day05 【多态、内部类、权限修饰符、代码块】

今日内容

- 多态
- 内部类
- 权限修饰符
- 代码块

教学目标

- [ ] 能够说出多态的前提
- [ ] 能够写出多态的格式
- [ ] 能够理解多态向上转型和向下转型
- [ ] 能够说出内部类概念
- [ ] 能够理解匿名内部类的编写格式
- [ ] 能够说出每种权限修饰符的作用

第一章 多态

1.1 概述

引入

多态是继封装、继承之后，面向对象的第三大特性。

生活中，比如跑的动作，小猫、小狗和大象，跑起来是不一样的。再比如飞的动作，昆虫、鸟类和飞机，飞起来也是不一样的。可见，同一行为，通过不同的事物，可以体现出来的不同的形态。多态，描述的就是这样的状态。

定义

- **多态**：是指同一行为，具有多个不同表现形式。

前提【重点】

1. 继承或者实现【二选一】
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】

1.2 多态的体现

多态体现的格式：

```
父类类型 变量名 = new 子类对象;
变量名.方法名();
```

> 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。

代码如下：

```
Fu f = new Zi();
f.method();
```

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；如果有，执行的是子类重写后方法。

代码如下：

定义父类：

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

定义子类：

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

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

定义测试类：

```
public class Test {
    public static void main(String[] args) {
        // 多态形式，创建对象
        Animal a1 = new Cat();
        // 调用的是 Cat 的 eat
        a1.eat();

        // 多态形式，创建对象
        Animal a2 = new Dog();
        // 调用的是 Dog 的 eat
        a2.eat();
    }
}
```

> 多态在代码中的体现为父类引用指向子类对象。

1.3 多态的好处

实际开发的过程中，父类类型作为方法形式参数，传递子类对象给方法，进行方法的调用，更能体现出多态的扩展性与便利。代码如下：

定义父类：

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

定义子类：

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

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

定义测试类：

```
public class Test {
    public static void main(String[] args) {
        // 多态形式，创建对象
        Cat c = new Cat();
        Dog d = new Dog();

        // 调用showCatEat
        showCatEat(c);
        // 调用showDogEat
        showDogEat(d);

        /*
        以上两个方法，均可以被showAnimalEat(Animal a)方法所替代
        而执行效果一致
        */
        showAnimalEat(c);
        showAnimalEat(d);
    }

    public static void showCatEat (Cat c){
        c.eat();
    }

    public static void showDogEat (Dog d){
        d.eat();
    }

    public static void showAnimalEat (Animal a){
```

```
        a.eat();
    }
}
```

由于多态特性的支持，showAnimalEat方法的Animal类型，是Cat和Dog的父类类型，父类类型接收子类对象，当然可以把Cat对象和Dog对象，传递给方法。

当eat方法执行时，多态规定，执行的是子类重写的方法，那么效果自然与showCatEat、showDogEat方法一致，所以showAnimalEat完全可以替代以上两方法。

不仅仅是替代，在扩展性方面，无论之后再多的子类出现，我们都不需要编写showXxxEat方法了，直接使用showAnimalEat都可以完成。

所以，多态的好处，体现在，可以使程序编写的更简单，并有良好的扩展。

> 小结：多态的好处是提高程序的灵活性，扩展性

1.4 引用类型转换

多态的转型分为向上转型与向下转型两种：

向上转型

- **向上转型**：多态本身是子类类型向父类类型向上转换的过程，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();
```

向下转型

- **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:Cat c =(Cat) a;
```

为什么要转型

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：

```
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
    }
}
```

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：

```
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。这两个类型并没有任何继承关系，不符合类型转换的定义。

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，返回true。
如果变量不属于该数据类型，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
        }
    }
}
```

> 小结：多态向上转型是将子类类型转成父类类型，多态向下转型是将父类类型转成子类类型。

第二章 内部类

2.1 概述

什么是内部类

将一个类A定义在另一个类B里面，里面的那个类A就称为**内部类**，B则称为**外部类。**

2.2 成员内部类

- **成员内部类**：定义在**类中方法外**的类。

定义格式：

```
class 外部类 {
    class 内部类{

    }
}
```

在描述事物时，若一个事物内部还包含其他事物，就可以使用内部类这种结构。比如，汽车类 `Car` 中包含发动机类 `Engine`，这时，`Engine` 就可以使用内部类来描述，定义在成员位置。

代码举例：

```
class Car { //外部类
    class Engine { //内部类

    }
}
```

访问特点

- 内部类可以直接访问外部类的成员，包括私有成员。
- 外部类要访问内部类的成员，必须要建立内部类的对象。

创建内部类对象格式：

```
外部类名.内部类名 对象名 = new 外部类型().new 内部类型();
```

访问演示，代码如下：

定义类：

```
public class Person {
    private  boolean live = true;
    class Heart {
        public void jump() {
            // 直接访问外部类成员
            if (live) {
                System.out.println("心脏在跳动");
            } else {
                System.out.println("心脏不跳了");
            }
        }
    }

    public boolean isLive() {
        return live;
    }

    public void setLive(boolean live) {
        this.live = live;
    }

}
```

定义测试类：

```
public class InnerDemo {
    public static void main(String[] args) {
        // 创建外部类对象
        Person p  = new Person();
        // 创建内部类对象
        Person.Heart heart = p.new Heart();
        // 调用内部类方法
        heart.jump();
        // 调用外部类方法
```

```
            p.setLive(false);
            // 调用内部类方法
            heart.jump();
        }
}
输出结果:
心脏在跳动
心脏不跳了
```

> 内部类仍然是一个独立的类，在编译之后会内部类会被编译成独立的.class文件，但是前面冠以外部类的类名和$符号 。
>
> 比如，Person$Heart.class

> 小结：内部类是定义在一个类中的类。

2.3 匿名内部类

- **匿名内部类**：是内部类的简化写法。它的本质是一个`带具体实现的` `父类或者父接口的` `匿名的` **子类对象。**

开发中，最常用到的内部类就是匿名内部类了。以接口举例，当你使用一个接口时，似乎得做如下几步操作，

1. 定义子类
2. 重写接口中的方法
3. 创建子类对象
4. 调用重写后的方法

我们的目的，最终只是为了调用方法，那么能不能简化一下，把以上四步合成一步呢？匿名内部类就是做这样的快捷方式。

前提

存在一个**类或者接口**，这里的**类可以是具体类也可以是抽象类。**

格式

```
new 父类名或者接口名(){
    // 方法重写
    @Override
    public void method() {
        // 执行语句
    }
};
```

使用方式

以接口为例，匿名内部类的使用，代码如下：

定义接口：

```
public abstract class FlyAble{
    public abstract void fly();
}
```

匿名内部类可以通过多态的形式接受

```
public class InnerDemo01 {
    public static void main(String[] args) {
        /*
            1.等号右边:定义并创建该接口的子类对象
            2.等号左边:是多态,接口类型引用指向子类对象
       */
        FlyAble  f = new FlyAble(){
            public void fly() {
                System.out.println("我飞了~~~");
            }
        };
    }
}
```

匿名内部类直接调用方法

```
public class InnerDemo02 {
    public static void main(String[] args) {
        /*
            1.等号右边:定义并创建该接口的子类对象
            2.等号左边:是多态,接口类型引用指向子类对象
       */
        new FlyAble(){
            public void fly() {
                System.out.println("我飞了~~~");
            }
        }.fly();
    }
}
```

方法的形式参数是接口或者抽象类时，也可以将匿名内部类作为参数传递

```
public class InnerDemo3 {
    public static void main(String[] args) {
        /*
        1.等号右边:定义并创建该接口的子类对象
        2.等号左边:是多态,接口类型引用指向子类对象
       */
        FlyAble  f = new FlyAble(){
            public void fly() {
                System.out.println("我飞了~~~");
            }
        };
        // 将f传递给showFly方法中
        showFly(f);
    }
    public static void showFly(FlyAble f) {
        f.fly();
    }
```

```
}
```

以上可以简化，代码如下：

```
public class InnerDemo2 {
    public static void main(String[] args) {
        /*
        创建匿名内部类,直接传递给showFly(FlyAble f)
        */
        showFly( new FlyAble(){
            public void fly() {
                System.out.println("我飞了~~~");
            }
        });
    }

    public static void showFly(FlyAble f) {
        f.fly();
    }
}
```

> 小结：匿名内部类做的事情是创建一个类的子类对象

第三章 权限修饰符

3.1 概述

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，

- public：公共的
- protected：受保护的
- (空的)：默认的
- private：私有的

3.2 不同权限的访问能力

	public	protected	(空的)	private
同一类中	√	√	√	√
同一包中(子类与无关类)	√	√	√	
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。

第四章 代码块

4.1 构造代码块

- 构造代码块：定义在成员位置的代码块{}
  - 执行：每次创建对象都会执行构造代码块

```
public class Person{
    {
        构造代码块执行了
    }
}
```

4.2 静态代码块

- **静态代码块**：定义在成员位置，使用static修饰的代码块{ }。
  - 位置：类中方法外。
  - 执行：随着类的加载而执行且执行一次，优先构造方法的执行。

格式：

```
public class Person {
    private String name;
    private int age;
 //静态代码块
    static{
        System.out.println("静态代码块执行了");
    }
}
```