多态和抽象类

一 内容回顾（列举前一天重点难点内容）

1.1 教学重点:

```
1.掌握继承的基本使用
2.掌握方法的重写
3.掌握继承中构造方法的使用
4.掌握final关键字的基本使用
5.掌握Object类的常用方法使用
```

1.2 教学难点:

```
1.继承思想的理解
    理解思想不是一朝一夕的事情,大家可以先学会基本的语法,使用,再慢慢的不断深入理解
2.继承中使用构造方法
3.空白final
```

二 教学目标

```
1.掌握多态的基本概念
2.掌握多态的优缺点
3.掌握多态中的向上转型,向下转型
4.掌握抽象类的基本概念
5.掌握抽象类的基本使用
6.了解多态中instanceof关键字的使用
7.了解多态的实现原理
```

三 教学导读

3.1. 多态

生活中的多态， 是指的客观的事物在人脑中的主观体现。 例如， 在路上看到一只哈士奇， 你可以看做是哈士奇， 可以看做是狗， 也可以看做是动物。

主观意识上的类别， 与客观存在的事物， 存在 `is a` 的关系的时候， 即形成了多态。

在程序中， 一个类的引用指向另外一个类的对象， 从而产生多种形态。 当二者存在直接或者间接的继承关系时， 父类引用指向子类的对象， 即形成多态。

多态是面向对象三大特性之一,学好多态对我们学习java非常重要,记住继承是多态的前提,如果类与类之间没有继承关系,也不会存在多态.

3.2. 抽象类

程序,是来描述现实世界,解决现实问题的.

例如,我们在进行百度图片搜索的时候,搜索的关键字是“动物”, 但是搜索的结果却都是“动物”的子类对象.

在现实世界中,存在的都是“动物”具体的子类对象,并不存在“动物”对象.所以Animal类不应该被独立创建成对象.

对于这样的场景,我们可以将动物类,设计为抽象类.抽象类不能被实例化对象,只是提供了所有的子类共有的部分。

四 教学内容

4.1. 多态

4.1.1. 多态实现原理(会)

- 多态:在代码中的描述是用父类的引用指向子类的对象

```
父子关系:Student  extends  Person      Person extends Object
//直接父类的引用指向子类对象---多态
Person person = new Student();
//Object不是Student的直接父类,但是是间接父类,这里也是多态
Object o = new Student();
```

- java程序运行分成两个阶段:编译,运行

编译阶段:从打开程序到执行运行之前---只能识别=前面的引用类型,不会识别=后面的对象

运行阶段:从运行开始---识别=后面对象,对象开始干活儿

```
例如:Person person = new Student();
编译阶段识别: person  是Person类的引用
运行阶段识别:new出来的Student对象
```

- 动态机制:(了解)

类型:动态类型,动态绑定,动态加载

动态加载:我们在编译阶段不能确定具体的对象类型,只有到了运行阶段才能确定真正的干活儿的对象.

多态就是典型的动态加载

- 在多态下只能直接调用父类有的方法,不能直接调用子类特有的方法?.

例如:下面代码中person不能直接调用run方法

工作机制:1.首先通过Person保存的地址找到Student对象 2.Student对象再去调用run方法.

原因:在编译的时候识别的是引用类型,不识别对象.所以只能识别出Person里面的方法,而不能直接调用子类特有的方法.

```
public class Demo4 {
    public static void main(String[] args) {
        //继承--使用的一定是子类
        Student student = new Student();
        student.show();

        //多态:在代码中的描述就是用父类的引用指向子类的对象
        Person person = new Student();

       //可以调用,在show方法在父类Person里面
        person.show();

        //这里运行会报错,原因:在Person类里面找不到run方法
        //person.run();
    }
}

class  Person{
    String name;
    public  void  show(){
        System.out.println("show");
```

```
    }
}
class  Student extends  Person{
    int age;
    public  void  run(){
        System.out.println("run");
    }
}
```

4.1.2. 多态的优点(会)

可以提高代码的扩展性,使用之前定义好的功能,后面直接拿来使用,不用再创建新的方法.

- 实例:喂动物

使用多态之前

说明:我们发现喂每一种动物都需要单独写feed方法,这样会造成大量的代码冗余,降低代码的扩展性,我们可以使用多态解决这个问题.

```
public class Demo5 {
    public static void main(String[] args) {
        //创建狗,猫,动物对象
        Dog dog = new Dog();
        Cat cat = new Cat();
        Animal animal = new Animal();

        //分别调用方法喂猫,喂狗,喂动物
        feedAnimal(animal);
        feedDog(dog);
        feedCat(cat);
    }

```

```
14      //喂狗,喂猫,喂动物
15      public static void feedAnimal(Animal animal) {
16          animal.eat();
17      }
18      public static  void  feedDog(Dog dog) {
19          dog.eat();
20      }
21      public static  void  feedCat(Cat cat) {
22          cat.eat();
23      }
24  }
25
26  class Animal{
27      public void eat() {
28          System.out.println("动物吃");
29      }
30  }
31
32  class Dog extends Animal{
33      @Override
34      public void eat() {
35          System.out.println("狗吃骨头");
36      }
37  }
38
39  class Cat extends Animal{
40      @Override
41      public void eat() {
42          System.out.println("猫吃");
43      }
44  }
45
```

使用多态之后

说明:使用了多态,我们可以直接将feedDog,feedCat省略掉调,使用feedAnimal可以喂所有动物

分析:在调用feedAnimal(dog)方法时,我们会将dog作为参数传给feedAnimal方法,相当于animal=dog=new Dog(),这里Animal和Dog是父子关系,形成了多态.这时我们只要保证在Animal类和Dog类中都有eat方法即可,Animal中的eat在编译时识别,Dog中的eat会在运行时真正识别出来.

```
public class Demo5 {
    public static void main(String[] args) {
      //创建狗,猫,动物对象
        Dog dog = new Dog();
        Cat cat = new Cat();
        Animal animal = new Animal();

        feedAnimal(animal);
//    feedDog(dog);
//    feedCat(cat);
        feedAnimal(dog);
        feedAnimal(cat);
    }

    //喂狗,喂猫,喂动物
    public static void feedAnimal(Animal animal)
{//animal = dog = new Dog()   多态
        animal.eat();
    }
//  public static  void  feedDog(Dog dog) {
//    dog.eat();
//  }
//  public static  void  feedCat(Cat cat) {
//    cat.eat();
//  }
```

```
}

class Animal{
    public void eat() {
        System.out.println("动物吃");
    }
}

class Dog extends Animal{
    @Override
    public void eat() {
        System.out.println("狗吃骨头");
    }
}

class Cat extends Animal{
    @Override
    public void eat() {
        System.out.println("猫吃");
    }
}
```

再添加一个动物-猪

说明:此时问题就变得简单了很多,我们只需要添加一个Pig类,调用feedAnimal方法直接喂猪即可.

```
public class Demo5 {
    public static void main(String[] args) {
        //创建狗,猫,动物对象
        Dog dog = new Dog();
        Cat cat = new Cat();
        Animal animal = new Animal();

```

```
        //创建猪对象
        Pig pig = new Pig();

        feedAnimal(animal);

//    feedDog(dog);
//    feedCat(cat);

        feedAnimal(dog);
        feedAnimal(cat);

        //调用feedAnimal方法喂猪
        feedAnimal(pig);
    }

    //喂狗,喂猫,喂动物
    public static void feedAnimal(Animal animal)
{//animal = dog = new Dog()   多态
        animal.eat();
    }
//  public static  void  feedDog(Dog dog) {
//    dog.eat();
//  }
//  public static  void  feedCat(Cat cat) {
//    cat.eat();
//  }
//  public static  void  feedPig(Pig pig) {
//    pig.eat();
//  }
}

class Animal{
    public void eat() {
```

```
40          System.out.println("动物吃");
41      }
42  }
43
44  class Dog extends Animal{
45      @Override
46      public void eat() {
47          System.out.println("狗吃骨头");
48      }
49  }
50
51  class Cat extends Animal{
52      @Override
53      public void eat() {
54          System.out.println("猫吃");
55      }
56  }
57
58  class Pig extends Animal{
59      public void eat() {
60          System.out.println("猪吃");
61      }
62  }
63
```

4.1.3. 对象转型(会)

注意:在多态的前提下再说向上转型,向下转型.

向上转型

对象由子类类型， 转型为父类类型， 即是向上转型。

- 向上转型是一个隐式转换，相当于自动类型转换, 一定会转型成功。
- 向上转型后的对象， 只能访问父类中定义的成员。
- 作用:实现多态

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 向上转型
 */
class Test {
    public static void main(String[] args) {
        // 1. 实例化一个子类对象
        Dog dog = new Dog();
        // 2. 转成父类类型
        Animal animal = dog;
        // 此时， 这个animal引用只能访问父类中的成员
        animal.name = "animal";
        animal.age = 10;
        animal.furColor = "white";  // 这里是有问题的， 访问不到
    }
}
class Animal {
    String name;
    int age;
}
class Dog extends Animal {
    String furColor;
}
```

向下转型

对象由父类类型， 转型为子类类型， 即是向下转型。

- 向下转型是一个显式转换，相当于强制类型转换, 有可能转型失败.
- 向下转型后的对象， 将可以访问子类中独有的成员。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 向下转型
 */
class Test {
    public static void main(String[] args) {
        // 1. 实例化一个子类对象
        Dog dog = new Dog();
        // 2. 转成父类类型
        Animal animal = dog;
        // 3. 转成子类类型
        Dog sub = (Dog)animal;

        //注意:
        //1.这不是父类的引用指向子类的对象,不是向上转型.这是使用子类的引用指向父类的对象是错误的.
        Dog  dog1 = new Animal();
        //2.这里在编译的时候不会报错,但是运行时出错,还是相当于子类引用指向父类对象.
        Animal animal1 = new Animal();
        Dog dog2 = (Dog)animal1;
    }
}
class Animal {
    String name;
    int age;
}
class Dog extends Animal {
```

```
28      String furColor;
29  }
```

4.1.4. instanceof关键字(会)

向下转型， 存在失败的可能性。 如果引用实际指向的对象， 不是要转型的类型， 此时强制转换， 会出现 **ClassCastException** 异常。

所以， 在向下转型之前， 最好使用 instanceof 关键字进行类型检查。

- instanceof:是一个运算符
- 构成: 对象 instanceof 类或类的子类
- 原理说明:确定当前的对象是否是后面的类或者子类的对象,是返回true,不是false
- 作用:进行容错处理,增加用户体验

```
1  /**
2   * @Author 千锋大数据教学团队
3   * @Company 千锋好程序员大数据
4   * @Description 向下转型
5   */
6  public class Demo7 {
7      public static void main(String[] args) {
8          Person person = new Student();
9          person = new Teacher();
10         //发生错误代码
11         //当用Student类型的引用指向Teacher类型的对象时,因为Student和Teacher没有关系.
12         //所以发生ClasscastException:类型转换异常,一旦发生了异常,程序会立刻停止
```

```
13  //          Student student = (Student)person;
14  //          student.run();
15
16          //解决问题:容错处理
17          //如果person对应的对象不是Student或者Student的子类的
    对象,这里返回false
18          if (person instanceof  Student){
19              Student student = (Student)person;
20              student.run();
21          }else {
22              System.out.println("提供的不是Student类型的对
    象");
23              //类型转换异常.
24              Exception exception = new
    ClassCastException("发生了类型转换异常");
25              exception.printStackTrace();
26          }
27
28          //注意:instanceof前后必须有继承关系或者前面的对象跟后
    面的类直接对应也可以.
29  //      if (new Dog() instanceof Person) {
30  //          System.out.println("hehe");
31  //      }
32      }
33  }
34
35  class  Person{
36      String name;
37      public  void  show(){
38          System.out.println("show");
39      }
40  }
41  class  Student extends  Person{
```

```
    int age;
    public  void  run(){
        System.out.println("run");
    }
}

class  Teacher extends  Person{
    int weight;

    @Override
    public void show() {
        System.out.println("Teacher-show");
    }
}

class Dog{

}
```

4.1.5. 多态中的方法重写(会)

当向上转型后的对象， 调用父类中的方法。 如果这个方法已经被子类重写了， 此时调用的就是子类的重写实现!

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 方法重写
 */
class Test {
    public static void main(String[] args) {
        // 1. 实例化一个子类对象， 并向上转型
        Animal animal = new Dog();
```

```
        // 2. 调用父类方法
        animal.bark();  // 因为在子类中已经重写过这个方法了，此时的输出结果是 Won~
    }
}
class Animal {
    public void bark() {
        System.out.println("Animal Bark~");
    }
}
class Dog extends Animal {
    @Override
    public void bark() {
        System.out.println("Won~");
    }
}
```

4.1.6 父子类出现同名成员(了解)

- 继承下的调用规则
  - 成员变量:调用子类的
  - 成员方法:调用子类的,子类没有再去调用父类的.
- 多态下的调用规则:
  - 成员变量:编译的时候能不能访问看父类,运行的时候也看父类
  - 成员方法:编译的时候能不能访问看父类,运行的时候看子类
  - 静态成员方法:编译运行都看父类

```
public class Demo8 {
    public static void main(String[] args) {
        //继承
        Zi zi = new Zi();
        zi.show(); //调用的子类的
```

```
        System.out.println(zi.age);//

        //多态
        Fu fu = new Zi();
        fu.show();//调用的是父类的
        System.out.println(fu.age);//调用的是父类的
    }


}

class Fu{
    int age = 10;
    public  void run() {
        System.out.println("Fu-run");
    }

    public  static void show() {
        System.out.println("Fu-show");
    }
}

class Zi extends Fu{
    int age = 5;
    public  void eat() {
        System.out.println("Zi-eat");
    }

    public static void show() {
        System.out.println("Zi-show");
    }
}
```

4.2. 抽象类(会)

4.2.1. 抽象类定义

在继承中,提取父类方法的时候,每个子类都有自己具体的方法实现,父类不能决定他们各自的实现方法,所以父类干脆就不管了,在父类中只写方法的声明(负责制定一个规则),将方法的实现交给子类.在类中只有方法声明的方法叫抽象方法,拥有抽象方法的类叫抽象类

- abstract:抽象的
- 声明:不写函数体的函数,可以叫声明
- abstract修饰方法:抽象方法
- abstract修饰类:抽象类
- 抽象类的功能:1.可以节省代码 2.可以制定一批规则

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 抽象类
 */
abstract class Animal {
    String name;
    int age;
    public void eat() {}
}
```

4.2.2. 抽象方法

```java
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 抽象类
 */
abstract class Animal {
    public void bark() {
        System.out.println("Animal bark");
    }
}
class Dog extends Animal {
    @Override
    public void bark() {
        System.out.println("Won~");
    }
}
class Cat extends Animal {
    @Override
    public void bark() {
        System.out.println("Miao~");
    }
}
```

在上述代码中， 在父类Animal中，定义了一个bark方法， 但是这个方法无法满足子类的需求。

如果这个方法， 在所有的子类中都进行了重写， 那么这个方法在父类中怎么实现， 就没什么意义了。

此时， 如果实现了这个方法， 显得冗余； 如果不定义这个方法， 则表示所有的Animal类或子类对象都没有了bark方法。

针对这样的情况， 我们可以将这样的方法定义成抽象方法。

抽象方法的特点

- 抽象方法只有方法的声明， 没有实现。
- 抽象方法只能定义在抽象类中。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 抽象类
 */
abstract class Animal {
    // 抽象方法， 只有方法的声明， 没有方法的实现
    // 只能定义在抽象类中
    public abstract void bark();
}
```

4.2.3. 抽象类的继承

抽象方法有一个特点， 就是只能定义在抽象类中。 如果一个非抽象子类继承自一个抽象父类， 此时， 可以继承到抽象父类中的抽象方法。 那么这个时候， 抽象方法就存在于一个非抽象的子类中了。 此时会有问题。

所以， 非抽象的子类， 在继承自一个抽象父类的时候， 必须重写实现抽象父类中所有的抽象方法或者将自己也变成抽象类

注意:抽象类不能直接创建对象,可以通过子类间接的创建对象.

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
```

```
 * @Description 抽象类
 */
public class Demo8 {
    public static void main(String[] args) {
        //抽象类不能直接创建对象,可以通过子类间接的创建对象.
//        Animal animal = new Animal();
    }
}
abstract class Animal {
    // 抽象方法,  只有方法的声明,  没有方法的实现(前面必须添加abstract关键字)
    // 只能定义在抽象类中
    public abstract void bark();
}
//处理方法一:重写父类抽象方法
class Dog extends Animal {
    @Override
    public void bark() {
        System.out.println("won~");
    }
}

//处理方法二:将自己变成抽象方法
abstract class Dog extends Animal {
}
```

4.2.4. 抽象类总结

基本点总结:

- 抽象类不一定有抽象方法,但是有抽象方法的一定是抽象类.
- 继承了抽象类的子类一定要实现抽象方法,如果不实现就只能将自己也变成抽象的.

- 抽象类不能直接创建对象,必须通过子类实现,所以抽象类一定有子类

比较普通类与抽象类:

- 普通类可以直接创建对象
- 抽象类可以有抽象方法

比较:final,abstract,static,private

- 三个都是不能与abstract同时存在的关键字
- final:被final修饰的类不能有子类,方法不能重写,但是abstract必须有子类,必须重写
- static:修饰的方法可以通过类名调用,abstract必须通过子类实现
- private:修饰的方法不能重写,abstract必须重写

4.2.5. 抽象类的使用场景

上述说到，非抽象类在继承自一个抽象父类的同时，必须重写实现父类中所有的抽象方法。因此，抽象类可以用来做一些简单的规则制定。

在抽象类中制定一些规则，要求所有的子类必须实现，约束所有的子类的行为。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 抽象类
 */
abstract class Express {
    // 制定所有的子类必须实现的方法
    // 要求所有的子类，都必须实现这个方法
    public abstract void express();
```

```
}
class Shunfeng extends Express {
    @Override
    public void exress() {
        System.out.println("顺丰发送快递");
    }
}
class EMS extends Express {
    @Override
    public void express() {
        System.out.println("EMS发送快递");
    }
}
```

> 上述案例中，快递抽象父类中定义了抽象方法，发送快递。要求所有的子类必须实现。
>
> 此时，在父类中定义了所有的子类必须要实现的功能。

但是，如果抽象类进行行为的约束、规则的制定，又有很大的约束性。因为类是单继承的，一个类有且只能有一个父类。所以，如果一个类需要受到多种规则的约束，无法再继承其他的父类。 此时，可以使用接口进行这样复杂的规则制定。

课上练习

求圆和矩形的面积

```
abstract  class  Shape{
    public abstract double getArea();
}

class  Circle1 extends  Shape{
    final  double PI = 3.14;
```

```
    double r;
    @Override
    public double getArea() {
        return r*r*PI;
    }
}

class  Rect extends  Shape{
    double length;
    double wide;

    @Override
    public double getArea() {
        return length*wide;
    }
}
```