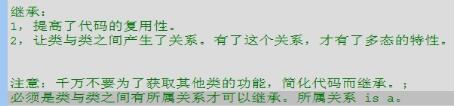
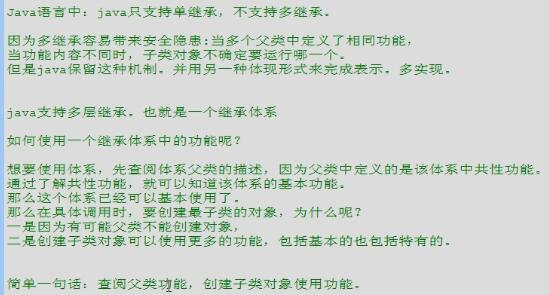
**P84 面向对象（继承-概述）**



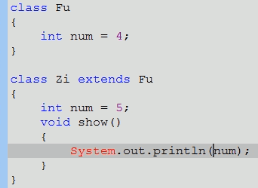
**P85 面向对象（继承-概述2）**



**P86 面向对象（聚集关系）**

**//**

**P87 面向对象（子父类中变量的特点）-变量**

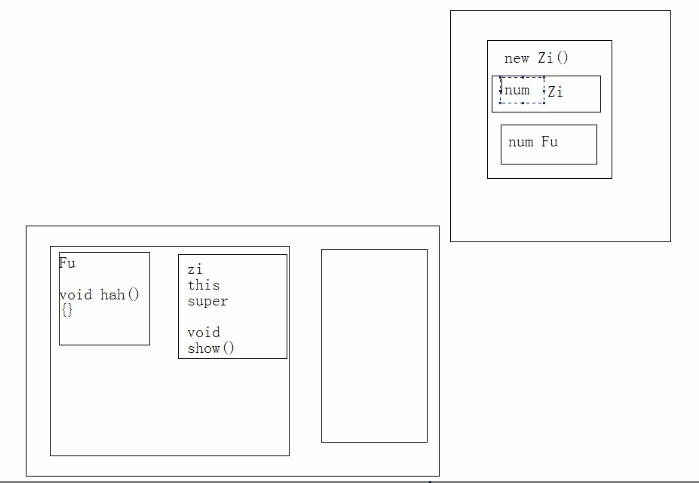


运行show()方法后输出的结果是5。

这很直观，但是为什么呢？因为num是非静态变量，前面省去了this.

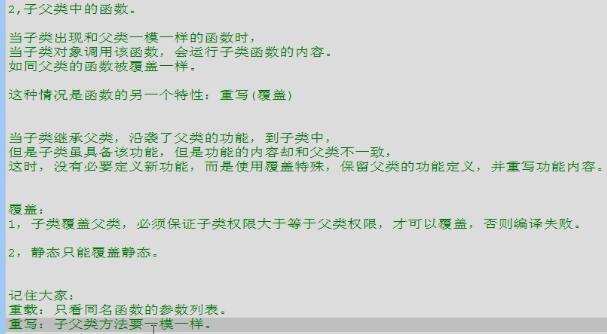
注意，把子类的int num = 5；注释掉之后，print（this.num）结果是4。因为此时Zi中没有再定义num，所以num Fu赋给了num Zi。

当new一个子类时，父类（包括静态变量，静态方法，非静态方法）先进入方法区，随后子类也跟着进入方法区。同时，父类和子类的特有变量，一起进入到堆内存中。

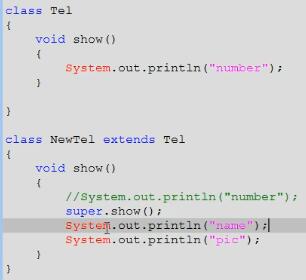


**P88 面向对象（子父类中变量的特点-覆盖）-函数**

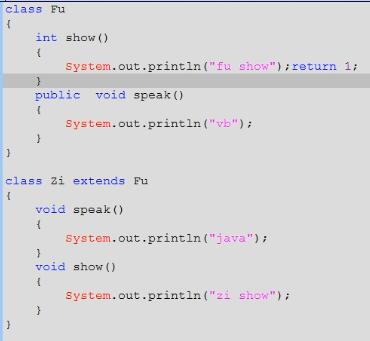
覆盖也叫重写。其实父类的方法也还存在在内存中。



覆盖的好处：用于功能的扩展。比如以前写了个功能，现在随着时代的方法想对该方法进行升级。直接对源码进行修改是大忌，所以只要对功能所在的类进行继承，然后对该方法进行覆盖即可。



该图就是只想对Tel中的show()进行一定的拓展。



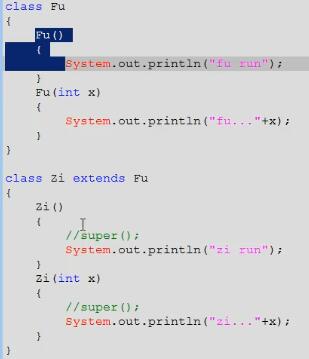
该图有两处错误：

1.show方法，因为子类和父类不是一模一样，所以这不是重写。

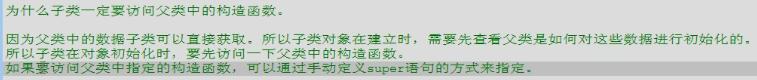
2.speak方法，因为父类的权重大于子类，所以不行。

**P89 面向对象（子父类中变量的特点-子类实例化过程）-构造函数**

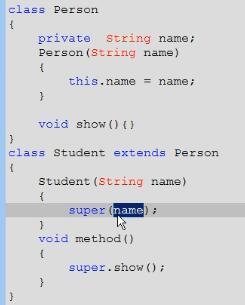
--当一个子类继承一个父类时，不管父类的构造函数有多少种，子类的构造函数有多少种，如果子类的构造函数没有特别说明，那么都是先执行默认的空参数的父类构造函数，再执行与传入参数对应的子类构造函数。



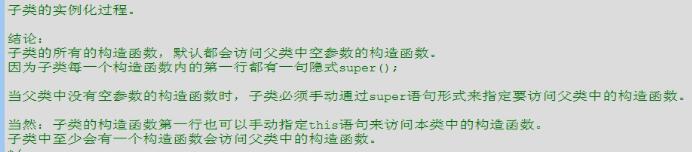
当父类没有空参数的构造函数，或者想用其他类型的父类构造函数时，子类的每个构造函数都必须手动去指定。



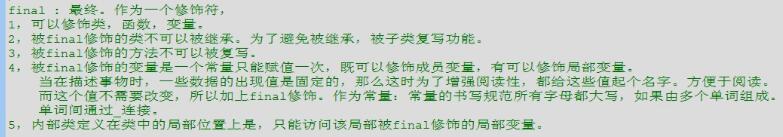
实例如下，这样一方面知道了父类的构造函数对父类中的变量是否做过修改，改了哪些地方，另一方面父类构造函数中做过了的操作，子类可以直接调用。



注意，super语句一定定义在子类构造函数的第一行，同时因为this也必须第一行，所以两者只能选一个。



**P90 面向对象（final关键字）**



**P91 面向对象（抽象类）**

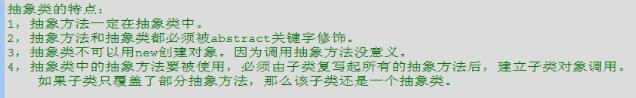
比如一个函数：

Void show()

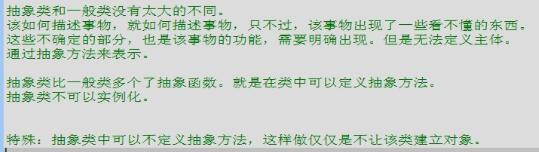
{}

第一行为函数的第一部分：函数声明，第二行为函数的第二部分：函数主体。

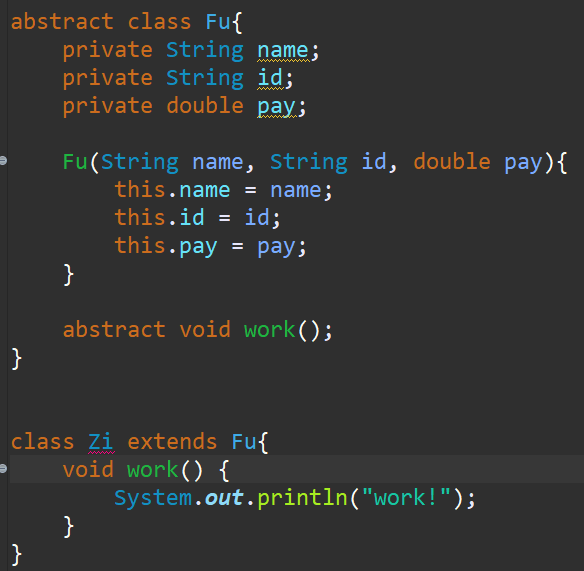
基于此，我们发现一种情况：当多个类中出现相同功能，但是功能主体不同，这是可以进行向上抽取的。这时，只抽取功能定义，而不抽取功能主体。



**P92 面向对象（抽象类2）**

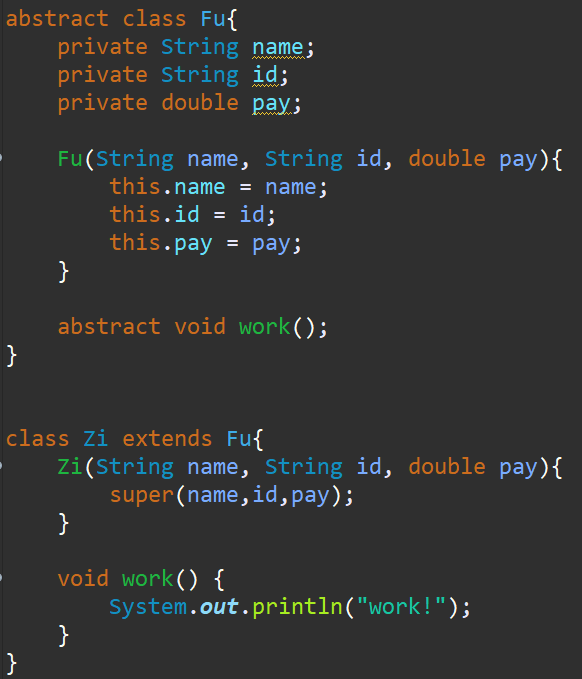


**P93 面向对象（抽象类练习）**



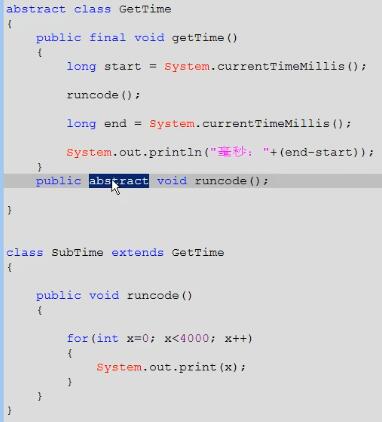
不要以为子类和父类的构造函数形式一样就可以不用写，不写并不是直接继承了父类的初始化方法，而是默认的Zi(){super()};

应该进行补充，但是可以调用父类的初始化



**P94 面向对象（模板方法模式）**

需求：获取程序运行的时间

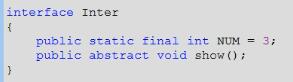


当然，runcode也并不是非要抽象，自定义一段代码作为默认代码也行。



**P95 面向对象（接口）**





当然，可以简写成：

Interface Inter

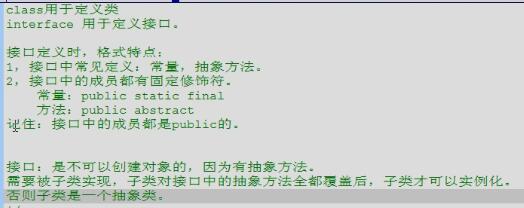
{

int NUM = 3;

void show();

}

但是，为了代码的阅读性，还是推荐第一种。同时注意了，碰到第二种写法，你在定义类重写该方法时，不要忘了权限是public



**P96 面向对象（接口2）**

类与类之间是单继承

类与接口之间是实现

接口与接口之间是继承，而且可以多继承。但是注意，比如接口C继承接口A和接口B，A和B可以有同名方法，但是返回值的类型必须一致。

**P97 面向对象（接口的特点）**

//

**P98 面向对象（接口举例实现）**

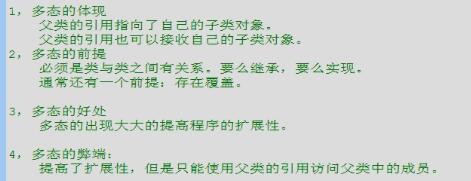
//

**P99 面向对象（多态-概念）**

多态的表现，举个例子



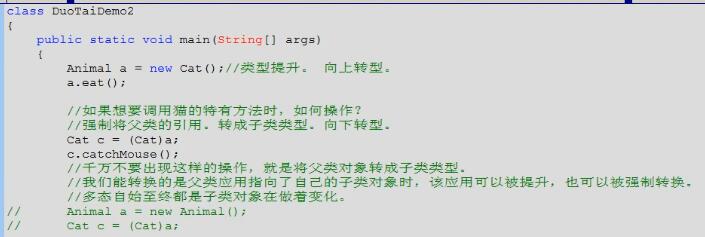
**P100 面向对象（多态-概念）**



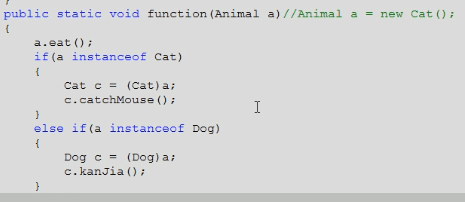
如何理解提高程序的扩展性：比如有个父类A，你继承A得到了B，想对B进行操作（该操作A也得有，即B覆盖了A的该操作），于是定义了一个函数，传入B b。过了一段时间，你又继承A得到了C，想对C进行同样的操作，于是重载了该函数，传入C c。这样一来，每次出现A的子类，你都得重载该函数。针对这个问题，使用多态可以一步到位：你定义该函数，传入A a既可。

**P101 面向对象（多态-转型）**

举个例子，其中Animal是父类，Cat是子类



指向的对象自始至终都是子类对象，只不过引用类型可以是父类，也可以是子类，所以只是子类对象在做着变化。



这就是多态的好处，提升了代码的拓展性，怎么体现的呢：如上，定义了一个函数，它接受的是父类类型，而不是子类类型，那么所有继承该父类的子类在向上提升后都能传入。

接下来，当需要调用子类的特有方法时，我们根据条件判断，将父类类型强制向下转化成与其对象相对应的子类类型（因为多态自始至终创建的对象都是子类对象）。

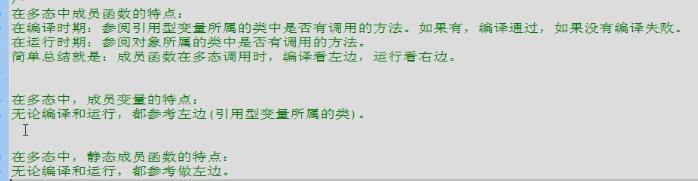
语句 // a instanceof Cat

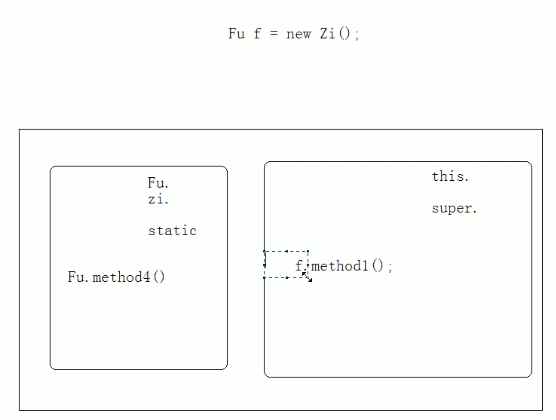
它的作用是判断a指向的对象是否是Cat，返回一个布尔值。注意，如果写成a instanceof Animal，那么它返回的是true

**P102 面向对象（多态-示例）**

多态的好处：以前一次性只能指挥一个对象做某些事情，而多态则对这些对象进行向上抽取（也就是得到它们的父类），然后对一次性指挥一个父类做某些事情，这样就相当于一次性指挥了一堆子类做事情。

**P103 面向对象（多态中成员的特点）**

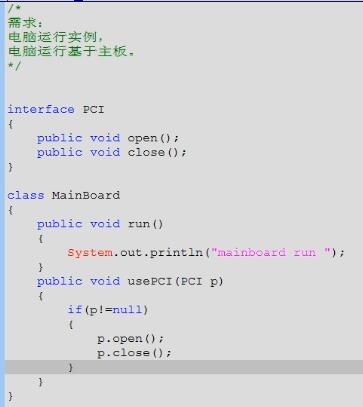




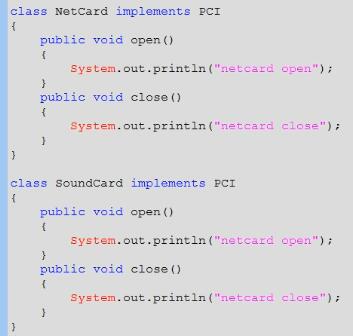
因为在方法区中，静态方法在静态区，属于类名调用，也就是静态绑定。成员方法在非静态区，属于对象调用，也就是动态绑定。

**P104 面向对象（多态的主板示例）**

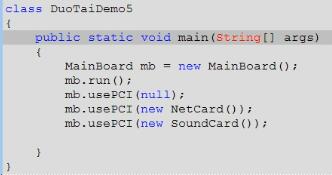
功能实现的主体



后期功能的扩展

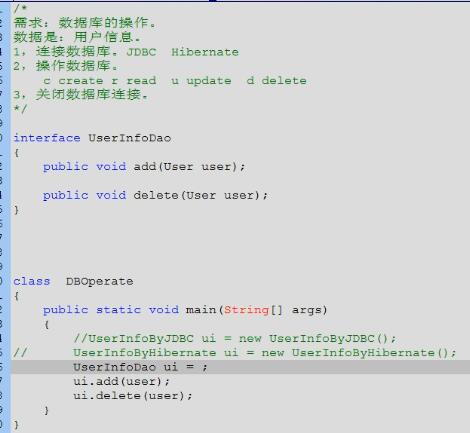


主函数运行



**P105 面向对象（多态的扩展示例）**

功能实现主体



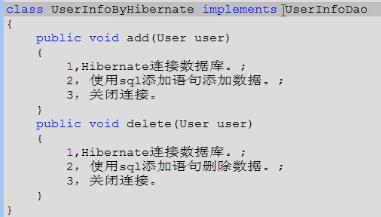


然后若干年后出现JDBC连接，只需让其实现UserInfoDao，并重写抽象方法。最后在功能主体new一个对象即可：

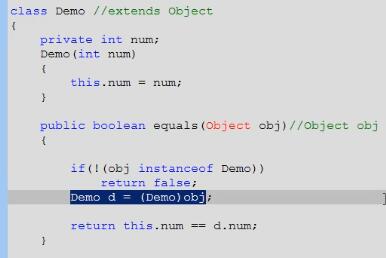
UserInfiDao ui = new UserInfoByJDBC();

又过了几年，出现了Hibernate，只需让其实现UserInfoDao，并重写抽象方法。最后在功能主体new一个对象即可：

UserInfiDao ui = new UserInfoByHibernate ();



**P106 面向对象（Object类-equals()）**



自己在重写equals方法时，注意不要忘记强制转换，因为父类的Object没有num这个成员变量。这么编译会报错说找不到d.num。

我们可以看到，父类Object自带equals方法，不过该比较方法往往意义不大，所以可以重写自己想要的结果。方法的传入参数任为Object类型是为了能和任何的对象进行比较，而不是只能同类间比较。通过一个判断语句：不同类直接返回false，同类则强制转换，最后返回转换后进行比较的结果。

**P107 面向对象（Object类toString()）**

//