

8. 继承和多态

【本节目标】

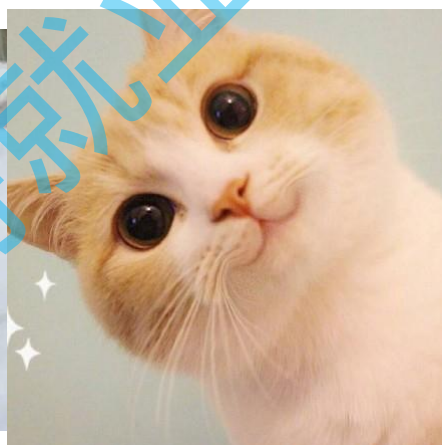
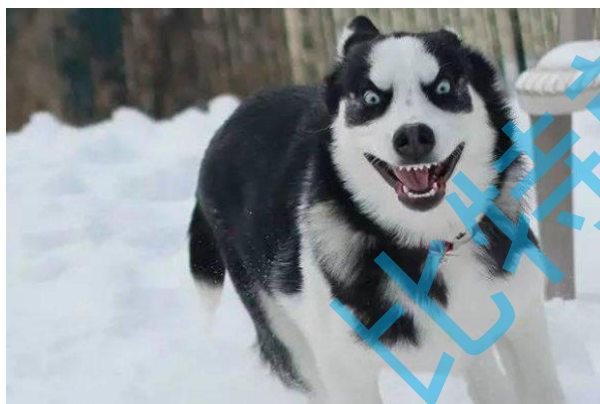
1. 继承
2. 组合
3. 多态

1 继承

1.1 为什么需要继承

Java中使用类对现实世界中实体来进行描述，类经过实例化之后的产物对象，则可以用来表示现实中的实体，但是现实世界错综复杂，事物之间可能会存在一些关联，那在设计程序是就需要考虑。

比如：狗和猫，它们都是一个动物。



使用Java语言来进行描述，就会设计出：

```
// Dog.java
public class Dog{
    string name;
    int age;
    float weight;

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

    public void sleep(){
        System.out.println(name + "正在睡觉");
    }
}
```

```

void Bark(){
    System.out.println(name + "汪汪汪~~~");
}
}

// Cat.java
public class Cat{
    string name;
    int age;
    float weight;

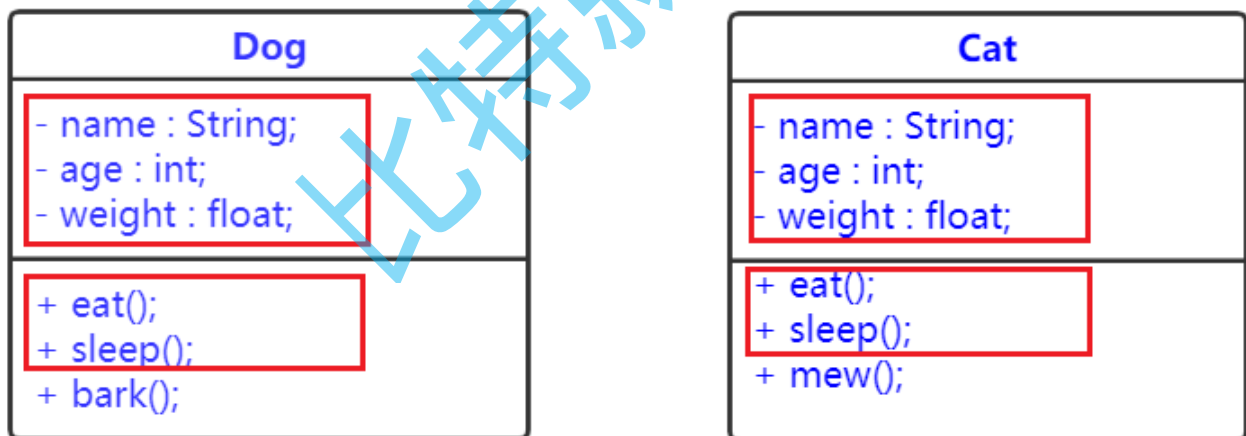
    public void eat(){
        System.out.println(name + "正在吃饭");
    }

    public void sleep()
    {
        System.out.println(name + "正在睡觉");
    }

    void mew(){
        System.out.println(name + "喵喵喵~~~");
    }
}

```

通过观察上述代码会发现，猫和狗的类中存在大量重复，如下所示：

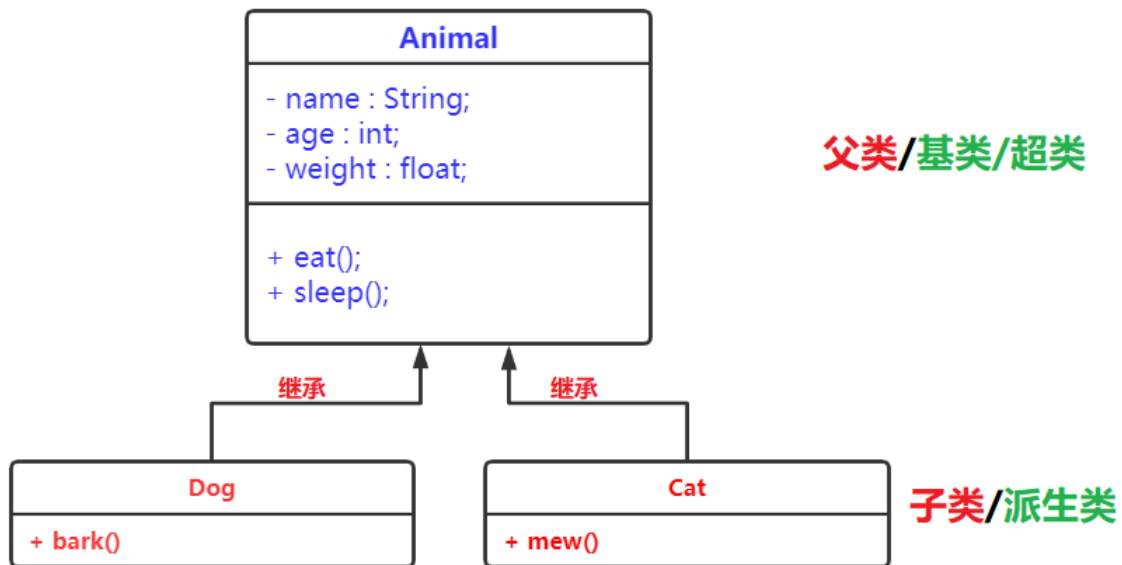


那能否将这些共性抽取呢？面向对象思想中提出了继承的概念，专门用来进行共性抽取，实现代码复用。

1.2 继承概念

继承(inheritance)机制：是面向对象程序设计使代码可以复用的最重要手段，它允许程序员在保持原有类特性的基础上进行扩展，增加新功能，这样产生新的类，称**派生类**。继承呈现了面向对象程序设计的层次结构，体现了由简单到复杂的认知过程。继承主要解决的问题是：**共性的抽取，实现代码复用**。

例如：狗和猫都是动物，那么我们就可以将共性的内容进行抽取，然后采用继承的思想来达到共用。



上述图示中，Dog和Cat都继承了Animal类，其中：Animal类称为父类/基类或超类，Dog和Cat可以称为Animal的子类/派生类，继承之后，子类可以复用父类中成员，子类在实现时只需关心自己新增加的成员即可。

从继承概念中可以看出继承最大的作用就是：实现代码复用，还有就是来实现多态(后序讲)。

1.3 继承的语法

在Java中如果要表示类之间的继承关系，需要借助**extends**关键字，具体如下：

```
修饰符 class 子类 extends 父类 {
    // ...
}
```

对1.2中场景使用继承方式重新设计：

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

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

    public void sleep(){
        System.out.println(name + "正在睡觉");
    }
}

// Dog.java
public class Dog extends Animal{
    void bark(){
        System.out.println(name + "汪汪汪~~~");
    }
}
```

```
// Cat.java
public class Cat extends Animal{
    void mew(){
        System.out.println(name + "喵喵喵~~~");
    }
}

// TestExtend.java
public class TestExtend {
    public static void main(String[] args) {
        Dog dog = new Dog();
        // dog类中并没有定义任何成员变量，name和age属性肯定是从父类Animal中继承下来的
        System.out.println(dog.name);
        System.out.println(dog.age);

        // dog访问的eat()和sleep()方法也是从Animal中继承下来的
        dog.eat();
        dog.sleep();
        dog.bark();
    }
}
```

注意：

1. 子类会将父类中的成员变量或者成员方法继承到子类中了
2. 子类继承父类之后，必须要新添加自己特有的成员，体现出与基类的不同，否则就没有必要继承了

1.4 父类成员访问

在继承体系中，子类将父类中的方法和字段继承下来了，那在子类中能否直接访问父类中继承下来的成员呢？

1.4.1 子类中访问父类的成员变量

1. 子类和父类不存在同名成员变量

```
public class Base {
    int a;
    int b;
}

public class Derived extends Base{
    int c;
    public void method(){
        a = 10; // 访问从父类中继承下来的a
        b = 20; // 访问从父类中继承下来的b
        c = 30; // 访问子类自己的c
    }
}
```

2. 子类和父类成员变量同名

```

public class Base {
    int a;
    int b;
    int c;
}

////////////////////////////////////
public class Derived extends Base{
    int a;        // 与父类中成员a同名，且类型相同
    char b;       // 与父类中成员b同名，但类型不同

    public void method(){
        a = 100;    // 访问父类继承的a，还是子类自己新增的a?
        b = 101;    // 访问父类继承的b，还是子类自己新增的b?
        c = 102;    // 子类没有c，访问的肯定是从父类继承下来的c
        // d = 103; // 编译失败，因为父类和子类都没有定义成员变量b
    }
}

```

在子类方法中 或者 通过子类对象访问成员时：

- 如果访问的成员变量子类中有，优先访问自己的成员变量。
- 如果访问的成员变量子类中无，则访问父类继承下来的，如果父类也没有定义，则编译报错。
- 如果访问的成员变量与父类中成员变量同名，则优先访问自己的。

成员变量访问遵循就近原则，自己有优先自己的，如果没有则向父类中找。

1.4.2 子类中访问父类的成员方法

1. 成员方法名字不同

```

public class Base {
    public void methodA(){
        System.out.println("Base中的methodA()");
    }
}

public class Derived extends Base{
    public void methodB(){
        System.out.println("Derived中的methodB()方法");
    }

    public void methodC(){
        methodB();    // 访问子类自己的methodB()
        methodA();    // 访问父类继承的methodA()
        // methodD(); // 编译失败，在整个继承体系中没有发现方法methodD()
    }
}

```

总结：成员方法没有同名时，在子类方法中或者通过子类对象访问方法时，则优先访问自己的，自己没有时再到父类中找，如果父类中也没有则报错。

2. 成员方法名字相同

```

public class Base {
    public void methodA(){
        System.out.println("Base中的methodA()");
    }

    public void methodB(){
        System.out.println("Base中的methodB()");
    }
}

public class Derived extends Base{
    public void methodA(int a) {
        System.out.println("Derived中的method(int)方法");
    }

    public void methodB(){
        System.out.println("Derived中的methodB()方法");
    }

    public void methodC(){
        methodA();    // 没有传参，访问父类中的methodA()
        methodA(20);  // 传递int参数，访问子类中的methodA(int)
        methodB();    // 直接访问，则永远访问到的都是子类中的methodB()，基类的无法访问到
    }
}

```

【说明】

- 通过子类对象访问父类与子类中不同名方法时，优先在子类中找，找到则访问，否则在父类中找，找到则访问，否则编译报错。
- 通过派生类对象访问父类与子类同名方法时，如果父类和子类同名方法的参数列表不同(重载)，根据调用方法适传递的参数选择合适的方法访问，如果没有则报错；

问题：如果子类中存在与父类中相同的成员时，那如何在子类中访问父类相同名称的成员呢？

1.5 super关键字

由于设计不好，或者因场景需要，子类和父类中可能会存在相同名称的成员，如果要在子类方法中访问父类同名成员时，该如何操作？直接访问是无法做到的，Java提供了**super关键字**，该关键字主要作用：**在子类方法中访问父类的成员。**

```

public class Base {
    int a;
    int b;
    public void methodA(){
        System.out.println("Base中的methodA()");
    }

    public void methodB(){
        System.out.println("Base中的methodB()");
    }
}

```

```

public class Derived extends Base{
    int a; // 与父类中成员变量同名且类型相同
    char b; // 与父类中成员变量同名但类型不同
    // 与父类中methodA()构成重载
    public void methodA(int a) {
        System.out.println("Derived中的method()方法");
    }

    // 与基类中methodB()构成重写(即原型一致, 重写后序详细介绍)
    public void methodB(){
        System.out.println("Derived中的methodB()方法");
    }

    public void methodC(){
        // 对于同名的成员变量, 直接访问时, 访问的都是子类的
        a = 100; // 等价于: this.a = 100;
        b = 101; // 等价于: this.b = 101;
        // 注意: this是当前对象的引用

        // 访问父类的成员变量时, 需要借助super关键字
        // super是获取到子类对象中从基类继承下来的部分
        super.a = 200;
        super.b = 201;

        // 父类和子类中构成重载的方法, 直接可以通过参数列表区分访问父类还是子类方法
        methodA(); // 没有传参, 访问父类中的methodA()
        methodA(20); // 传递int参数, 访问子类中的methodA(int)

        // 如果在子类中要访问重写的基类方法, 则需要借助super关键字
        methodB(); // 直接访问, 则永远访问到的都是子类中的methodA(), 基类的无法访问到
        super.methodB(); // 访问基类的methodB()
    }
}

```

在子类方法中, 如果想要明确访问父类中成员时, 借助super关键字即可。

【注意事项】

1. 只能在非静态方法中使用
2. 在子类方法中, 访问父类的成员变量和方法。

super的其他用法在后文中介绍。

1.6 子类构造方法

父子父子, 先有父再有子, 即: 子类对象构造时, 需要先调用基类构造方法, 然后执行子类的构造方法。

```

public class Base {
    public Base(){
        System.out.println("Base()");
    }
}

```

```

}

public class Derived extends Base{
    public Derived(){
        // super(); // 注意子类构造方法中默认会调用基类的无参构造方法：super(),
        // 用户没有写时,编译器会自动添加, 而且super()必须是子类构造方法中第一条语句,
        // 并且只能出现一次
        System.out.println("Derived()");
    }
}

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

```

结果打印:

```

Base()
Derived()

```

在子类构造方法中，并没有写任何关于基类构造的代码，但是在构造子类对象时，先执行基类的构造方法，然后执行子类的构造方法，因为：**子类对象中成员是有两部分组成的，基类继承下来的以及子类新增加的部分。父子父子肯定是先有父再有子，所以在构造子类对象时候，先要调用基类的构造方法，将从基类继承下来的成员构造完整，然后再调用子类自己的构造方法，将子类自己新增加的成员初始化完整。**

注意：

1. 若父类显式定义无参或者默认的构造方法，在子类构造方法第一行默认有隐含的super()调用，即调用基类构造方法
2. 如果父类构造方法是带有参数的，此时需要用户为子类显式定义构造方法，并在子类构造方法中选择合适的父类构造方法调用，否则编译失败。
3. 在子类构造方法中，super(...)调用父类构造时，必须是子类构造函数中第一条语句。
4. super(...)只能在子类构造方法中出现一次，并且不能和this同时出现

1.7 super和this

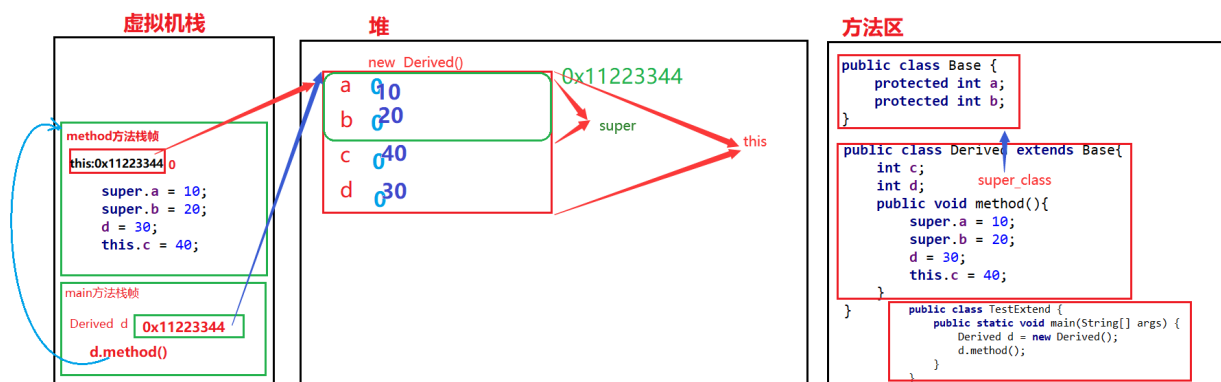
super和this都可以在成员方法中用来访问：成员变量和调用其他的成员函数，都可以作为构造方法的第一条语句，那他们之间有什么区别呢？

【相同点】

1. 都是Java中的关键字
2. 只能在类的非静态方法中使用，用来访问非静态成员方法和字段
3. 在构造方法中调用时，必须是构造方法中的第一条语句，并且不能同时存在

【不同点】

1. this是当前对象的引用，当前对象即调用实例方法的对象，super相当于是子类对象中从父类继承下来部分成员的引用



2. 在非静态成员方法中，`this`用来访问本类的方法和属性，`super`用来访问父类继承下来的方法和属性
3. 在构造方法中：`this(...)`用于调用本类构造方法，`super(...)`用于调用父类构造方法，两种调用不能同时在构造方法中出现
4. 构造方法中一定会存在`super(...)`的调用，用户没有写编译器也会增加，但是`this(...)`用户不写则没有

1.8 再谈初始化

我们还记得之前讲过的代码块吗？我们简单回顾一下几个重要的代码块：实例代码块和静态代码块。在没有继承关系时的执行顺序。

```
class Person {
    public String name;
    public int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("构造方法执行");
    }
    {
        System.out.println("实例代码块执行");
    }
    static {
        System.out.println("静态代码块执行");
    }
}

public class TestDemo {
    public static void main(String[] args) {
        Person person1 = new Person("bit", 10);
        System.out.println("=====");
        Person person2 = new Person("gaobo", 20);
    }
}
```

执行结果：

静态代码块执行

实例代码块执行

构造方法执行

=====

实例代码块执行

构造方法执行

1. 静态代码块先执行，并且只执行一次，在类加载阶段执行
2. 当有对象创建时，才会执行实例代码块，实例代码块执行完成后，最后构造方法执行

【继承关系上的执行顺序】

```
class Person {  
    public String name;  
    public int age;  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
        System.out.println("Person: 构造方法执行");  
    }  
    {  
        System.out.println("Person: 实例代码块执行");  
    }  
    static {  
        System.out.println("Person: 静态代码块执行");  
    }  
}  
  
class Student extends Person{  
  
    public Student(String name,int age) {  
        super(name,age);  
        System.out.println("Student: 构造方法执行");  
    }  
  
    {  
        System.out.println("Student: 实例代码块执行");  
    }  
  
    static {  
        System.out.println("Student: 静态代码块执行");  
    }  
  
}  
  
public class TestDemo4 {  
  
    public static void main(String[] args) {  
        Student student1 = new Student("张三",19);  
        System.out.println("=====");  
        Student student2 = new Student("gaobo",20);  
    }  
}
```

```

}

public static void main1(String[] args) {
    Person person1 = new Person("bit",10);
    System.out.println("=====");
    Person person2 = new Person("gaobo",20);
}
}

```

执行结果：

```

Person: 静态代码块执行
Student: 静态代码块执行
Person: 实例代码块执行
Person: 构造方法执行
Student: 实例代码块执行
Student: 构造方法执行
=====
Person: 实例代码块执行
Person: 构造方法执行
Student: 实例代码块执行
Student: 构造方法执行

```

通过分析执行结果，得出以下结论：

- 1、父类静态代码块优先于子类静态代码块执行，且是最早执行
- 2、父类实例代码块和父类构造方法紧接着执行
- 3、子类的实例代码块和子类构造方法紧接着再执行
- 4、第二次实例化子类对象时，父类和子类的静态代码块都将不会再执行

1.9 protected 关键字

在类和对象章节中，为了实现封装特性，Java中引入了访问限定符，主要限定：类或者类中成员能否在类外或者其他包中被访问。

No	范围	private	default	protected	public
1	同一包中的同一类	✓	✓	✓	✓
2	同一包中的不同类		✓	✓	✓
3	不同包中的子类			✓	✓
4	不同包中的非子类				✓

那父类中不同访问权限的成员，在子类中的可见性又是什么样子的呢？

```
// 为了掩饰基类中不同访问权限在子类中的可见性，为了简单类B中就不设置成员方法了
// extend01包中
public class B {
    private int a;
    protected int b;
    public int c;
    int d;
}

// extend01包中
// 同一个包中的子类
public class D extends B {
    public void method(){
        // super.a = 10;    // 编译报错，父类private成员在相同包子类中不可见
        super.b = 20;      // 父类中protected成员在相同包子类中可以直接访问
        super.c = 30;      // 父类中public成员在相同包子类中可以直接访问
        super.d = 40;      // 父类中默认访问权限修饰的成员在相同包子类中可以直接访问
    }
}

// extend02包中
// 不同包中的子类
public class C extends B {
    public void method(){
        // super.a = 10;    // 编译报错，父类中private成员在不同包子类中不可见
        super.b = 20;      // 父类中protected修饰的成员在不同包子类中可以直接访问
    }
}
```

```

    super.c = 30;    // 父类中public修饰的成员在不同包子类中可以直接访问
    //super.d = 40;  // 父类中默认访问权限修饰的成员在不同包子类中不能直接访问
}
}

// extend02包中
// 不同包中的类
public class TestC {
    public static void main(String[] args) {
        C c = new C();
        c.method();
        // System.out.println(c.a); // 编译报错, 父类中private成员在不同包其他类中不可见
        // System.out.println(c.b); // 父类中protected成员在不同包其他类中不能直接访问
        System.out.println(c.c);    // 父类中public成员在不同包其他类中可以直接访问
        // System.out.println(c.d); // 父类中默认访问权限修饰的成员在不同包其他类中不能直接访问
    }
}
}

```

注意：父类中private成员变量虽然在子类中不能直接访问，但是也继承到子类中了

什么时候下用哪一种呢？

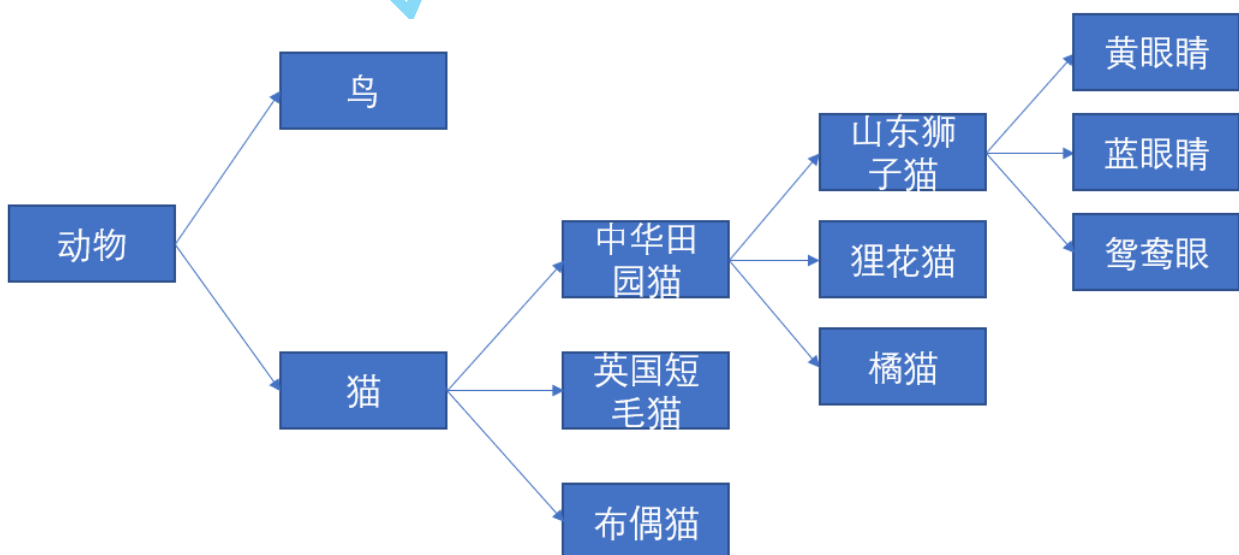
我们希望类要尽量做到 "封装", 即隐藏内部实现细节, 只暴露出 **必要** 的信息给类的调用者。

因此我们在使用的時候应该尽可能的使用 **比较严格** 的访问权限。例如如果一个方法能用 private, 就尽量不要用 public。

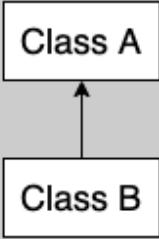
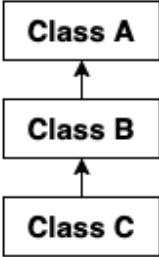
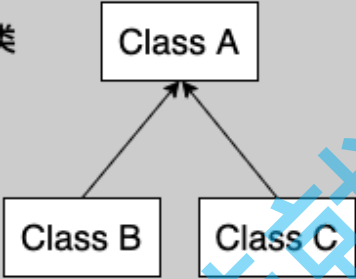
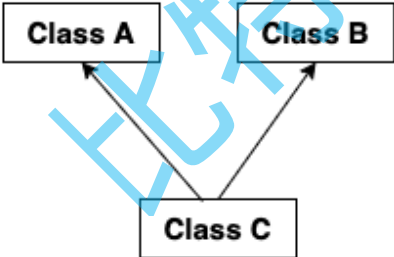
另外, 还有一种 **简单粗暴** 的做法: 将所有的字段设为 private, 将所有的方法设为 public。不过这种方式属于是对访问权限的滥用, 还是更希望同学们能写代码的时候认真思考, 该类提供的字段方法到底给 "谁" 使用(是类内部自己用, 还是类的调用者使用, 还是子类使用)。

1.10 继承方式

在现实生活中, 事物之间的关系是非常复杂, 灵活多样, 比如:



但在Java中只支持以下几种继承方式：

<div>单继承</div> <div><pre>graph BT; B[Class B] --> A[Class A]</pre></div>	<pre>public class A { } public class B extends A { }</pre>
<div>多层继承</div> <div><pre>graph BT; C[Class C] --> B[Class B]; B --> A[Class A]</pre></div>	<pre>public class A {.....} public class B extends A {.....} public class C extends B {.....}</pre>
<div>不同类继承同一个类</div> <div><pre>graph BT; B[Class B] --> A[Class A]; C[Class C] --> A</pre></div>	<pre>public class A {.....} public class B extends A {.....} public class C extends A {.....}</pre>
<div>多继承 (不支持)</div> <div><pre>graph BT; C[Class C] --> A[Class A]; C --> B[Class B]</pre></div>	<pre>public class A {.....} public class B {.....} public class C extends A, B { } // Java 不支持多继承</pre>

注意：Java中不支持多继承。

时刻牢记，我们写的类是现实事物的抽象。而我们真正在公司中所遇到的项目往往业务比较复杂，可能会涉及到一系列复杂的概念，都需要我们使用代码来表示，所以我们真实项目中所写的类也会有很多。类之间的关系也会更加复杂。

但是即使如此，我们并不希望类之间的继承层次太复杂。一般我们不会出现超过三层的继承关系。如果继承层次太多，就需要考虑对代码进行重构了。

如果想从语法上进行限制继承，就可以使用 final 关键字

1.11 final 关键字

final 关键字可以用来修饰变量、成员方法以及类。

1. 修饰变量或字段，表示常量(即不能修改)

```
final int a = 10;  
a = 20; // 编译出错
```

2. 修饰类：表示此类不能被继承

```
final public class Animal {  
    ...  
}  
  
public class Bird extends Animal {  
    ...  
}  
  
// 编译出错  
Error:(3, 27) java: 无法从最终com.bit.Animal进行继
```

```
public final class String  
    implements java.io.Serializable, Comparable<String>, CharSequ  
    /** The value is used for character storage. */  
    private final char value[];  
  
    /** Cache the hash code for the string */  
    private int hash; // Default to 0
```

我们平时是用的 String 字符串类, 就是用 final 修饰的, 不能被继承.

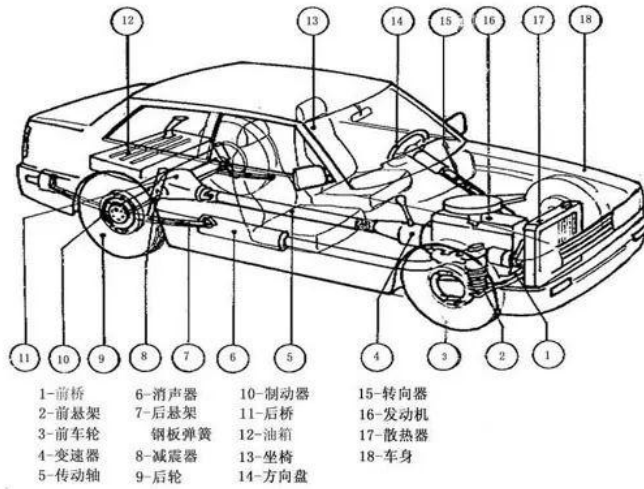
3. 修饰方法：表示该方法不能被重写(后序介绍)

1.12 继承与组合

和继承类似, 组合也是一种表达类之间关系的方式, 也是能够达到代码重用的效果。组合并没有涉及到特殊的语法(诸如 extends 这样的关键字), 仅仅是将一个类的实例作为另外一个类的字段。

继承表示对象之间是is-a的关系, 比如: 狗是动物, 猫是动物

组合表示对象之间是has-a的关系, 比如: 汽车



汽车和其轮胎、发动机、方向盘、车载系统等关系就应该是组合，因为汽车是有这些部件组成的。

```
// 轮胎类
class Tire{
    // ...
}

// 发动机类
class Engine{
    // ...
}

// 车载系统类
class VehicleSystem{
    // ...
}

class Car{
    private Tire tire;    // 可以复用轮胎中的属性和方法
    private Engine engine; // 可以复用发动机中的属性和方法
    private VehicleSystem vs; // 可以复用车载系统中的属性和方法

    // ...
}

// 奔驰是汽车
class Benz extend Car{
    // 将汽车中包含的：轮胎、发动机、车载系统全部继承下来
}
}
```


组合和继承都可以实现代码复用，应该使用继承还是组合，需要根据应用场景来选择，一般建议：能用组合尽量用组合。

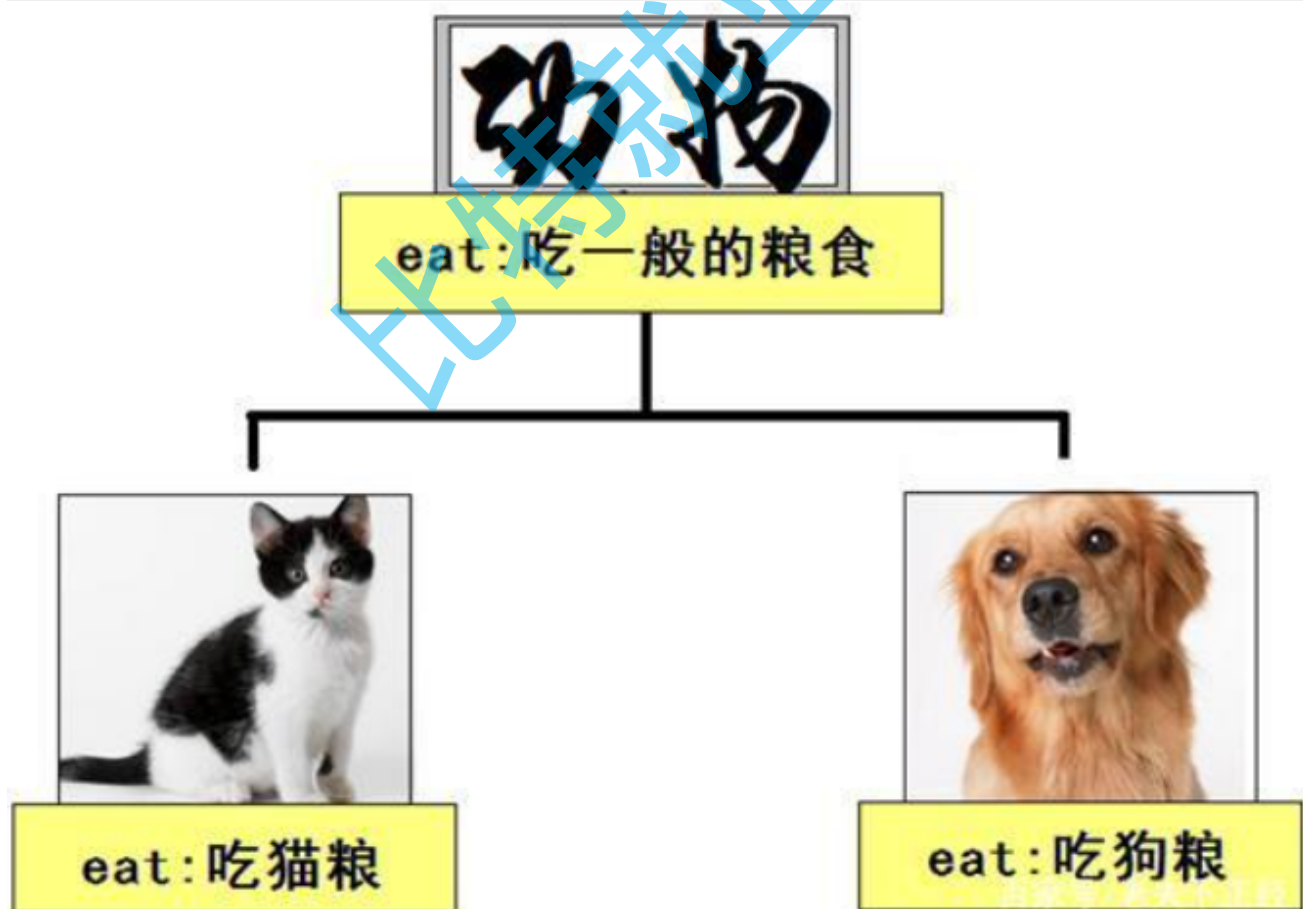
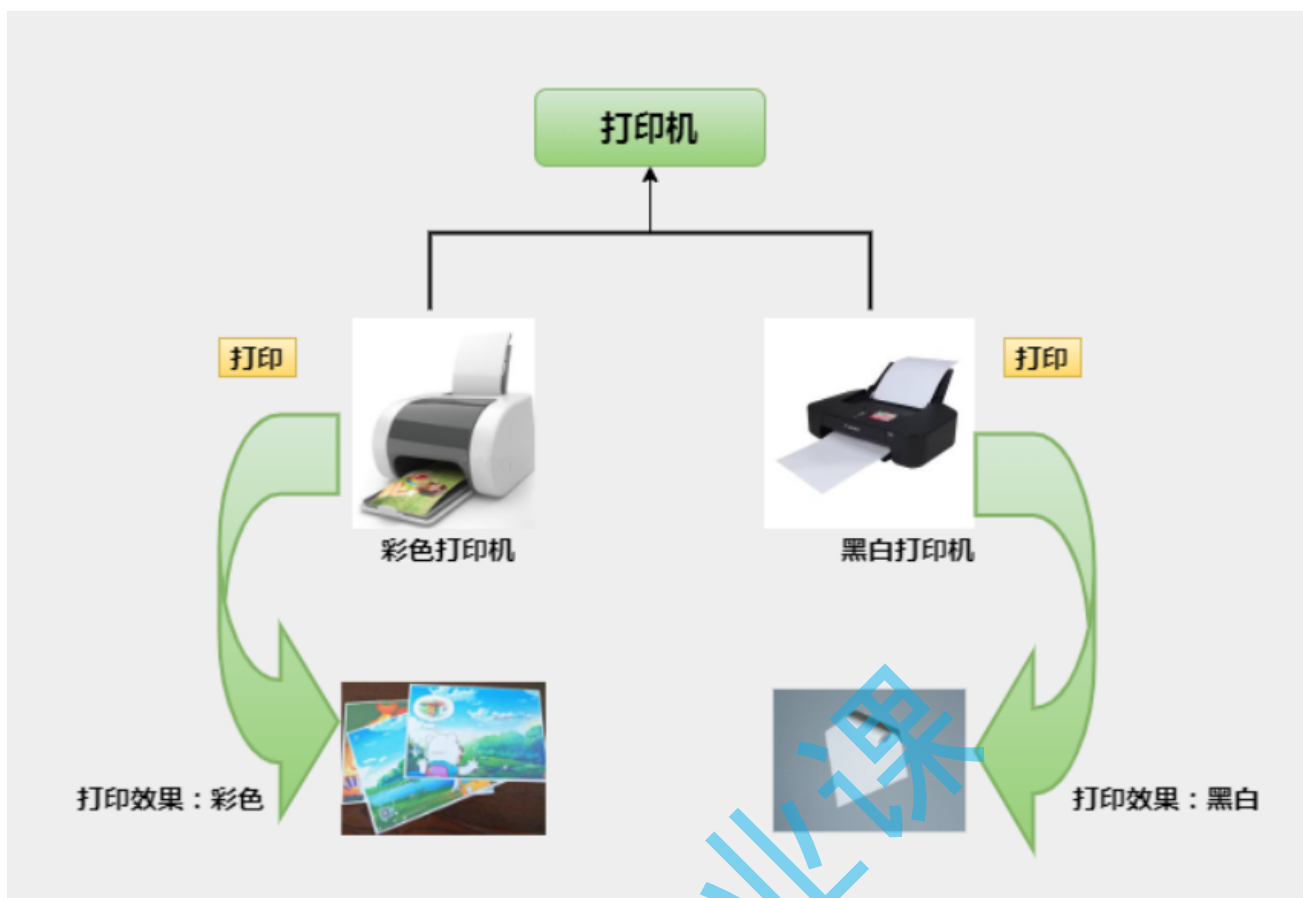
[继承和组合](#)

2 多态

2.1 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。

比特就业课



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。

2.2 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。

```
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();
}

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

    eat(cat);
    eat(dog);
}
}

```

运行结果:

元宝吃鱼~~~

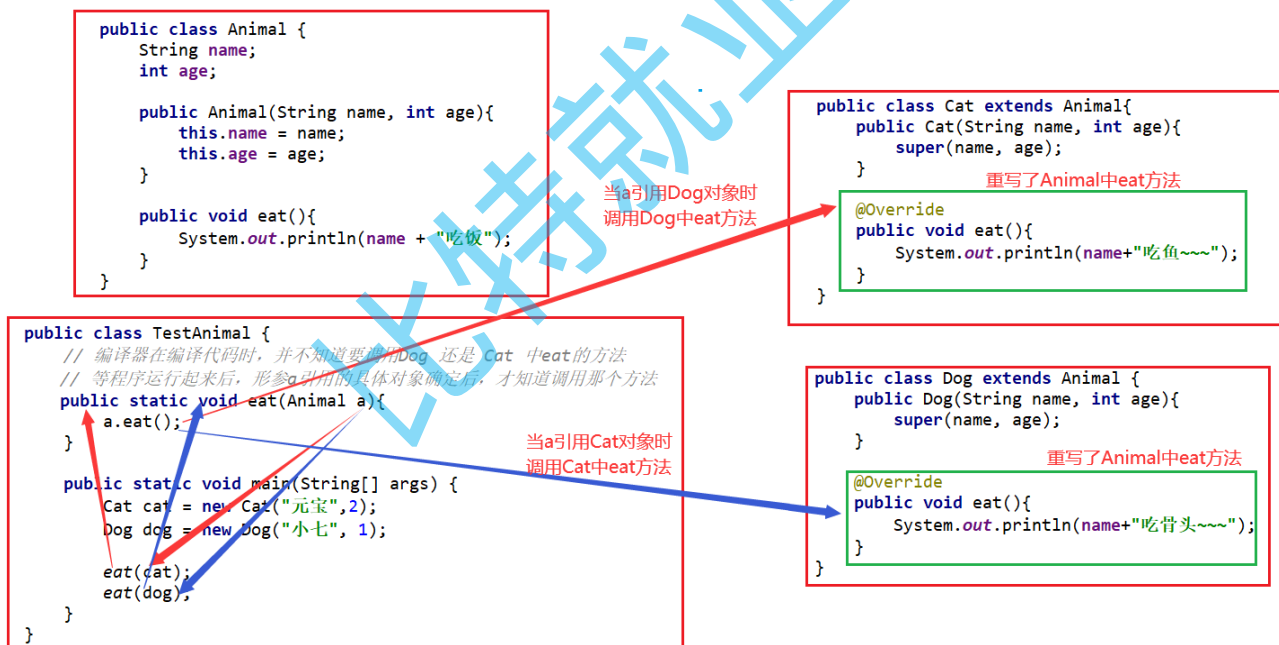
元宝正在睡觉

小七吃骨头~~~

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**。



2.3 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。

Overriding 重写

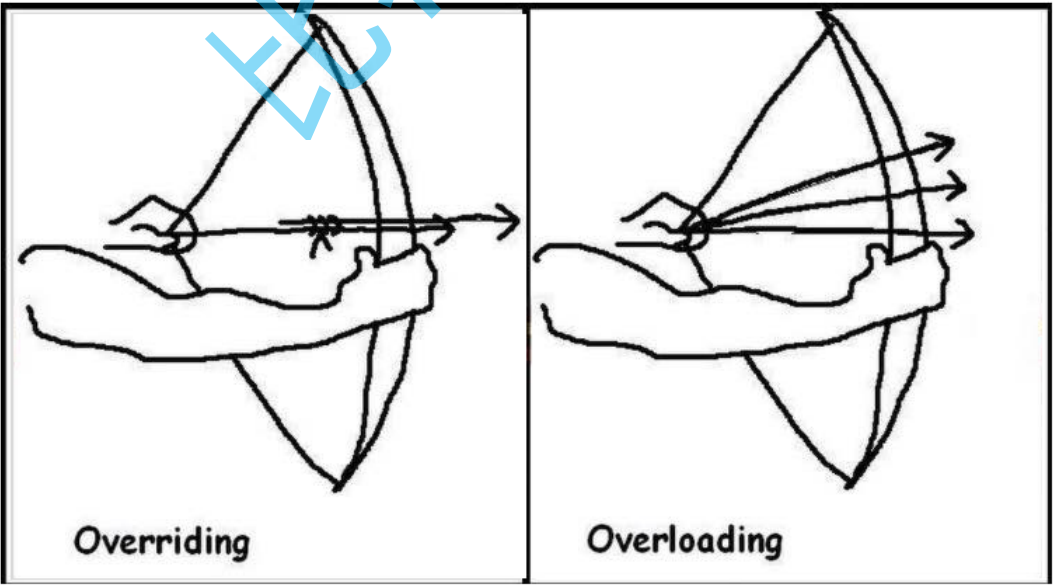
```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }
    public void bark(){
        System.out.println("bowl");
    }
}
```

方法名与参数都一样

Overloading 重载

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

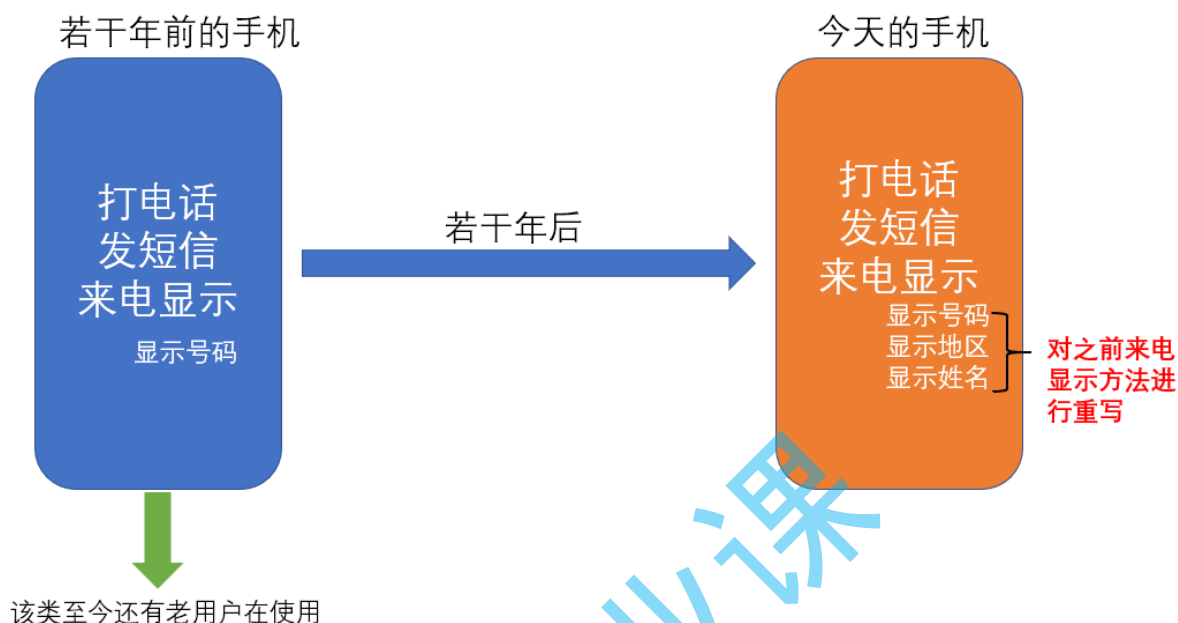
方法名相同，参数不同



【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。

2.4 向上转移和向下转型

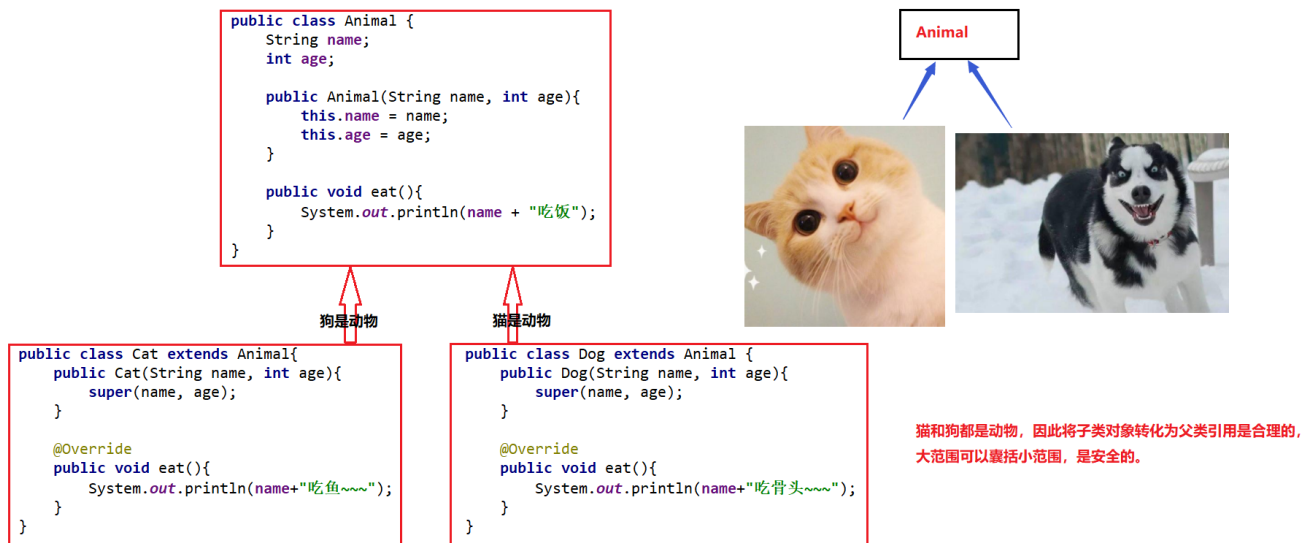
2.4.1 向上转型

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" == var){
            return new Dog("狗狗",1);
        }else if("猫" == 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。

2.4.2 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。



```
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
        // 程序可以通过编译，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>

2.5 多态的优缺点

假设有如下代码：

```
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

什么叫 "圈复杂度" ?

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. 而如果有许多的条件分支或者循环语句, 就认为理解起来更复杂.

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.

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:

```
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.

多态缺陷: 代码的运行效率降低。

1. 属性没有多态性

当父类和子类都有同名属性的时候, 通过父类引用, 只能引用父类自己的成员属性

2. 构造方法没有多态性

见如下代码~

2.6 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 {
    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. 如果具备多态性, num 的值应该是 1.
- 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.

比特就业课