01;

public class Father {

//为了演示案例的方便，这里我们使用public修饰了成员变量，实际开发中，用的是private

//年龄

public int age = 45;

}

package com.itheima\_01;

/\*

\* Java继承中成员变量的特点：

\* A:成员变量名称不一样，使用的时候非常简单。

\* B:成员变量名称一样的情况

\* 在子类方法中访问变量：

\* a:在方法的局部范围找，如果有就使用

\* b:在子类的成员范围找，如果有就使用

\* c:在父类的成员范围找，如果有就使用

\* d:如果还找不到，就报错

\* 就近原则。

\*/

public class Son extends Father {

//身高

public int height = 170;

//年龄

public int age = 20;

public void show() {

System.out.println(height);

System.out.println(age);

}

public void printAge() {

int age = 10;

System.out.println(age);

}

}

package com.itheima\_01;

public class ExtendsTest {

public static void main(String[] args) {

Son s = new Son();

//s.show();

s.printAge();

}

}

# super关键字以及继承中的方法重写

## super关键字的概述和使用

### super关键字的概述

* super的用法和this很像
  + this代表本类对象的引用
  + super代表父类存储空间的标识（可以理解为父类对象引用）
* 用法（this和super均可如下使用）
  + 访问成员变量
    - this.成员变量 super.成员变量
    - 访问构造方法
      * this(…) super(…)
    - 访问成员方法
      * this.成员方法() super.成员方法()

### 示例代码

package com.itheima\_02;

public class Father {

public int age = 45;

}

package com.itheima\_02;

/\*

\* super和this的用法很像：

\* this:代表本类对象的引用

\* super:代表父类的存储空间(可以理解为代表父类对象的引用)

\*

\* 用法：

\* 访问成员变量：

\* this.成员变量

\* super.成员变量

\* 访问构造方法：

\* this(...)

\* super(...)

\* 访问成员方法：

\* this.成员方法()

\* super.成员方法()

\*/

public class Son extends Father {

public int age = 20;

public void printAge() {

int age = 10;

System.out.println(age);

//我要访问成员范围的age?

System.out.println(this.age);

//我要访问父类成员范围的age?

System.out.println(super.age);

}

}

package com.itheima\_02;

public class ExtendsTest {

public static void main(String[] args) {

Son s = new Son();

s.printAge();

}

}

## Java继承中构造方法的特点

* 子类中所有的构造方法默认都会访问父类中空参数的构造方法
* 为什么呢？
  + 因为子类会继承父类中的数据，可能还会使用父类的数据，所以，子类初始化之前，一定要先完成父类数据的初始化
  + 每一个构造方法的第一条默认语句都是super
* 如果父类中没有构造方法，该怎么办呢？
  + 在父类中加一个无参的构造方法
  + 通过使用super关键字去显示的调用父类的带参构造方法
  + 通过这里我们发现第一种解决方案最简单，所以，建议我们自定义类的时候永远自己给出无参构造方法

### 示例代码

package com.itheima\_03;

public class Father {

/\*

public Father() {

System.out.println("Father无参构造方法");

}

\*/

public Father(String name) {

System.out.println("Father带参构造方法");

System.out.println(name);

}

}

package com.itheima\_03;

/\*

\* Java继承中构造方法的访问特点：

\* A:子类构造方法执行前都会先执行父类无参构造方法

\* B:为什么呢?

\* 因为子类继承父类，会继承父类的非私有成员。

\* 而子类在初始化的时候，可能会使用父类的数据，如果父类数据没有先初始化，

\* 子类就不能使用这些数据，所以，在子类初始化之前，一定要先完成父类数据的初始化。

\*

\* 注意：在子类的构造方法中，默认第一行有一条语句：super()

\*

\* 问题：假如父类中没有无参构造方法，怎么办呢?

\* A:在父类中添加一个无参构造方法。

\* B:可以通过super去访问父类的带参构造方法。

\* 建议使用第一种解决方案，其实就是要求我们写代码的时候，每次都手动的给出无参构造方法。

\*/

public class Son extends Father {

public Son() {

//super();

super("林青霞");

System.out.println("Son无参构造方法");

}

public Son(String name) {

//super();

super("林青霞");

System.out.println("Son带参构造方法");

System.out.println(name);

}

}

package com.itheima\_03;

public class ExtendsTest {

public static void main(String[] args) {

Son s = new Son();

System.out.println("---------");

Son s2 = new Son("林青霞");

}

}

## Java继承中成员方法的特点

### Java 继承中成员方法的特点

* 通过子类对象去访问一个方法
  + 首先在子类中找
  + 然后在父类中找
  + 如果还是没有就会报错

### 示例代码

package com.itheima\_04;

public class Father {

public void show() {

System.out.println("Father show");

}

}

package com.itheima\_04;

/\*

\* Java继承中成员方法的访问特点：

\* A:子类中方法和父类中方法的声明不一样，这个太简单

\* B:子类中方法和父类中方法的声明一样，调用的到底是谁的呢?

\* 执行的是子类中的方法。

\*

\* 通过子类对象调用方法：

\* a:在子类中找，有就使用

\* b:在父类中找，有就使用

\* c:如果没有就报错

\*/

public class Son extends Father {

public void method() {

System.out.println("Son method");

}

public void show() {

System.out.println("Son show");

}

}

package com.itheima\_04;

public class ExtendsTest {

public static void main(String[] args) {

Son s = new Son();

s.method();

s.show();

//s.function();

}

}

## 方法重写的概述和使用

### 方法重写的概述

* 方法重写：子类中出现了和父类中一摸一样的方法声明

### 方法重写的应用

* 当子类需要父类的功能，而功能主体子类有自己特有的内容时，可以重写中的方法，这样重写父类中的方法，这样，即沿袭了父类的功能，又定义了子类特有的内容

### 方法重写的注意事项

· 注解

* + - @Override
    - 表明该方法的重写父类的方法
* 方法重写的注意事项
  + - 父类中私有方法不能被重写
    - 子类重写父类方法时，访问权限不能更低
    - 子类重写父类方法时，建议访问权限一摸一样

### 示例代码

package com.itheima\_01;

public class Phone {

public void call(String name) {

System.out.println("给"+name+"打电话");

}

}

package com.itheima\_01;

public class NewPhone extends Phone {

public void call(String name) {

System.out.println("开启视频功能");

//System.out.println("给"+name+"打电话");

super.call(name);

}

}

package com.itheima\_01;

/\*

\* 方法重写：子类中出现了和父类中一模一样的方法声明的情况。

\*

\* 方法重写的应用：

\* 当子类需要父类的功能，而功能主体子类又有自己的特有内容的时候，就考虑使用方法重写，

\* 这样即保证了父类的功能，还添加了子类的特有内容。

\*/

public class PhoneTest {

public static void main(String[] args) {

Phone p = new Phone();

p.call("林青霞");

System.out.println("-----------");

NewPhone np = new NewPhone();

np.call("林青霞");

}

}

# 多态

## 多态的概述和代码体现

* 多态概述
  + 某一个事物，在不同时刻表现出来的不同状态。
* 举例
  + 猫可以是猫的类型。猫 m = new 猫();
  + 同时猫也是动物的一种，也可以把猫称为动物
    - 动物 d = new 猫();
    - 水在不同时刻的状态
* 多态的前提和体现
  + 有继承关系
  + 有方法重写
  + 有父类引用指向子类对象

### 案例代码

package com.itheima\_01;

/\*

\* 多态：同一个对象，在不同时刻体现出来的不同状态。

\* 举例：

\* 猫：猫是猫，猫是动物。

\* 水：液体，固体，气体。

\*

\* Java中多态的前提：

\* A:有继承关系

\* B:有方法重写

\* C:有父类引用指向子类对象

\* Fu f = new Fu();

\* Zi z = new Zi();

\*

\* Fu f = new Zi();

\*/

public class DuoTaiDemo {

public static void main(String[] args) {

//有父类引用指向子类对象

Animal a = new Cat();

}

}

## 多态中成员的访问特点

### 多态中成员访问特点

·成员变量访问特点

* + - 编译看左边，运行看左边

·成员方法访问特点

* + - 编译看左边，运行在左

### 示例代码

package com.itheima\_02;

/\*

\* 多态中成员的访问特点：

\* A:成员变量

\* 编译看左边，执行看左边。

\* B:成员方法

\* 编译看左边，执行看右边。

\*

\* 为什么成员变量和成员方法的访问不一样呢?

\* 因为成员方法有重写，而变量没有。

\*/

public class DuoTaiDemo {

public static void main(String[] args) {

//多态

Animal a = new Cat();

System.out.println(a.age);

//System.out.println(a.weight);

a.eat();

//a.playGame();

}

}

package com.itheima\_02;

public class Cat extends Animal {

public int age = 20;

public int weight = 10;

public void eat() {

System.out.println("猫吃鱼");

}

public void playGame() {

System.out.println("猫捉迷藏");

}

}

package com.itheima\_02;

public class Animal {

public int age = 40;

public void eat() {

System.out.println("吃东西");

}

}

## 多态的好处和弊端

* 多态的好处
  + - 提高了程序的扩展性
* 多态的弊端
  + - 不能访问子类特有功能
    - 那么如何访问子类的特有功能呢？
      * 通过多态中的转型

示例代码

package com.itheima\_03;

/\*

\* 多态的好处：提高了程序的扩展性

\* 具体体现：定义方法的时候，使用父类型作为参数，将来在使用的时候，使用具体的子类型参与操作。

\* 多态的弊端：不能使用子类的特有功能

\*/

public class DuoTaiDemo {

public static void main(String[] args) {

AnimalOperator ao = new AnimalOperator();

Cat c = new Cat();

ao.useAnimal(c);

Dog d = new Dog();

ao.useAnimal(d);

Pig p = new Pig();

ao.useAnimal(p);

}

}

package com.itheima\_03;

public class AnimalOperator {

/\*

public void useAnimal(Cat c) { //Cat c = new Cat();

c.eat();

}

public void useAnimal(Dog d) { //Dog d = new Dog();

d.eat();

}

\*/

public void useAnimal(Animal a) { //Animal a = new Cat();

a.eat();

//a.lookDoor();

}

}

package com.itheima\_03;

public class Cat extends Animal {

public void eat() {

System.out.println("猫吃鱼");

}

}

package com.itheima\_03;

public class Dog extends Animal {

public void eat() {

System.out.println("狗吃骨头");

}

public void lookDoor() {

System.out.println("狗看门");

}

}

package com.itheima\_03;

public class Pig extends Animal {

public void eat() {

System.out.println("猪吃白菜");

}

}

## 多态中的转型问题

* 向上转型
  + - 从子到父
    - 父类引用指向子类对象
* 向下转型
  + - 从父到子
    - 父类引用转为子类对象

### 示例代码

package com.itheima\_04;

/\*

\* 向上转型

\* 从子到父

\* 父类引用指向子类对象

\* 向下转型

\* 从父到子

\* 父类引用转为子类对象

\*/

public class DuoTaiDemo {

public static void main(String[] args) {

//多态

Animal a = new Cat(); //向上转型

a.eat();

//a.playGame();

//多态的弊端：无法访问子类特有方法

//现在我就想使用子类特有方法，怎么办呢?

//创建子类对象就可以了

/\*

Cat c = new Cat();

c.eat();

c.playGame();

\*/

//现在的代码虽然可以访问子类的特有功能，但是不合理

//因为我们发现内存中有两个猫类的对象

//这个时候，我们得想办法把多态中的猫对象还原

//这个时候，就要使用多态中的转型了

//父类引用转为子类对象

Cat c = (Cat)a;

c.eat();

c.playGame();

}

}

package com.itheima\_04;

public class Cat extends Animal {

public void eat() {

System.out.println("猫吃鱼");

}

public void playGame() {

System.out.println("猫捉迷藏");

}

}

package com.itheima\_04;

public class Animal {

public void eat() {

System.out.println("吃东西");

}

}

## 多态中的内存图

