方法重写和多态

1. 方法重写概念

> 方法重写 Overrides 方法重载 Overload
>
> 1.存在于父子类之间(子类对父类的方法进行重写 必须有继承关系)
>
> 2.方法名称相同
>
> 3.参数列表相同
>
> 4.返回值相同(或者是其子类)
>
> 5.访问权限不能小于(严于)父类
>
> 6.静态方法可以继承 但是不能被重写
>
> 7.不能抛出、声明比父类更多的异常
>
> @Override 注解表示子类重写父类的方法 可以提高代码的阅读性

```
package com.atguigu.test1;

/**
 *  宠物类 父类
 *  父类中书写子类共有的信息（属性和方法）
 */
public class Pet {
    private String name;
    private int health;
    private int love;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getHealth() {
        return health;
    }

    public void setHealth(int health) {
        this.health = health;
    }

    public int getLove() {
```

```
        return love;
    }

    public void setLove(int love) {
        this.love = love;
    }

     void print(){
        System.out.println("宠物的名字是: " + name);
        System.out.println("宠物的健康值是: " + health);
        System.out.println("宠物的亲密值是: " + love);
    }


    public Pet(){
    }

    public Pet(String name,int health,int love){
        this.name = name;
        this.health = health;
        this.love = love;
    }
}
```

```
package com.atguigu.test1;

/**
 *  狗类
 */
public class Dog extends Pet {
    private String strain;
    public String getStrain() {
        return strain;
    }
    public void setStrain(String strain) {
        this.strain = strain;
    }

    public Dog(){
    }

    public Dog(String strain){
        this.strain = strain;
    }


    public Dog(String name,int health,int love,String strain){
        super(name,health,love);
        this.strain = strain;
    }

    /**
```

```
     *  方法重写 Overrides  方法重载 Overload
     *  1.存在于父子类之间(子类对父类的方法进行重写 必须有继承关系)
     *  2.方法名称相同
     *  3.参数列表相同
     *  4.返回值相同(或者是其子类)
     *  5.访问权限不能小于(严于)父类
     *  6.静态方法可以继承 但是不能被重写
     *
     *  7.不能抛出、声明比父类更多的异常
     *
     * @Override 注解表示子类重写父类的方法 可以提高代码的阅读性
     */
    @Override
    protected void print() {
        super.print();
        System.out.println("狗狗的品种是:" + strain);
    }


}
```

```
package com.atguigu.test1;

/**
 *  企鹅类
 */
public class Penguin extends Pet {
    private String sex;

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public Penguin(){}

    public Penguin(String name,int health,int love,String sex){
        super(name,health,love); // alt + shift + ↑↓ 移动整行代码
        this.sex = sex;
    }


    public void print(){
        super.print();
        System.out.println("企鹅的性别是: " + sex);
    }

}
```

2. Object类

> Object类是所有类的父类 所有对象包括数组都实现了此类的方法。
>
> 我们通常对Object类中的方法进行重写，以实现自定义需求的效果(私人订制)，

3. 重写toString方法

> 当我们直接打印一个对象 相当于调用此对象的toString方法
>
> 1.为什么直接输出对象会出现包名+ 类名(全限定名) + hash值
>
> 因为直接打印一个对象 相当于调用此对象的toString方法
>
> 2.为什么调用toString就会出现包名+ 类名(全限定名) + hash值
>
> 因为父类Object类中就是这样实现的
>
> 3.我们自定义的类 为什么要重写toString呢?
>
> 因为我们通常获取到包名类名hash值 是没有用的 我们希望直接打印对象 就获取到此对象的属性名 和 属性值

```
package com.atguigu.test2;

public class Student{
    String name;
    int age;

    public String toString(){
        return "Student{name = '" + name + "',age = " + age + "}";
    }

    public static void main(String[] args) {
        Student stu1 = new Student();
        stu1.name = "赵四";
        stu1.age = 25;

        // 当我们直接打印一个对象 相当于调用此对象的toString方法
        // 1.为什么直接输出对象会出现包名+ 类名(全限定名) + hash值
        // 因为直接打印一个对象 相当于调用此对象的toString方法
        // 2.为什么调用toString就会出现包名+ 类名(全限定名) + hash值
        // 因为父类Object类中就是这样实现的
        // 3.我们自定义的类 为什么要重写toString呢?
        // 因为我们通常获取到包名类名hash值 是没有用的 我们希望直接打印对象 就获取到此对象的属性名 和 属性值
        System.out.println(stu1);

        System.out.println(stu1.toString());


    }
```

```
}
```

4. 重写equals方法

> **面试题: == 和equals的区别?**
>
> ==比较基本数据类型 比较值
>
> ==比较引用数据类型 比较地址
>
> equals本身也比较地址 但是我们可以重写equals方法 自定义比较规则
>
> String类就是对Object类中的equals方法进行了重写： 将原本的比较地址 重写了为了比较地址 并且 比较内容
>
> 为什么要重写equals方法?
>
> 因为父类中的equals方法实现原本为比较地址 但是在实际开发中 我们要结合现实生活中的情况
>
> 来比较对象 比如 比较两个人 根据名字 和 身份证号比较更加合理
>
> 所以我们就可以重写equals方法 将原本的比较地址 重写为比较名字和身份证号
>
> 如何区分当前调用的哪个类中的方法 或者 哪个类中的属性?
>
> 观察点之前的对象是哪个类的对象 即调用哪个类中的属性和方法

```
package com.atguigu.test3;

/**
 *  alt + insert
 *
 *  equals方法本身的作用?
 *  比较两个对象的地址值是否相同 如果是则返回为true  如果不是则返回为false
 *
 *  面试题: == 和equals的区别?
 *  ==比较基本数据类型 比较值
 *  ==比较引用数据类型 比较地址
 *  equals本身也比较地址 但是我们可以重写equals方法 自定义比较规则
 *  String类就是对Object类中的equals方法进行了重写 ： 将原本的比较地址 重写了为了比较地址 并且 比较内容
 *
 *  为什么要重写equals方法?
 *  因为父类中的equals方法实现原本为比较地址 但是在实际开发中 我们要结合现实生活中的情况
 *  来比较对象 比如 比较两个人 根据名字 和 身份证号比较更加合理
 *  所以我们就可以重写equals方法 将原本的比较地址 重写为比较名字和身份证号
 *
 *  如何区分当前调用的哪个类中的方法 或者 哪个类中的属性?
 *  观察点之前的对象是哪个类的对象 即调用哪个类中的属性和方法
 *
 *
 *
 */
```

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getIdCard() {
        return idCard;
    }

    public void setIdCard(String idCard) {
        this.idCard = idCard;
    }

    public Person(String name, String idCard) {
        this.name = name;
        this.idCard = idCard;
    }

    public Person() {
    }


    public boolean equals(Object obj){
        if(this == obj){
            return true;
        }
        // 为什么形参的Object类型的 ?
        // 因为这里是方法重写 要求参数列表必须跟父类相同 而父类中就是Object类型

        // 为什么这里要强制类型转换?
        // 因为Object类型的参数是无法访问name和idCard这两个属性的 所以要向下转型
        Person p1 = (Person)obj;

        if(this.name.equals(p1.name) && this.idCard.equals(p1.idCard)){
            return true;
        }
        return false;
    }




    public static void main(String[] args) {
        Person p1 = new Person("赵四", "56789454798764612984552");

        Person p2 = new Person("赵四", "56789454798764612984552");
```

```
        System.out.println(p1 == p2); // false
        System.out.println(p1.equals(p2)); // false

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

        String str1 = new String("abc");
        String str2 = new String("abc");

        System.out.println(str1 == str2); // false
        System.out.println(str1.equals(str2)); // true


        String str3 = new String("hello world");
        String str4 = str3;

        System.out.println(str3 == str4);
    }


}
```

5.重写hashCode方法

> hashCode方法的作用？ 返回当前对象的hash值
>
> hash值是什么?
>
> hash值并不是地址值 Java中的地址我们是无法获取到的
>
> 根据对象的地址等信息 使用杂凑算法所计算出来的一个数值
>
> 为什么要重写hashCode方法?
>
> 1.默认情况下 hashCode 是根据地址所计算出来的 equals方法是比较地址
>
> 所以 两个对象equals比较为true 则hash值相同/相等 但是目前我们重写了equals方法 打破了这种默认规则
>
> 所以 我们要继续重写hashCode以维持这种规则
>
> 2.在散列数据结构中 默认以equals比较为true 并且hashCode相同作为去除重复的依据
>
> 3.在实际开发中 equals方法和 hashCode方法是绑定在一起 要么都重写 要么都不重写
>
> 总结：最终重写hashCode()方法要实现的效果 当equals比较为true 则hashCode相等

```
package com.atguigu.test4;

import java.util.Objects;

/**
 *  hashCode方法的作用？ 返回当前对象的hash值
```

```
 *
 *  hash值是什么?
 *  hash值并不是地址值 Java中的地址我们是无法获取到的
 *  根据对象的地址等信息 使用杂凑算法所计算出来的一个数值
 *
 *  为什么要重写hashCode方法?
 *  1.默认情况下 hashCode 是根据地址所计算出来的  equals方法是比较地址
 *  所以 两个对象equals比较为true 则hash值相同/相等  但是目前我们重写了equals方法 打破了这种默认规则
 *  所以 我们要继续重写hashCode以维持这种规则
 *
 *  2.在散列数据结构中 默认以equals比较为true 并且hashCode相同作为去除重复的依据
 *
 *  3.在实际开发中 equals方法和 hashCode方法是绑定在一起 要么都重写 要么都不重写
 *
 *  总结: 最终重写hashCode()方法要实现的效果 当equals比较为true 则hashCode相等
 *
 */
public class Person {
    private String name;
    private String idCard;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getIdCard() {
        return idCard;
    }

    public void setIdCard(String idCard) {
        this.idCard = idCard;
    }

    public Person(String name, String idCard) {
        this.name = name;
        this.idCard = idCard;
    }

    public Person() {
    }



//    public boolean equals(Object obj){
//        if(this == obj){
//            return true;
//        }
//        Person p1 = (Person)obj;
```

```
//
//        if(this.name.equals(p1.name) && this.idCard.equals(p1.idCard)){
//            return true;
//        }
//        return false;
//    }

//    public  int hashCode() {
//        int result = 1; // 最终要返回的hash值结果、
//        int prime = 12; // 权重 31 计算hash值的重要条件
//        result = result * prime + this.name == null ? 0 : this.name.hashCode();
//        result = result * prime + this.idCard == null ? 0 : this.idCard.hashCode();
//
//        return result;
//    }


    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Person person = (Person) o;

        if (!Objects.equals(name, person.name)) return false;
        return Objects.equals(idCard, person.idCard);
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + (idCard != null ? idCard.hashCode() : 0);
        return result;
    }

    public static void main(String[] args) {
        Person p1 = new Person("赵四", "56789454798");
        Person p2 = new Person("赵四", "56789454798");

        System.out.println(p1 == p2);
        System.out.println(p1.equals(p2));


        System.out.println(p1.hashCode());
        System.out.println(p2.hashCode());


    }


}
```

为什么使用权重31计算hash值?

1.因为别人都用31

2.因为31是一个特殊的质数 任何数乘以31 等于这个数 左移5位 减去这个数本身

n * 31 = (n << 5) - n

总结：使用哪个数值计算hash值都可以 但是使用31效率更高

```
package com.atguigu.test4;

/**
 *  为什么使用权重31计算hash值?
 *  1.因为别人都用31
 *  2.因为31是一个特殊的质数  任何数乘以31  等于这个数  左移5位  减去这个数本身
 *  n * 31 = (n << 5) - n
 *
 *  总结：使用哪个数值计算hash值都可以  但是使用31效率更高
 */
public class TestPrime {
    public static void main(String[] args) {
        System.out.println(3 * 31);
        System.out.println((3 << 5) - 3);
    }
}
```

6. 多态

多态： 同一个引用类型 使用不同的实例而执行不同操作

父类引用指向子类对象 属于多态向上转型

向上转型：此时通过父类的引用 可以访问的是 子类重写父类的方法 或者 继承 父类的方法

不能访问子类独有的方法

使用向上转型将无法访问子类独有的方法，为什么还要使用呢?

因为使用了向上转型以后 虽然调用的方法个数受到了影响 但是我们可以通过这种方式提高程序的灵活性

多态向上转型具体的表现：

1.父类作为形参 子类作为实参

2.父类作为声明返回值 实际返回值为子类类型

3.父类类型的数组、集合 其元素为子类类型

对于创建对象的语句而言： 等号左边的称之为引用 等号右边的为对象

```
package com.atguigu.test5;

/**
```

```java
 *  宠物类 父类
 *  父类中书写子类共有的信息（属性和方法）
 */
public class Pet {
    private String name;
    private int health;
    private int love;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getHealth() {
        return health;
    }

    public void setHealth(int health) {
        this.health = health;
    }

    public int getLove() {
        return love;
    }

    public void setLove(int love) {
        this.love = love;
    }

     void print(){
        System.out.println("宠物的名字是: " + name);
        System.out.println("宠物的健康值是: " + health);
        System.out.println("宠物的亲密值是: " + love);
    }


    public Pet(){
    }

    public Pet(String name,int health,int love){
        this.name = name;
        this.health = health;
        this.love = love;
    }

    public void cure(){
        System.out.println("宠物看病");
    }
```

```
}
```

```
package com.atguigu.test5;

/**
 *  狗类
 */
public class Dog extends Pet {
    private String strain;
    public String getStrain() {
        return strain;
    }
    public void setStrain(String strain) {
        this.strain = strain;
    }

    public Dog(){
    }

    public Dog(String strain){
        this.strain = strain;
    }


    public Dog(String name,int health,int love,String strain){
        super(name,health,love);
        this.strain = strain;
    }

    protected void print() {
        super.print();
        System.out.println("狗狗的品种是:" + strain);
    }

    public void cure(){
        System.out.println("狗狗看病，打针，吃药，吃骨头，健康值恢复");
        setHealth(100);
    }


}
```

```
package com.atguigu.test5;

/**
 *  企鹅类
 */
public class Penguin extends Pet {
    private String sex;
```

```
    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public Penguin(){}

    public Penguin(String name,int health,int love,String sex){
        super(name,health,love); // alt + shift + ↑↓ 移动整行代码
        this.sex = sex;
    }


    public void print(){
        super.print();
        System.out.println("企鹅的性别是: " + sex);
    }

    public void cure(){
        System.out.println("企鹅看病, 打针, 疗养, 吃小鱼, 健康值恢复");
        super.setHealth(100);
    }

}
```

```
package com.atguigu.test5;

/**
 *  主人类
 *  属性: 名字 年龄 性别 ……
 *  方法:
 *      1.带宠物去看病
 */
public class Master {
    public void toHospitalWithDog(Dog dog){
        dog.cure();
    }

    public void toHospitalWithPenguin(Penguin penguin){
        penguin.cure();
    }

    // 以上两个方法可以实现给现有宠物看病 但是未来如果有更多的宠物子类
    // 则还需要编写更多的方法来实现宠物看病
    // 这种方式 不符合 开闭原则(软件设计中的一个原则)
    // 开 对扩展开放  闭 对修改源代码关闭
    // 我们应该编写一个方法 用于实现给所有的宠物看病
```

```
    public void toHospitalWithPet(Pet pet){
        pet.cure();
    }









}
```

```
package com.atguigu.test5;


/**
 *  多态的方式实现宠物看病 测试类
 */
public class Test2 {
    public static void main(String[] args) {
        Master master = new Master();

        Dog dog = new Dog("大黄", 50, 100, "金毛");
        Penguin penguin = new Penguin("大白", 20, 100, "雄");
        Cat cat = new Cat();


        master.toHospitalWithPet(dog); // Pet pet = new Dog();
        master.toHospitalWithPet(penguin); // Pet pet = new Penguin();

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

        int a = 100;
        m1(a);
    }

    public static void m1(double num){
        System.out.println(num);
    }


}
```

> 多态实现学生去学校案例
>
> 分析：电脑有不同的具体的子类类型 而学生去学校 只需要根据实际的情况选择具体的子类产品即可 在代码编写阶段 我们只需要考虑需要什么产品 而不必考虑产品具体的属性 、类型

```
package com.atguigu.test6;
```

```java
/**
 *  学生类
 *  方法：去学校
 */
public class Student {

    public void gotoSchool(Computer computer){
        computer.printInfo();
    }

    public static void main(String[] args) {
        Student student = new Student();
        DeskComputer deskComputer = new DeskComputer();
        NoteBook noteBook = new NoteBook();
        IPad iPad = new IPad();

        student.gotoSchool(deskComputer);
        student.gotoSchool(noteBook);
        student.gotoSchool(iPad);
    }

}

class Computer{
    public void printInfo(){
        System.out.println("电脑信息打印");
    }
}

class DeskComputer extends Computer{
    public void printInfo(){
        System.out.println("台式电脑");
    }
}

class NoteBook extends Computer{
    public void printInfo(){
        System.out.println("笔记本电脑");
    }
}

class IPad extends Computer{
    public void printInfo(){
        System.out.println("平板电脑");
    }
}
```