day14

今日内容

- 多态
- 包
- final
- 权限修饰符
- 代码块

教学目标

- √ 能够说出使用多态的前提条件
- √ 理解多态的向上转型
- √ 理解多态的向下转型
- √ 能够知道多态的使用场景
- √ 包的作用
- √ public和private权限修饰符的作用
- √ 描述final修饰的类的特点
- √ 描述final修饰的方法的特点
- √ 描述final修饰的变量的特点

第一章 多态

1.1 多态的形式

多态是继封装、继承之后，面向对象的第三大特性。

多态是出现在继承或者实现关系中的。

多态体现的格式：

```
父类类型 变量名 = new 子类/实现类构造器;
变量名.方法名();
```

多态的前提：有继承关系，子类对象是可以赋值给父类类型的变量。例如Animal是一个动物类型，而Cat是一个猫类型。Cat继承了Animal，Cat对象也是Animal类型，自然可以赋值给父类类型的变量。

1.2 多态的使用场景

如果没有多态，在下图中register方法只能传递学生对象，其他的Teacher和administrator对象是无法传递给register方法方法的，在这种情况下，只能定义三个不同的register方法分别接收学生，老师和管理员。

有了多态之后，方法的形参就可以定义为共同的父类Person。

要注意的是：

- 当一个方法的形参是一个类，我们可以传递这个类所有的子类对象。
- 当一个方法的形参是一个接口，我们可以传递这个接口所有的实现类对象（后面会学）。
- 而且多态还可以根据传递的不同对象来调用不同类中的方法。

代码示例：

```
父类：
public class Person {
    private String name;
    private int age;

    空参构造
    带全部参数的构造
    get和set方法

    public void show(){
        System.out.println(name + ", " + age);
    }
}

子类1：
public class Administrator extends Person {
    @Override
    public void show() {
        System.out.println("管理员的信息为：" + getName() + ", " +
getAge());
    }
}
```

```
22
23  子类2：
24  public class Student extends Person{
25
26      @Override
27      public void show() {
28          System.out.println("学生的信息为：" + getName() + ", " +
    getAge());
29      }
30  }
31
32  子类3：
33  public class Teacher extends Person{
34
35      @Override
36      public void show() {
37          System.out.println("老师的信息为：" + getName() + ", " +
    getAge());
38      }
39  }
40
41  测试类：
42  public class Test {
43      public static void main(String[] args) {
44          //创建三个对象，并调用register方法
45
46          Student s = new Student();
47          s.setName("张三");
48          s.setAge(18);
49
50
51          Teacher t = new Teacher();
52          t.setName("王建国");
53          t.setAge(30);
54
55          Administrator admin = new Administrator();
56          admin.setName("管理员");
57          admin.setAge(35);
58
59
60
61          register(s);
```

```
62          register(t);
63          register(admin);
64
65
66      }
67
68
69
70      //这个方法既能接收老师，又能接收学生，还能接收管理员
71      //只能把参数写成这三个类型的父类
72      public static void register(Person p){
73          p.show();
74      }
75  }
```

1.3 多态的定义和前提

多态： 是指同一行为，具有多个不同表现形式。

从上面案例可以看出，Cat和Dog都是动物，都是吃这一行为，但是出现的效果（表现形式）是不一样的。

前提【重点】

1. 有继承或者实现关系
2. 方法的重写【意义体现：不重写，无意义】
3. 父类引用指向子类对象【格式体现】

> *父类类型：指子类对象继承的父类类型，或者实现的父接口类型。*

1.4 多态的运行特点

调用成员变量时：编译看左边，运行看左边

调用成员方法时：编译看左边，运行看右边

代码示例：

```
Fu f = new Zi();
//编译看左边的父类中有没有name这个属性，没有就报错
//在实际运行的时候，把父类name属性的值打印出来
System.out.println(f.name);
//编译看左边的父类中有没有show这个方法，没有就报错
//在实际运行的时候，运行的是子类中的show方法
f.show();
```

1.5 多态的弊端

我们已经知道多态编译阶段是看左边父类类型的，如果子类有些独有的功能，此时多态的写法就无法访问子类独有功能了。

```
class Animal{
    public  void eat() {
        System.out.println("动物吃东西！")
    }
}
class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }

    public void catchMouse() {
        System.out.println("抓老鼠");
    }
}

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

class Test{
    public static void main(String[] args){
        Animal a = new Cat();
        a.eat();
        a.catchMouse();//编译报错，编译看左边，Animal没有这个方法
    }
}
```

1.6 引用类型转换

1.6.1 为什么要转型

多态的写法就无法访问子类独有功能了。

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误。也就是说，**不能调用**子类拥有，而父类没有的方法。编译都错误，更别说运行了。这也是多态给我们带来的一点"小麻烦"。所以，想要调用子类特有的方法，必须做向下转型。

回顾基本数据类型转换

- 自动转换: 范围小的赋值给范围大的.自动完成:double d = 5;
- 强制转换: 范围大的赋值给范围小的,强制转换:int i = (int)3.14

多态的转型分为向上转型（自动转换）与向下转型（强制转换）两种。

1.6.2 向上转型（自动转换）

- **向上转型**：多态本身是子类类型向父类类型向上转换（自动转换）的过程，这个过程是默认的。
  当父类引用指向一个子类对象时，便是向上转型。
  使用格式：

```
父类类型   变量名 = new 子类类型();
如：Animal a = new Cat();
```

原因是：父类类型相对与子类来说是大范围的类型，Animal是动物类，是父类类型。Cat是猫类，是子类类型。Animal类型的范围当然很大，包含一切动物。所以子类范围小可以直接自动转型给父类类型的变量。

1.6.3 向下转型（强制转换）

- **向下转型**：父类类型向子类类型向下转换的过程，这个过程是强制的。
  一个已经向上转型的子类对象，将父类引用转为子类引用，可以使用强制类型转换的格式，便是向下转型。

使用格式：

```
子类类型 变量名 = (子类类型) 父类变量名;
如:Aniaml a = new Cat();
   Cat c =(Cat) a;
```

1.6.4 案例演示

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误。也就是说，**不能调用**子类拥有，而父类没有的方法。编译都错误，更别说运行了。这也是多态给我们带来的一点"小麻烦"。所以，想要调用子类特有的方法，必须做向下转型。

转型演示，代码如下：

定义类：

```
abstract class Animal {
    abstract void eat();
}

class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
    public void catchMouse() {
        System.out.println("抓老鼠");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
    public void watchHouse() {
        System.out.println("看家");
    }
}
```

定义测试类：

```java
public class Test {
    public static void main(String[] args) {
        // 向上转型
        Animal a = new Cat();
        a.eat();                // 调用的是 Cat 的 eat

        // 向下转型
        Cat c = (Cat)a;
        c.catchMouse();         // 调用的是 Cat 的 catchMouse
    }
}
```

1.6.5 转型的异常

转型的过程中，一不小心就会遇到这样的问题，请看如下代码：

```java
public class Test {
    public static void main(String[] args) {
        // 向上转型
        Animal a = new Cat();
        a.eat();                // 调用的是 Cat 的 eat

        // 向下转型
        Dog d = (Dog)a;
        d.watchHouse();         // 调用的是 Dog 的 watchHouse 【运行报错】
    }
}
```

这段代码可以通过编译，但是运行时，却报出了 `ClassCastException` ，类型转换异常！这是因为，明明创建了Cat类型对象，运行时，当然不能转换成Dog对象的。

1.6.6 instanceof关键字

为了避免ClassCastException的发生，Java提供了 `instanceof` 关键字，给引用变量做类型的校验，格式如下：

```
变量名 instanceof 数据类型
如果变量属于该数据类型或者其子类类型，返回true。
如果变量不属于该数据类型或者其子类类型，返回false。
```

所以，转换前，我们最好先做一个判断，代码如下：

```
public class Test {
    public static void main(String[] args) {
        // 向上转型
        Animal a = new Cat();
        a.eat();               // 调用的是 Cat 的 eat

        // 向下转型
        if (a instanceof Cat){
            Cat c = (Cat)a;
            c.catchMouse();         // 调用的是 Cat 的 catchMouse
        } else if (a instanceof Dog){
            Dog d = (Dog)a;
            d.watchHouse();        // 调用的是 Dog 的 watchHouse
        }
    }
}
```

1.6.7 instanceof新特性

JDK14的时候提出了新特性，把判断和强转合并成了一行

```
//新特性
//先判断a是否为Dog类型，如果是，则强转成Dog类型，转换之后变量名为d
//如果不是，则不强转，结果直接是false
if(a instanceof Dog d){
    d.lookHome();
}else if(a instanceof Cat c){
    c.catchMouse();
}else{
    System.out.println("没有这个类型，无法转换");
}
```

1.7 综合练习

```
需求：根据需求完成代码：
    1.定义狗类
        属性：
            年龄，颜色
        行为：
```

```
            eat(String something)(something表示吃的东西)
            看家lookHome方法(无参数)
2.定义猫类
    属性:
        年龄，颜色
    行为:
        eat(String something)方法(something表示吃的东西)
        逮老鼠catchMouse方法(无参数)
3.定义Person类//饲养员
    属性:
        姓名，年龄
    行为:
        keepPet(Dog dog,String something)方法
            功能：喂养宠物狗，something表示喂养的东西
    行为:
        keepPet(Cat cat,String something)方法
            功能：喂养宠物猫，something表示喂养的东西
    生成空参有参构造，set和get方法
4.定义测试类(完成以下打印效果):
    keepPet(Dog dog,String somethind)方法打印内容如下:
        年龄为30岁的老王养了一只黑颜色的2岁的狗
        2岁的黑颜色的狗两只前腿死死的抱住骨头猛吃
    keepPet(Cat cat,String somethind)方法打印内容如下:
        年龄为25岁的老李养了一只灰颜色的3岁的猫
        3岁的灰颜色的猫眯着眼睛侧着头吃鱼
5.思考:
    1.Dog和Cat都是Animal的子类，以上案例中针对不同的动物，定义了不同的keepPet方法，过于繁琐，能否简化，并体会简化后的好处?
    2.Dog和Cat虽然都是Animal的子类，但是都有其特有方法，能否想办法在keepPet中调用特有方法?
```

画图分析：

根据需求完成代码：

1.定义狗类

属性：

年龄，颜色

行为：

eat(String something)(something表示吃的东西)

看家lookHome方法(无参数)

2.定义猫类

属性：

年龄，颜色

行为：

eat(String something)方法(something表示吃的东西)

逮老鼠catchMouse方法(无参数)

3.定义Person类//饲养员

属性：

姓名，年龄

行为：

keepPet(Dog dog,String something)方法

功能：喂养宠物狗，something表示喂养的东西

行为：

keepPet(Cat cat,String something)方法

功能：喂养宠物猫，something表示喂养的东西

生成空参有参构造，set和get方法

4.定义测试类(完成以下打印效果)：

keepPet(Dog dog,String somethind)方法打印内容如下：

年龄为30岁的老王养了一只黑颜色的2岁的宠物

2岁的黑颜色的狗两只前腿死死的抱住骨头猛吃

keepPet(Cat cat,String somethind)方法打印内容如下：

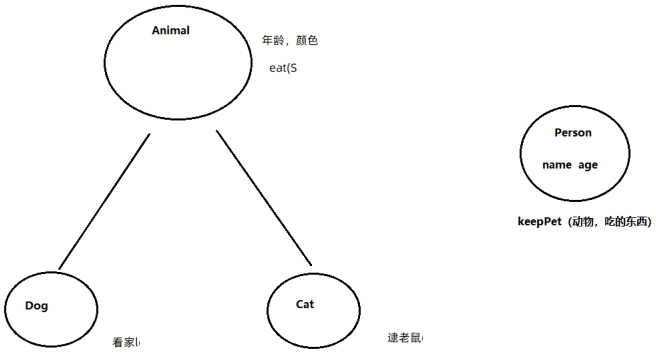
年龄为25岁的老李养了一只灰颜色的3岁的宠物

3岁的灰颜色的猫眯着眼睛侧着头吃鱼

5.思考：

1.Dog和Cat都是Animal的子类，以上案例中针对不同的动物，定义了不同的keepPet方法，过于繁琐，能否简化，并体会简化后的好处？

2.Dog和Cat虽然都是Animal的子类，但是都有其特有方法，能否想办法在keepPet中调用特有方法？



代码示例：

```java
//动物类（父类）
public class Animal {
    private int age;
    private String color;


    public Animal() {
    }

    public Animal(int age, String color) {
        this.age = age;
        this.color = color;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getColor() {
        return color;
    }

```

```
27      public void setColor(String color) {
28          this.color = color;
29      }
30
31      public void eat(String something){
32          System.out.println("动物在吃" + something);
33      }
34  }
35
36  //猫类（子类）
37  public class Cat extends Animal {
38
39      public Cat() {
40      }
41
42      public Cat(int age, String color) {
43          super(age, color);
44      }
45
46      @Override
47      public void eat(String something) {
48          System.out.println(getAge() + "岁的" + getColor() + "颜
    色的猫眯着眼睛侧着头吃" + something);
49      }
50
51      public void catchMouse(){
52          System.out.println("猫抓老鼠");
53      }
54
55  }
56
57  //狗类（子类）
58  public class Dog extends Animal {
59      public Dog() {
60      }
61
62      public Dog(int age, String color) {
63          super(age, color);
64      }
65
66      //行为
67      //eat(String something)(something表示吃的东西)
```

```
68      //看家lookHome方法(无参数)
69      @Override
70      public void eat(String something) {
71          System.out.println(getAge() + "岁的" + getColor() + "颜
    色的狗两只前腿死死的抱住" + something + "猛吃");
72      }
73
74      public void lookHome(){
75          System.out.println("狗在看家");
76      }
77  }
78
79
80  //饲养员类
81  public class Person {
82      private String name;
83      private int age;
84
85      public Person() {
86      }
87
88      public Person(String name, int age) {
89          this.name = name;
90          this.age = age;
91      }
92
93      public String getName() {
94          return name;
95      }
96
97      public void setName(String name) {
98          this.name = name;
99      }
100
101     public int getAge() {
102         return age;
103     }
104
105     public void setAge(int age) {
106         this.age = age;
107     }
108
```

```
109     //饲养狗
110    /* public void keepPet(Dog dog, String something) {
111         System.out.println("年龄为" + age + "岁的" + name + "养了
    一只" + dog.getColor() + "颜色的" + dog.getAge() + "岁的狗");
112         dog.eat(something);
113     }
114 
115     //饲养猫
116     public void keepPet(Cat cat, String something) {
117         System.out.println("年龄为" + age + "岁的" + name + "养了
    一只" + cat.getColor() + "颜色的" + cat.getAge() + "岁的猫");
118         cat.eat(something);
119     }*/
120 
121 
122     //想要一个方法，能接收所有的动物，包括猫，包括狗
123     //方法的形参：可以写这些类的父类 Animal
124     public void keepPet(Animal a, String something) {
125         if(a instanceof Dog d){
126             System.out.println("年龄为" + age + "岁的" + name +
    "养了一只" + a.getColor() + "颜色的" + a.getAge() + "岁的狗");
127             d.eat(something);
128         }else if(a instanceof Cat c){
129             System.out.println("年龄为" + age + "岁的" + name +
    "养了一只" + c.getColor() + "颜色的" + c.getAge() + "岁的猫");
130             c.eat(something);
131         }else{
132             System.out.println("没有这种动物");
133         }
134     }
135 }
136 
137 //测试类
138 public class Test {
139     public static void main(String[] args) {
140         //创建对象并调用方法
141        /* Person p1 = new Person("老王",30);
142         Dog d = new Dog(2,"黑");
143         p1.keepPet(d,"骨头");
144 
145 
146         Person p2 = new Person("老李",25);
```

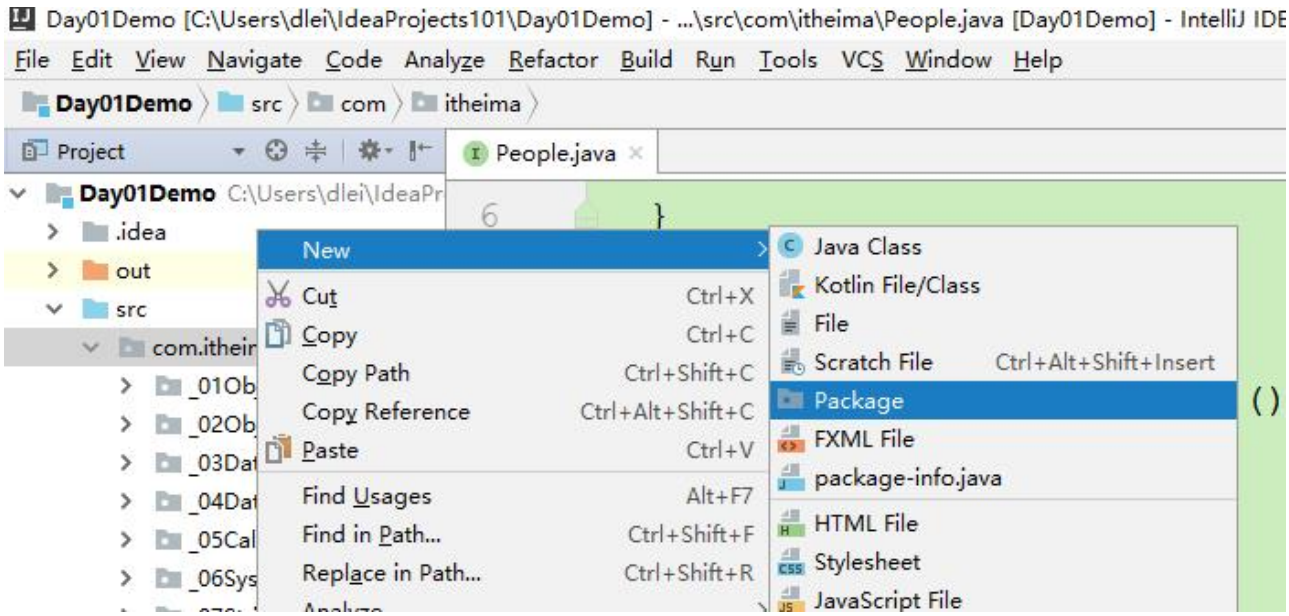
```
147         Cat c = new Cat(3,"灰");
148         p2.keepPet(c,"鱼");*/
149
150
151         //创建饲养员的对象
152         Person p = new Person("老王",30);
153         Dog d = new Dog(2,"黑");
154         Cat c = new Cat(3,"灰");
155         p.keepPet(d,"骨头");
156         p.keepPet(c,"鱼");
157
158     }
159 }
```

第二章 包

2.1 包

包在操作系统中其实就是一个文件夹。**包是用来分门别类的管理技术，不同的技术类放在不同的包下，**方便管理和维护。

在IDEA项目中，建包的操作如下：



包名的命名规范：

```
路径名.路径名.xxx.xxx
// 例如：com.itheima.oa
```

- 包名一般是公司域名的倒写。例如：黑马是www.itheima.com,包名就可以定义成com.itheima.技术名称。
- 包名必须用”.“连接。
- 包名的每个路径名必须是一个合法的标识符，而且不能是Java的关键字。

2.2 导包

什么时候需要导包？

情况一：在使用Java中提供的非核心包中的类时

情况二：使用自己写的其他包中的类时

什么时候不需要导包？

情况一：在使用Java核心包（java.lang）中的类时

情况二：在使用自己写的同一个包中的类时

2.3 使用不同包下的相同类怎么办？

假设demo1和demo2中都有一个Student该如何使用？

代码示例：

```
//使用全类名的形式即可。
//全类名：包名 + 类名
//拷贝全类名的快捷键：选中类名crtl + shift + alt + c 或者用鼠标点copy，再点击copy Reference
com.itheima.homework.demo1.Student s1 = new com.itheima.homework.demo1.Student();
com.itheima.homework.demo2.Student s2 = new com.itheima.homework.demo2.Student();
```

第三章 权限修饰符

3.1 权限修饰符

在Java中提供了四种访问权限，使用不同的访问权限修饰符修饰时，被修饰的内容会有不同的访问权限，我们之前已经学习过了public 和 private，接下来我们研究一下protected和默认修饰符的作用。

- public：公共的，所有地方都可以访问。
- protected：本类，本包，其他包中的子类都可以访问。
- 默认（没有修饰符）：本类，本包可以访问。

  注意：默认是空着不写，不是default
- private：私有的，当前类可以访问。
  `public > protected > 默认 > private`

3.2 不同权限的访问能力

	PUBLIC	PROTECTED	默认	PRIVATE
同一类中	√	√	√	√
同一包中的类	√	√	√	
不同包的子类	√	√		
不同包中的无关类	√			

可见，public具有最大权限。private则是最小权限。

编写代码时，如果没有特殊的考虑，建议这样使用权限：

- 成员变量使用`private`，隐藏细节。
- 构造方法使用`public`，方便创建对象。
- 成员方法使用`public`，方便调用方法。

> *小贴士：不加权限修饰符，就是默认权限*

第四章 final关键字

4.1 概述

学习了继承后，我们知道，子类可以在父类的基础上改写父类内容，比如，方法重写。

如果有一个方法我不想别人去改写里面内容，该怎么办呢？

Java提供了`final`关键字，表示修饰的内容不可变。

- **final**： 不可改变，最终的含义。可以用于修饰类、方法和变量。
  - 类：被修饰的类，不能被继承。
  - 方法：被修饰的方法，不能被重写。
  - 变量：被修饰的变量，有且仅能被赋值一次。

4.2 使用方式

4.2.1 修饰类

final修饰的类，不能被继承。

格式如下：

```
final class 类名 {
}
```

代码:

```
final class Fu {
}
// class Zi extends Fu {} // 报错,不能继承final的类
```

查询API发现像`public final class String`、`public final class Math`、`public final class Scanner`等，很多我们学习过的类，都是被final修饰的，目的就是供我们使用，而不让我们所以改变其内容。

4.2.2 修饰方法

final修饰的方法，不能被重写。
格式如下：

```
修饰符 final 返回值类型 方法名(参数列表){
    //方法体
}
```

代码:

```
class Fu2 {
    final public void show1() {
        System.out.println("Fu2 show1");
    }
    public void show2() {
        System.out.println("Fu2 show2");
    }
}

class Zi2 extends Fu2 {
//  @Override
//  public void show1() {
//      System.out.println("Zi2 show1");
//  }
    @Override
    public void show2() {
        System.out.println("Zi2 show2");
    }
}
```

4.2.3 修饰变量-局部变量

1. 局部变量——基本类型
   基本类型的局部变量，被final修饰后，只能赋值一次，不能再更改。代码如下：

```
public class FinalDemo1 {
    public static void main(String[] args) {
        // 声明变量，使用final修饰
        final int a;
        // 第一次赋值
```

```
        a = 10;
        // 第二次赋值
        a = 20; // 报错,不可重新赋值

        // 声明变量，直接赋值，使用final修饰
        final int b = 10;
        // 第二次赋值
        b = 20; // 报错,不可重新赋值
    }
}
```

思考，下面两种写法，哪种可以通过编译？

写法1：

```
final int c = 0;
for (int i = 0; i < 10; i++) {
    c = i;
    System.out.println(c);
}
```

写法2：

```
for (int i = 0; i < 10; i++) {
    final int c = i;
    System.out.println(c);
}
```

根据 `final` 的定义，写法1报错！写法2，为什么通过编译呢？因为每次循环，都是一次新的变量c。这也是大家需要注意的地方。

4.2.4 修饰变量-成员变量

成员变量涉及到初始化的问题，初始化方式有显示初始化和构造方法初始化，只能选择其中一个：

- 显示初始化(在定义成员变量的时候立马赋值)（常用）；

```
public class Student {
    final int num = 10;
}
```

- 构造方法初始化(在构造方法中赋值一次)（不常用，了解即可）。

  注意：每个构造方法中都要赋值一次！

```
public class Student {
    final int num = 10;
    final int num2;

    public Student() {
        this.num2 = 20;
//      this.num2 = 20;
    }

     public Student(String name) {
        this.num2 = 20;
//      this.num2 = 20;
    }
}
```

> *被final修饰的常量名称，一般都有书写规范，所有字母都大写。*