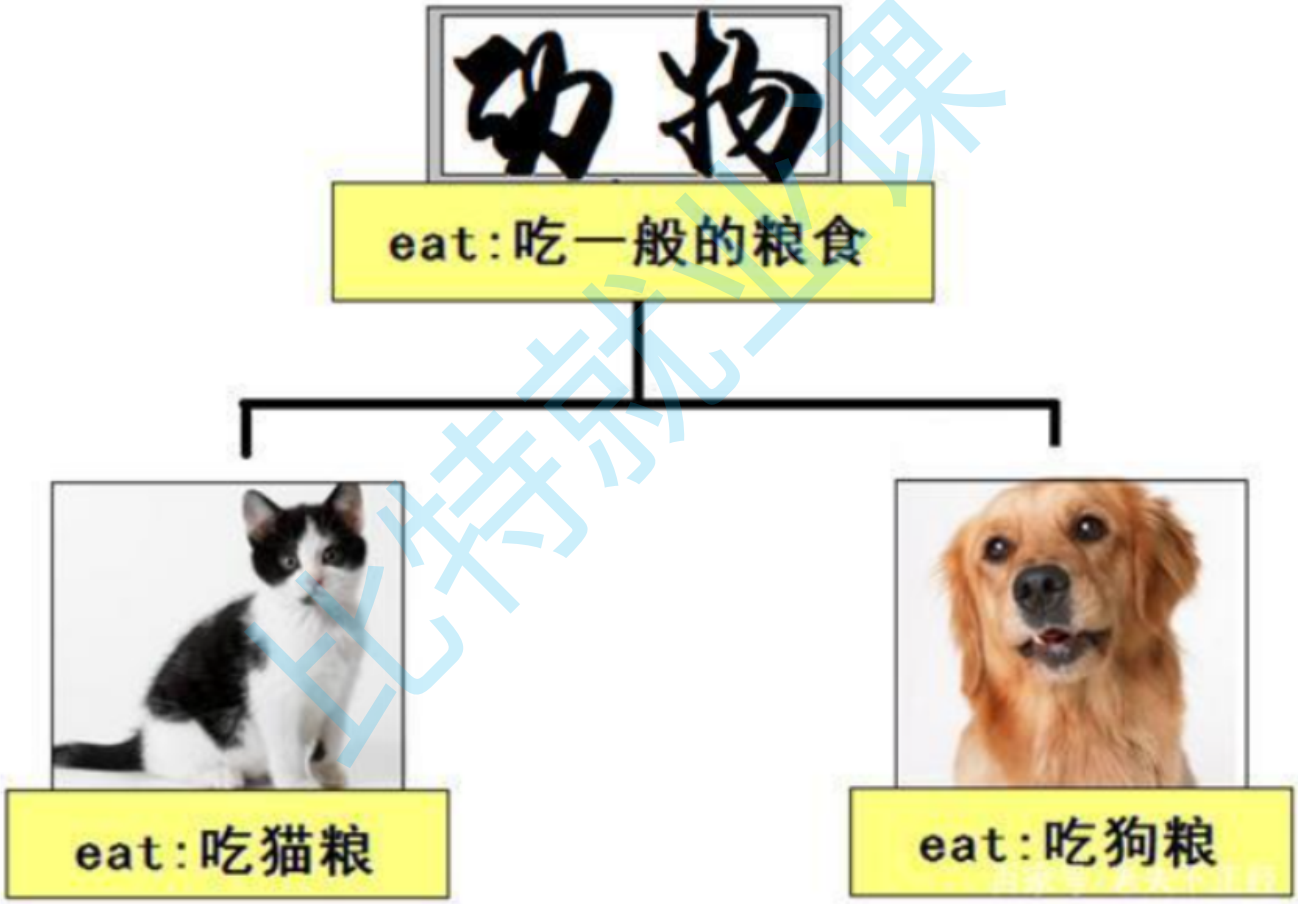
12.多态

【本节目标】

1. 多态

1. 多态的概念

多态的概念：通俗来说，就是多种形态，**具体点就是去完成某个行为，当不同的对象去完成时会产生出不同的状态。**



总的来说：同一件事情，发生在不同对象身上，就会产生不同的结果。

1.1 多态实现条件

在Java中要实现多态，必须要满足如下几个条件，缺一不可：

1. **必须在继承体系下**
2. **子类必须要对父类中方法进行重写**
3. **通过父类的引用调用重写的方法**

多态体现：在代码运行时，当传递不同类对象时，会调用对应类中的方法。

```java
public class Animal {
    String name;
    int age;

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

    public void eat(){
        System.out.println(name + "吃饭");
    }
}

public class Cat extends Animal{
    public Cat(String name, int age){
        super(name, age);
    }

    @Override
    public void eat(){
        System.out.println(name+"吃鱼~~~");
    }
}

public class Dog extends Animal {
    public Dog(String name, int age){
        super(name, age);
    }

    @Override
    public void eat(){
        System.out.println(name+"吃骨头~~~");
    }
}

//////////////////////////////分割线//////////////////////////////////////////

public class TestAnimal {
    // 编译器在编译代码时，并不知道要调用Dog 还是 Cat 中eat的方法
    // 等程序运行起来后，形参a引用的具体对象确定后，才知道调用那个方法
    // 注意：此处的形参类型必须时父类类型才可以
    public static void eat(Animal a){
        a.eat();
    }
```

```
46
47      public static void main(String[] args) {
48          Cat cat = new Cat("元宝",2);
49          Dog dog = new Dog("小七", 1);
50
51          eat(cat);
52          eat(dog);
53      }
54  }
55
56  运行结果:
57  元宝吃鱼~~~
58  元宝正在睡觉
59  小七吃骨头~~~
60  小七正在睡觉
```

在上述代码中, 分割线上方的代码是 **类的实现者** 编写的, 分割线下方的代码是 **类的调用者** 编写的.

当类的调用者在编写 `eat` 这个方法的时候, 参数类型为 Animal (父类), 此时在该方法内部并**不知道, 也不关注**当前的 a 引用指向的是哪个类型(哪个子类)的实例. 此时 a这个引用调用 eat方法可能会有多种不同的表现(和 a 引用的实例相关), 这种行为就称为 **多态**.

1.2 重写

重写(override)：也称为覆盖。重写是子类对父类非静态、非private修饰，非final修饰，非构造方法等的实现过程进行重新编写, **返回值和形参都不能改变。即外壳不变，核心重写！**重写的好处在于子类可以根据需要，定义特定于自己的行为。 也就是说子类能够根据需要实现父类的方法。

【方法重写的规则】

- 子类在重写父类的方法时，一般必须与父类方法原型一致： 返回值类型 方法名 (参数列表) 要完全一致
- 被重写的方法返回值类型可以不同，但是必须是具有父子关系的
- 访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类方法被public修饰，则子类中重写该方法就不能声明为 protected
- 父类被static、private修饰的方法、构造方法都不能被重写。
- 重写的方法, 可以使用 `@Override` 注解来显式指定. 有了这个注解能帮我们进行一些合法性校验. 例如不小心将方法名字拼写错了 (比如写成 aet), 那么此时编译器就会发现父类中没有 aet 方法, 就会编译报错, 提示无法构成重写.

【重写和重载的区别】

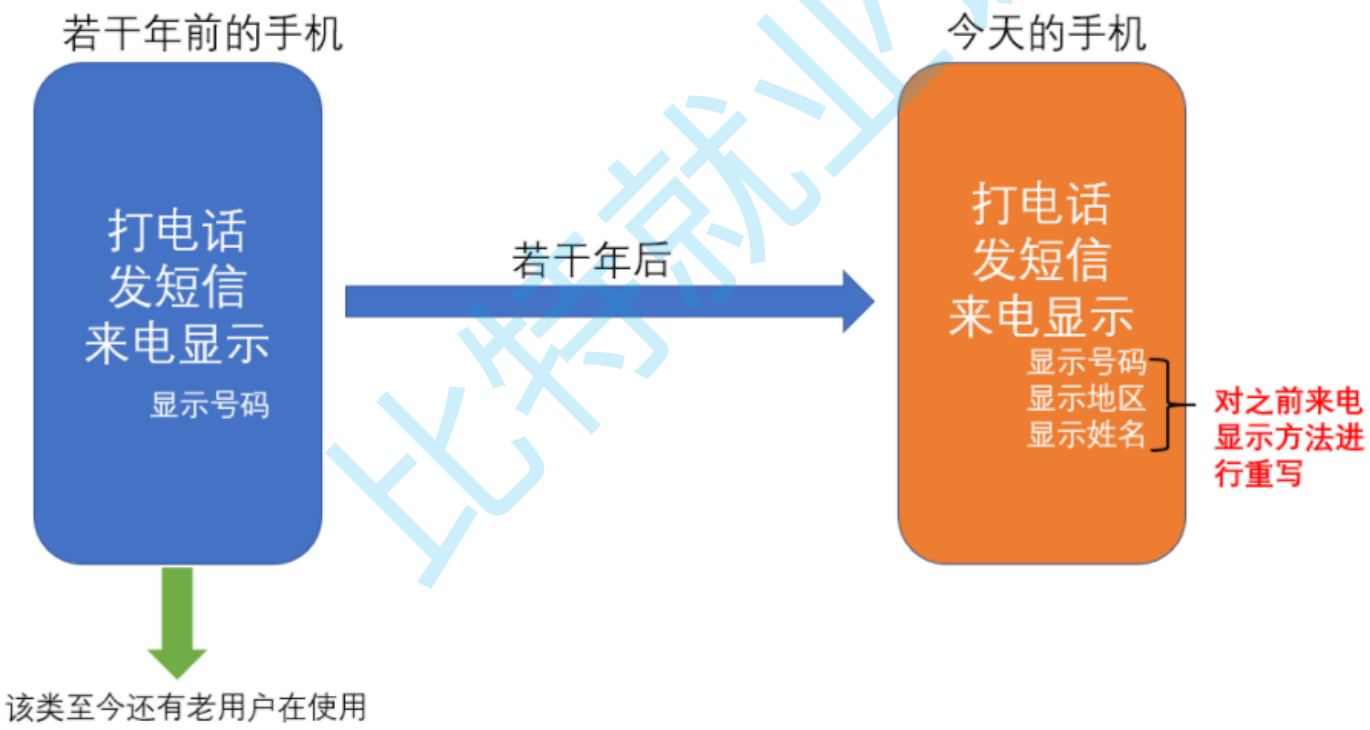
区别点	重写(override)	重载(override)
参数列表	一定不能修改	必须修改
返回类型	一定不能修改【除非可以构成父子类关系】	可以修改
访问限定符	一定不能做更严格的限制（可以降低限制）	可以修改

即：方法重载是一个类的多态性表现,而方法重写是子类与父类的一种多态性表现。

【重写的设计原则】

对于已经投入使用的类，尽量不要进行修改。最好的方式是：重新定义一个新的类，来重复利用其中共性的内容，并且添加或者改动新的内容。

例如：若干年前的手机，只能打电话，发短信，来电显示只能显示号码，而今天的手机在来电显示的时候，不仅仅可以显示号码，还可以显示头像，地区等。在这个过程当中，我们**不应该在原来老的类上进行修改，因为原来的类，可能还在有用户使用**，正确做法是：**新建一个新手机的类，对来电显示这个方法重写就好了，这样就达到了我们当今的需求了**。



静态绑定：也称为前期绑定(早绑定)，即在编译时，根据用户所传递实参类型就确定了具体调用那个方法。典型代表函数重载。

动态绑定：也称为后期绑定(晚绑定)，即在编译时，不能确定方法的行为，需要等到程序运行时，才能够确定具体调用那个类的方法。

1.3 向上转型和向下转型

1.3.1 向上转型

向上转型：实际就是创建一个子类对象，将其当成父类对象来使用。

语法格式：父类类型 对象名 = new 子类类型()

```
Animal animal = new Cat("元宝",2);
```

animal是父类类型，但可以引用一个子类对象，因为是从小范围向大范围的转换

【使用场景】

1. 直接赋值
2. 方法传参
3. 方法返回

```
public class TestAnimal {
    // 2. 方法传参：形参为父类型引用，可以接收任意子类的对象
    public static void eatFood(Animal a){
        a.eat();
    }

    // 3. 作返回值：返回任意子类对象
    public static Animal buyAnimal(String var){
        if("狗".equals(var) ){
            return new Dog("狗狗",1);
        }else if("猫" .equals(var)){
            return new Cat("猫猫", 1);
        }else{
            return null;
        }
    }

    public static void main(String[] args) {
        Animal cat = new Cat("元宝",2);   // 1. 直接赋值：子类对象赋值给父类对象
        Dog dog = new Dog("小七", 1);

        eatFood(cat);
        eatFood(dog);

        Animal animal = buyAnimal("狗");
        animal.eat();

        animal = buyAnimal("猫");
        animal.eat();
    }
}
```

向上转型的优点：让代码实现更简单灵活。

向上转型的缺陷：不能调用到子类特有的方法。

1.3.2 向下转型

将一个子类对象经过向上转型之后当成父类方法使用，再无法调用子类的方法，但有时候可能需要调用子类特有的方法，此时：将父类引用再还原为子类对象即可，即向下转换。


```java
public class TestAnimal {
    public static void main(String[] args) {
        Cat cat = new Cat("元宝",2);
        Dog dog = new Dog("小七", 1);

        // 向上转型
        Animal animal = cat;
        animal.eat();
        animal = dog;
        animal.eat();

        // 编译失败，编译时编译器将animal当成Animal对象处理
        // 而Animal类中没有bark方法，因此编译失败
        // animal.bark();

        // 向上转型
        // 程序可以通过编程，但运行时抛出异常---因为：animal实际指向的是狗
```

```
        // 现在要强制还原为猫，无法正常还原，运行时抛出：ClassCastException
        cat = (Cat)animal;
        cat.mew();

        // animal本来指向的就是狗，因此将animal还原为狗也是安全的
        dog = (Dog)animal;
        dog.bark();
    }
}
```

向下转型用的比较少，而且不安全，万一转换失败，运行时就会抛异常。Java中为了提高向下转型的安全性，引入了 `instanceof` ，如果该表达式为true，则可以安全转换。

```
public class TestAnimal {
    public static void main(String[] args) {
        Cat cat = new Cat("元宝",2);
        Dog dog = new Dog("小七", 1);

        // 向上转型
        Animal animal = cat;
        animal.eat();
        animal = dog;
        animal.eat();

        if(animal instanceof Cat){
            cat = (Cat)animal;
            cat.mew();
        }

        if(animal instanceof Dog){
            dog = (Dog)animal;
            dog.bark();
        }
    }
}
```


`instanceof` 关键词官方介绍：https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.20.2

1.4 多态的优缺点

假设有如下代码

```java
class Shape {
    //属性....
    public void draw() {
        System.out.println("画图形! ");
    }
}
class Rect extends Shape{
    @Override
    public void draw() {
        System.out.println("♦");
    }
}
class Cycle extends Shape{
    @Override
    public void draw() {
        System.out.println("●");
    }
}
class Flower extends Shape{
    @Override
    public void draw() {
        System.out.println("❀");
    }
}
```

【使用多态的好处】

1. **能够降低代码的 "圈复杂度", 避免使用大量的 if - else**

> 1. 什么叫 "圈复杂度" ?
> 2. 圈复杂度是一种描述一段代码复杂程度的方式. 一段代码如果平铺直叙, 那么就比较简单容易理解. 而如果有很多的条件分支或者循环语句, 就认为理解起来更复杂.
> 3. 因此我们可以简单粗暴的计算一段代码中条件语句和循环语句出现的个数, 这个个数就称为 "圈复杂度". 如果一个方法的圈复杂度太高, 就需要考虑重构.
> 4. 不同公司对于代码的圈复杂度的规范不一样. 一般不会超过 10 .

例如我们现在需要打印的不是一个形状了, 而是多个形状. 如果不基于多态, 实现代码如下:

```java
public static void drawShapes() {
    Rect rect = new Rect();
    Cycle cycle = new Cycle();
    Flower flower = new Flower();
    String[] shapes = {"cycle", "rect", "cycle", "rect", "flower"};

    for (String shape : shapes) {
```

```
        if (shape.equals("cycle")) {
            cycle.draw();
        } else if (shape.equals("rect")) {
            rect.draw();
        } else if (shape.equals("flower")) {
            flower.draw();
        }
    }
}
```

如果使用使用多态, 则不必写这么多的 if - else 分支语句, 代码更简单.

```
public static void drawShapes() {
    // 我们创建了一个 Shape 对象的数组.
    Shape[] shapes = {new Cycle(), new Rect(), new Cycle(),
                      new Rect(), new Flower()};
    for (Shape shape : shapes) {
        shape.draw();
    }
}
```

2. 可扩展能力更强

如果要新增一种新的形状, 使用多态的方式代码改动成本也比较低.

```
class Triangle extends Shape {
    @Override
    public void draw() {
        System.out.println("△");
    }
}
```

对于类的调用者来说(drawShapes方法), 只要创建一个新类的实例就可以了, 改动成本很低.

而对于不用多态的情况, 就要把 drawShapes 中的 if - else 进行一定的修改, 改动成本更高.

1.5 避免在构造方法中调用重写的方法

一段有坑的代码. 我们创建两个类, B 是父类, D 是子类. D 中重写 func 方法. 并且在 B 的构造方法中调用 func

```
class B {
```

```java
    public B() {
        // do nothing
        func();
    }

    public void func() {
        System.out.println("B.func()");
    }
}

class D extends B {
    private int num = 1;
    @Override
    public void func() {
        System.out.println("D.func() " + num);
    }
}

public class Test {
    public static void main(String[] args) {
        D d = new D();
    }
}

// 执行结果
D.func() 0
```

- 构造 D 对象的同时, 会调用 B 的构造方法.
- B 的构造方法中调用了 func 方法, 此时会触发动态绑定, 会调用到 D 中的 func
- 此时 D 对象自身还没有构造, 此时 num 处在未初始化的状态, 值为 0.
- 所以在构造函数内，尽量避免使用实例方法，除了final和private方法。

结论: "用尽量简单的方式使对象进入可工作状态", 尽量不要在构造器中调用方法(如果这个方法被子类重写, 就会触发动态绑定, 但是此时子类对象还没构造完成), 可能会出现一些隐藏的但是又极难发现的问题.

2. 小试牛刀

1. 以下关于Java多态的描述，哪一个是错误的?

   A. 多态允许不同类的对象对同一消息做出响应

   B. 方法重载是编译时多态的一种形式

   C. 子类可以重写父类的final方法来实现运行时多态

D. 接口是实现多态的一种方式

2. 下列哪种情况下会发生向上转型（upcasting）？

A. 将子类对象赋值给父类引用

B. 将父类对象赋值给子类引用

C. 在子类中调用super()

D. 使用instanceof运算符

3. 关于方法重写（override），以下哪个陈述是正确的?

A. 重写方法的访问修饰符必须与被重写方法完全相同

B. 重写方法可以抛出比被重写方法更广泛的检查异常

C. 静态方法可以被重写

D. 重写方法的返回类型可以是被重写方法返回类型的子类

4. 在Java中，以下哪种情况下不会发生多态行为?

A. 使用接口引用调用实现类的方法

B. 使用父类引用调用被子类重写的方法

C. 调用重载的方法

D. 使用 final 关键字修饰的方法

答案解析：

```
1. C    2. A    3. D    4. D
```

完