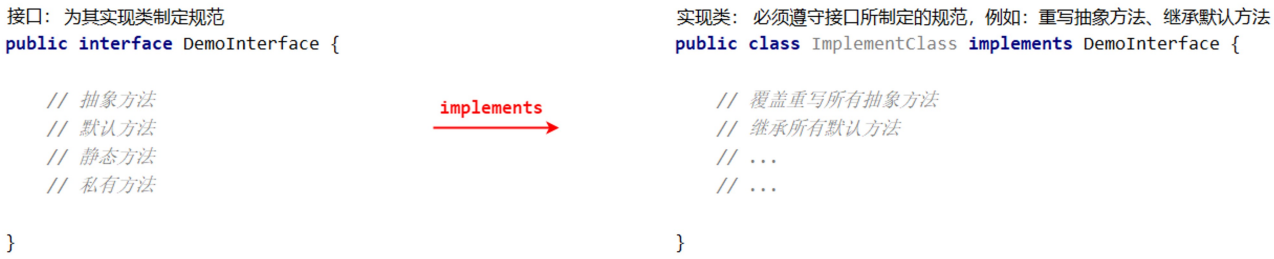
day10_接口、多态

一.接口

接口就是多个类的公共规范，是一种引用数据类型，是方法的集合。如果说类的内部封装了成员变量、成员方法，那么接口的内部主要就是封装了方法，主要包含以下内容：

- JDK7：常量、抽象方法（主要）
- JDK8：默认方法、静态方法（额外增加）
- JDK9：私有方法（额外增加）



(1).接口的定义：与定义类的方式类似，但是使用 interface 关键字，同样也会被编译成 .class字节码文件。

(2).接口的使用：接口不能直接使用，必须有一个 "实现类" 来 "实现" 该接口。一个实现接口的类，叫接口的实现类或子类，实现类必须实现（覆盖重写）接口中所有的抽象方法，否则它必须是一个抽象类。然后就可以创建实现类的对象，调用相应的方法了。

1.接口中的抽象方法

在任何版本的JDK中，接口当中都能定义抽象方法。定义格式如下：

```
public abstract 返回值类型 方法名称(参数列表);
```

(1).接口当中的抽象方法，修饰符必须是两个固定的关键字：public abstract。这两个修饰符，可以选择性地省略。(IDEA中显示为灰色即可省略)
(2).接口的实现类必须覆盖重写接口中的所有抽象方法，除非该实现类本身就是抽象类。此时可以认为，实现类会继承来自接口的所有抽象方法，所以当实现类不是抽象类的时候，必须覆盖重写接口当中的所有抽象方法。（覆盖重写抽象方法：去掉abstract，补充方法体）

2.接口中的默认方法

从Java 8开始，接口里允许定义默认方法。定义格式如下：

```
public default 返回值类型 方法名称(参数列表) {
        方法体;
}
```

(1).接口当中的默认方法，是为了解决接口升级的问题。当我们想更新接口，在接口中添加新的抽象方法时，所有的实现类就要对新添加的抽象方法进行覆盖重写，比较麻烦。所以此时引进了默认方法来对接口进行更新，因为接口中定义的默认方法是会被实现类继承下去的。

(2).接口中的默认方法必须使用 public default 作为修饰符，其中 public 可省略。

(3).接口当中的默认方法会被实现类所继承，所以接口的各个实现类都会拥有该默认方法。实现类可以对继承下来的默认方法进行覆盖重写，也可以不对其进行覆盖重写。（覆盖重写默认方法：去掉 default，补充方法体）

3.接口中的静态方法

从Java 8开始，接口当中允许定义静态方法。定义格式如下：

```
public static 返回值类型 方法名称(参数列表) {
        方法体;
}
```

(1).接口中的静态方法不会被其实现类所继承下去，所以不能通过接口实现类的对象来调用接口当中的静态方法。在其他类中，都可以使用接口名来调用静态方法：接口名.静态方法名(参数)；由此可见，接口的静态方法与类的静态方法类似，都是属于类、接口本身的一个静态方法。

(2).接口中的静态方法必须使用 public static 作为修饰符，其中 public 可省略。

4.接口中的私有方法

前面学习了接口的默认方法、静态方法，如果两个默认方法（静态方法）中有重复代码的时候，我们需要提取出来一个公共的方法来封装这部分重复的代码，以此来实现代码复用。但是这个公共方法不应该让实现类或者其他类使用，应该仅限于本接口内部使用，应该是私有化的。所以从JDK9开始，接口当中可以定义私有方法：

(1).普通私有方法，解决多个默认方法之间有重复代码的问题。定义格式如下：

```
private 返回值类型 方法名称(参数列表) {
        方法体;
}
```

(2).静态私有方法，解决多个静态方法之间有重复代码的问题。定义格式如下：

```
private static 返回值类型 方法名称(参数列表) {
        方法体;
}
```

由于静态方法中不能使用非静态的内容，所以 "被抽取的方法中" 一旦有一个是静态方法，那么抽取出来的公共方法必须是 "静态私有方法"。如果 "被抽取的方法中" 都是默认方法，那么抽取出来的是 "普通私有方法"、"静态私有方法" 均可。 （私有方法仅限于本接口内部使用！）

5.接口中的常量

在任何版本的JDK中，接口当中都可以定义"成员变量"，但是必须使用 public static final 这三个关键字进行修饰。从效果上看，这其实就是接口的 "常量"。定义格式如下：

```
public static final 数据类型 常量名称 = 数据值;
```

(1).接口当中的常量，可以省略 public static final 修饰符，不写也是默认这三个修饰符。
(2).接口当中的常量，必须进行初始化赋值。并且一旦使用final关键字进行修饰，说明不可改变。
(3).由于被static关键字进行修饰：a.接口中的常量不会被实现类所继承　b.在其他类中，使用接口名即可调用：接口名.常量名
(4).接口中常量名称的命名规则：使用完全大写的字母，用下划线进行分隔单词。

#.注意：

(1).接口中不能定义静态代码块或者构造方法。

(2).接口的多实现
在继承体系中，一个类只能继承一个父类。而对于接口而言，一个类是可以同时实现多个接口的，这叫做接口的多实现。并且一个类在继承一个父类的同时也可以实现多个接口。格式如下：

```
public class 实现类名 [extends 父类名] implements InterfaceA, InterfaceB ... {
        // 覆盖重写所有抽象方法
}
```

a.抽象方法：实现多个接口时，实现类必须重写所有接口的所有抽象方法。如果抽象方法有重名的，只需要重写一次。代码如下：

```
public interface A {

    public abstract void methodA();

    public abstract void method();

}



public interface B {

    public abstract void methodB();

    public abstract void method();

}
```

```
public class C implements A, B {

    @Override
    public void methodA() {
        System.out.println("重写接口A的抽象方法");
    }

    @Override
    public void methodB() {
        System.out.println("重写接口B的抽象方法");
    }

    @Override
    public void method() {
        System.out.println("只需要重写一次接口A和B重名的抽象方法");
    }

}
```

b.默认方法：实现多个接口时，实现类可以继承所有接口中的默认方法。实现类可以对默认方法重写，也不可以不重写默认方法。如果默认方法有重名的，必须重写一次。代码如下：

```java
public interface A {

    public default void methodA() {
    }

    public default void method() {
    }

}

public interface B {

    public default void methodB() {
    }

    public default void method() {
    }

}
```

```java
public class C implements A, B {

    @Override
    public void method() {
        System.out.println("必须重写接口A和B重名的默认方法");
    }

}
```

c.静态方法：实现多个接口时，如果存在同名的静态方法并不会冲突，因为静态方法不能被实现类继承，只能通过各自接口名访问静态方法。

d.私有方法：实现多个接口时，如果存在同名的私有方法并不会冲突，因为私有方法只能在本接口内部使用。

e.优先级问题：既继承一个父类，又实现多个接口时，父类中的成员方法与接口中的默认方法重名，子类就近选择执行父类的成员方法。即：从父类继承下来的成员方法比从接口继承下来的默认方法的优先级高。代码如下：

```java
public class Fu {

    public void method() {
        System.out.println("父类成员方法");
    }

}

public interface A {

    public default void method() {
        System.out.println("接口默认方法");
    }

}
```

```java
public class Zi extends Fu implements A {
    // 未重写method()方法，一旦重写，则二者均被覆盖重写
}
```

```java
public class DemoMain {
    public static void main(String[] args) {

        Zi zi = new Zi();
        zi.method();  // 父类成员方法

    }
}
```

(3).接口的多继承（了解内容）

类与类之间是单继承的，类与接口之间是可以多实现的。与类的继承相似，接口也可以使用extends关键字进行继承，但接口与接口之间是可以多继承的，即一个接口可以继承多个父接口。子接口会继承多个父接口的所有内容，下面仅以抽象方法、默认方法来举例说明。

a.父接口中重名的抽象方法在子接口中并不冲突，子接口的实现类只需实现一次父接口中重名的抽象方法即可

b.父接口中重名的默认方法必须在子接口中进行覆盖重写

(实现类对接口默认方法的重写必须要去掉default，但子接口对父接口默认方法的覆盖重写default不能去掉，在子接口中还是默认方法)

```java
public interface A {

    public abstract void methodA();

    // 重名的抽象方法
    public abstract void method();

    // 重名的默认方法
    public default void methodDefault(){
        System.out.println("AAA");
    }
}

public interface B {

    public abstract void methodB();

    // 重名的抽象方法
    public abstract void method();

    // 重名的默认方法
    public default void methodDefault(){
        System.out.println("BBB");
    }
}
```

```java
public interface C extends A, B {

    /*
    现在子接口中拥有以下继承过来的抽象方法：
    public abstract void methodA();
    public abstract void methodB();
    public abstract void method(); // 重名的抽象方法并不冲突(相当于合二为一)
     */

    // 重名的默认方法必须在子接口中进行覆盖重写
    @Override
    public default void methodDefault() {
        System.out.println("CCC");
    }
}
```

```java
public class CImpl implements C {

    @Override
    public void methodA() {

    }

    @Override
    public void methodB() {
```

```
    public default void methodDefault(){
        System.out.println("BBB");
    }
}
```

```
    }

    @Override
    public void methodB() {

    }

    // 子接口的实现类只需实现一次父接口中重名的抽象方法即可
    @Override
    public void method() {

    }
}
```

二.多态

多态是继封装、继承之后，面向对象的第三大特性。生活中，比如动物跑的动作，小猫、小狗和大象，跑的动作是不一样的。再比如动物飞的动作，昆虫、老鹰，飞的动作也是不一样的。可见，同一行为通过不同的事物，可以体现出不同的形态。多态，描述的就是同一行为，在不同事物(不同子类的对象)的身上具有多种不同的表现形式。

1.多态的用法

我们说多态，是同一行为在不同子类的对象上有多种不同的表现形式。那么该如何在代码中体现这一特性呢？首先由继承extends、接口实现implements产生的父子类关系是实现多态的前提。其实在代码中体现多态，就是一句话：父类引用指向子类对象。

父类名称 对象名 = new 子类名称();　　接口名称 对象名 = new 实现类名称();

当我们使用多态的方式调用成员方法时：首先检查父类中是否有该方法。如果没有，则编译报错；如果有，那么再检查子类是否覆盖重写了该方法，如果子类覆盖重写了该方法，则执行的是子类重写后方法，否则就执行父类方法。

由此可以看出，我们只使用父类引用，不需要创建各个子类对象，就可以展现出同一行为在不同子类的各种表现形式。而且此时也不用关心各个子类是如何覆盖重写父类方法的，只需要使用父类引用调用父类方法就行了。

所以什么是多态呢？就是只使用父类引用调用父类方法，就可以访问到各个子类覆盖重写后的方法，使得同一行为在各个子类中呈现出各种各样的状态，即是多态。

#.注意:

a.成员变量的访问特点：由于成员变量没有覆盖重写的概念，所以使用多态的方式调用成员变量时，无论子类是否有重名的成员变量，只会访问父类的成员变量，永远不会访问子类的成员变量。

b.成员方法的访问特点：当子类中有重名但不是覆盖重写的方法时，只会访问父类的成员方法，不会访问子类的成员方法。

2.使用多态的好处

(1).正如前面所说，使用多态的方式创建对象。我们只需使用父类引用来调用父类方法，即可访问到各个子类覆盖重写后的方法，此时不用关心子类方法是如何实现的，只需要关注父类的方法即可。(多态是 "向后兼容"：之前都是从子类的角度向上找父类，多态是从父类的角度向下看子类)

(2).实际开发的过程中，父类类型作为方法的形式参数，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。
　　如下例：只使用showAnimalEat()方法即可代替上述两个showXxxEat()方法。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()方法，将子类对象传递进来即可。

3.引用类型转换

类作为一种引用数据类型，也是可以进行数据类型转换的，但是仅限于父、子类之间进行类型转换，称为引用类型转换。分为以下两种：

(1).向上转型：子类类型向上转换为父类类型的过程，叫做向上转型。这个过程是默认的，当父类引用指向一个子类对象时，即多态便是向上转型。

父类名称 对象名 = new 子类名称();　　Animal a = new Cat();

(2).向下转型：一个已经向上转型的子类对象，将转型后的父类引用强制转换为子类引用，叫做向下转型。

子类类型 变量名 = (子类类型) 父类变量名;　　Cat c = (Cat) a;

#.注意:

a.我们为什么要使用转型呢？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。这也是多态给我们带来的一点"小麻烦"。所以，要想使用父类引用来调用子类特有的方法，父类引用必须做向下转型。

b.转型的过程中，一不小心就会遇到转型异常的问题。如下：

```
Animal a = new Cat();
Dog d = (Dog) a;
d.watchHouse();
```

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，明明创建的是Cat类型对象，运行时，当然不能转换成Dog对象。这两个类型并没有任何继承关系，不符合类型转换的定义。为了避免类型转换异常的发生，Java提供了 instanceof 关键字，给引用变量做类型的校验，格式如下：

```
引用变量名 instanceof 类名
```

如果引用变量指向该类的对象，返回true；如果引用变量不指向该类的对象，返回false。所以在进行向下转型时，最好先做一个类型判断，代码如下：

```
Animal a = new Cat();

if (a instanceof Cat) {
    Cat c = (Cat) a;
    c.catchMouse();
}

if (a instanceof Dog) {
    Dog d = (Dog) a;
    d.watchHome();
}
```

三.接口、多态的综合案例