Java OOP 系列课程

时间: 2017/11/01
作者:袁毅雄
邮箱: 896778954@qq.com

目录

Day01

JavaOOP

对象：在现实生活中，具体客观存在的东西就是对象

特征

静态特征[名词] (属性):是，有，属于...

顾客的姓名是张三

动态特征[动词](方法):可以，能，在...

顾客正在购买商品

```
//java 的类
public class Client {

   //属性,成员变量
   //访问修饰符  数据类型    变量名;
   public    String    name;
   public    int         age;


   //方法,成员方法,函数
   //访问修饰符  返回值类型  方法的名称(形参列表){
   //方法体
   //}
   public    void shopping(){

   }
```

}

类和对象的关系

类是模板，对象是类的具体实例

方法

成员方法

```
//形参列表(形式上告诉你需要什么参数)
    public int add(int num1,int num2){
        int sum=num1+num2;
        return sum;
    }

//实参列表
    int sum=对象名.add(50,20);
```

构造方法

```
public class Student {
    // 姓名、年龄、班级、爱好
    public String name;
    public int age;
    public String clazz;
    public String hobby;
    // shift +Alt + s
    // shift +Alt + f

    // 自我介绍
    public void showInfo() {
        System.out.println(name);
        System.out.println("年龄：" + age);
        System.out.println("就读于：" + clazz);
        System.out.println("爱好:" + hobby);
    }
```

```
    //构造方法
    //每个类默认都有一个无参的构造方法
    //一旦显示的提供了一个有参的构造方法,无参的将会被覆盖
    public Student(String name,int age,String clazz,String hobby){
        //数据初始化
        System.out.println("开始数据初始化 ");

        //当局部变量和成员变量相互冲突的时候，优先使用局部变量
        this.name=name;
        this.age=age;
        this.clazz=clazz;
        this.hobby=hobby;
        System.out.println("结束数据初始化 ");
    }
}



        Student arvin = new Student("Arvin",16,"java4","足球");
        arvin.showInfo();
```

this 关键字的使用

代表当前对象

构造方法的调用,只能发生在构造方法的第一行	
this()	在调用本类的无参构造方法
this(“arvin”)	在调用本类的有参构造方法
this.name	在调用本类的属性
this.setName(“arvin”)	在调用本类的方法

概念

[构造方法]每个类默认都有一个无参的构造方法

[构造方法]一旦显示的提供了一个有参的构造方法,无参的将会被覆盖

[局部变量和成员变量]当局部变量和成员变量相互冲突的时候，优先使用局部变量

[局部变量]局部变量没有默认值，必须赋值之后才能使用

[成员变量/属性] 成员变量有默认值，无需赋值便可使用

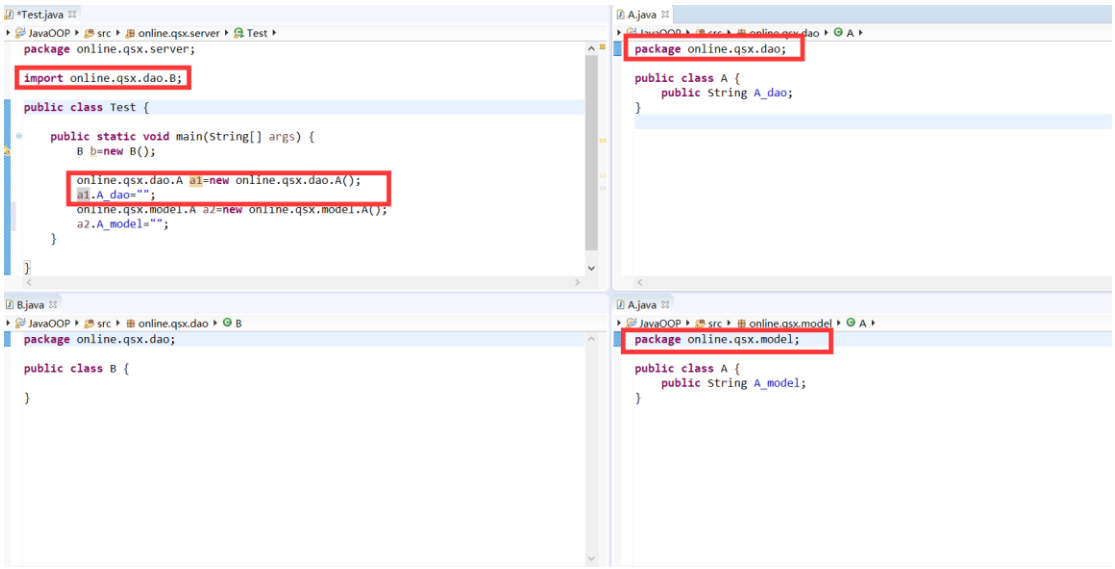
Day02

javaOOP

包

MVC(Model View Controller)命名		
com.baidu.www		
	model/vo/entity/pojo	[Model]
	dao	
	impl	
	server	
	impl	
	servlet/action/controller	[Conteoller]
	WebContent	[View]
	css	
	js	

jsp(动态文件) html(静态文件)



方法重载

```
public class Max {
    public static void main(String[] args) {
        Conuter conuter = new Conuter();

        System.out.println(conuter.add(50, 60, 80));
    }
}

class Conuter {
    //方法的重载
    //在同一个类里面
    //方法名相同
    //参数列表不同(个数不同/类型不同)
    //与
    //返回值                    无关
    //访问修饰符                无关

    public int add(int number1, int number2) {
        System.out.println("13123412");
        return number1 + number2;
    }

    private int add(int number1, int number2,int numner3) {
```

```
        System.out.println("1");
        return number1 + number2;
    }

    public double add(double number1, double number2, double number3) {
        System.out.println("2");
        return number1 + number2 + number3;
    }
}
```

访问修饰符

```
    //访问修饰符
    private String test1;     // 私有的                                *
    protected String test2;   // 受保护的        继承
    String test3;             // 默认的              包
    public String test4;      // 公开的              任意位置使用       *
```

修饰符 \ 作用域	同一个类中	同一个包中	子类中	任何地方
private	可以	不可以	不可以	不可以
默认修饰符	可以	可以	不可以	不可以
protected	可以	可以	可以	不可以
public	可以	可以	可以	可以

静态方法

```
public static int findMax(int[] numbers) {
        int max = numbers[0];
        for (int i = 1; i < numbers.length; i++) {
            if (numbers[i] > max) {
                max = numbers[i];
            }
        }
        return max;
    }

类名.findMax(numbers);
```

```
对象名. findMax(numbers);
```

静态变量

```
public static int name;

类名.name=15;//和对象没有关系
对象名.name=50;
```

封装

第一步：提供 private 属性

第二步：提供 public 的 getXxx/setXxx 方法访问属性

第三步[可选]:针对 setXxx 的数据进行校验

```
package online.qsx.model;

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student stu = new Student();
        stu.setAge(25);
        stu.name = "张三";
        stu.showInfo();
    }

}

class Student {
    private int age;

    public int getAge() {
        return age;
    }
```

```
    public void setAge(int age) {
        if (age > 150 || age <= 0) {
            this.age = 16;
        } else {
            this.age = age;
        }
    }

    public String name;

    public void showInfo() {
        System.out.println(name + "的年龄为:" + age);
    }
}
```

继承

要求 is a 关系： Dog is a Pet

```
package online.qsx.model;

public class Test {

    public static void main(String[] args) {
        Student stu = new Student();
        stu.age = 15;
        stu.name = "";
        stu.studentNo = "";
    }

}
//父类
class Person {
    public int age;
    public String name;
}
// Student is a person
//子类
class Student extends Person {
    public String studentNo;
```

```
}
// Teacher is a person
class Teacher extends Person {
    public String teacherNo;
}
//类的初始化过程：先父类在子类
```

Day03

JavaOOP

构造方法

```
// 先运行父类的构造方法在运行子类的构造方法

//方法名称和类名一致
//没有返回值类型
//所有类默认都有一个默认的无参的构造方法
//如果显示的编写了有参的构造方法，那么无参构造方法将会没有
//在继承情况下,父类如果没有无参的构造方法,子类没有显示的去写调用父类的哪一个构造方法，系统会默认调用父类的无参构造方法，所以会报错
```

super 关键字的使用

代表父类

```
构造方法的调用,只能发生在构造方法的第一行
super()                    在调用父类的无参构造方法
super (“arvin”)            在调用父类的有参构造方法

super.name                 在调用父类的属性

super.setName(“arvin”)     在调用父类的方法
```

向上向下转型

向上

```
class Pet {}
class Dog extends Pet {}
class Penguin extends Pet {}

Penguin penguin=new Penguin();
//父类类型 对象名=子类对象;
Pet pet= penguin; // 父类类型指向子类实例,向上类型转换

Pet pet = new Penguin();// 父类类型指向子类实例,向上类型转换
```

向下

```
class Pet {}
class Dog extends Pet {}
class Penguin extends Pet {}

//子类类型 对象名=(子类类型)父类对象;
Dog dog= (Dog) pet1; //向下类型转换,可能存在异常

//父类对象 instanceof 子类类型;
pet1 instanceof Dog   reurn true/false
```

方法重写

```
class Pet {
    protected void print() {
    }
}
class Dog extends Pet {
    // 方法的重写
    // 在继承情况下,子类中
    // 方法名相同,参数列表数据个数相同,类型兼容
    // 返回值类型相同或者兼容
    // 访问修饰符不能严与父类,最低和父类一致
    @Override // 可以用来验证该方法是否是方法的重写
```

```
    public void print() {
    }
}
```

抽象类/抽象方法

```
abstract class Pet {
    public Pet(){}
    //抽象方法可以不写方法体
    //有抽象方法类一定是抽象类
    //抽象类里面不一定有抽象方法
    //抽象类中可以有普通的成员方法
    //子类必须实现父类所有的抽象方法,除非子类也是一个抽象类
    //抽象类不能被直接实例化
    //抽象类可以有构造方法
    protected abstract void print();
}

class Dog extends Pet {

    @Override
    public void print() {

    }
}
```

多态

使用父类类型实现多态

父类类型变量

```
    public static void main(String[] args) {
        //多态的实现方式
        //父类类型指向子类实例
        Pet pet=new Penguin();
        //运行的子类方法
        pet.print();
    }
```

```
class Pet {
    public void print() {
        System.out.println("Pet");
    }
}
class Dog extends Pet {
    @Override
    public void print() {
        System.out.println("Dog");
    }
}
class Penguin extends Pet {
    @Override
    public void print() {
        System.out.println("Penguin");
    }
}
```

参数

```
    public static void main(String[] args) {
        Person arvin = new Person();

        Dog dog = new Dog();
        Penguin penguin = new Penguin();
        Cat cat=new Cat();

        arvin.feed(dog);
        arvin.feed(penguin);
        arvin.feed(cat);
    }

//人
class Person {
    /**
     * 喂养
     * @param pet
     */
    public void feed(Pet pet) {
        System.out.println("主人喂食---猪食");
```

```
        pet.eat();
    }
}
//动物
abstract class Pet {
    /*
     * 吃
     */
    public abstract void eat();
}
//狗
class Dog extends Pet {
    @Override
    public void eat() {
        System.out.println("狗开始吃饭了");
    }
}
//企鹅
class Penguin extends Pet {
    @Override
    public void eat() {
        System.out.println("企鹅开始吃饭了");
    }
}
//猫
class Cat extends Pet {
    @Override
    public void eat() {
        System.out.println("猫开始吃饭了");
    }
}
```

返回值

关键代码看下面的设计模式

23 种设计模式之-工厂模式(简单工厂)

```
public static void main(String[] args) {
        PetShop petShop=new PetShop();
        Pet pet=petShop.market("狗");
```

```
        pet.print();
    }

// 宠物商店
class PetShop {
    public Pet market(String code) {
        Pet pet=null;
        switch (code) {
        case "狗":
            pet=new Dog();
            break;
        case "企鹅":
            pet=new Penguin();
            break;
        }
        return pet;
    }
}
// 动物
abstract class Pet {
    public abstract void print();
}
// 狗
class Dog extends Pet {
    @Override
    public void print() {
        System.out.println("Dog....");
    }
}
// 企鹅
class Penguin extends Pet {
    @Override
    public void print() {
        System.out.println("Penguin....");
    }
}
```

Day04

JavaOOP

接口

多态

Day05

异常

异常处理

异常的结构体系

异常的分类

自定义异常