第6章　多态性

本章知识点

- 多态性（ instanceof 运算符）
- 抽象类
- 接口

6.1 多态

- 多态的基本概念
- 多态就是指同一种事物表现出来的多种形态。

例如：

饮料：可乐、雪碧、红牛、芬达、.......

宠物：狗、猫、鸟、......

整数：byte b=10；short s=10；int i=10；......

6.1 多态

- 多态的种类

C++多态分为两种：编译多态、运行时多态。

Java中的多态只有一种，就是运行时多态。是一种运行期间的行为，而不是编译期间的行为。

6.1 多态

- 多态的语法格式

父类类型 引用 = new 子类类型();

例如：父类：人类； 子类：学生

```
Person p=new Student();
```

课堂小案例：

自定义Person类实现封装，特征有：姓名和年龄

自定义Student类实现封装，特征有：学号

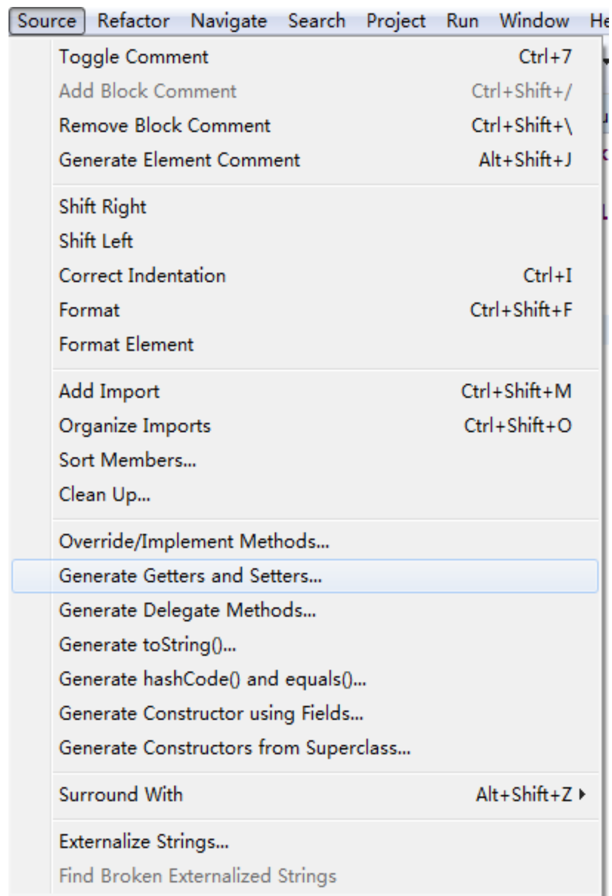
自定义TestPersonStudnet类，在main()方法中使用Person类的引用指向Student类的对象。

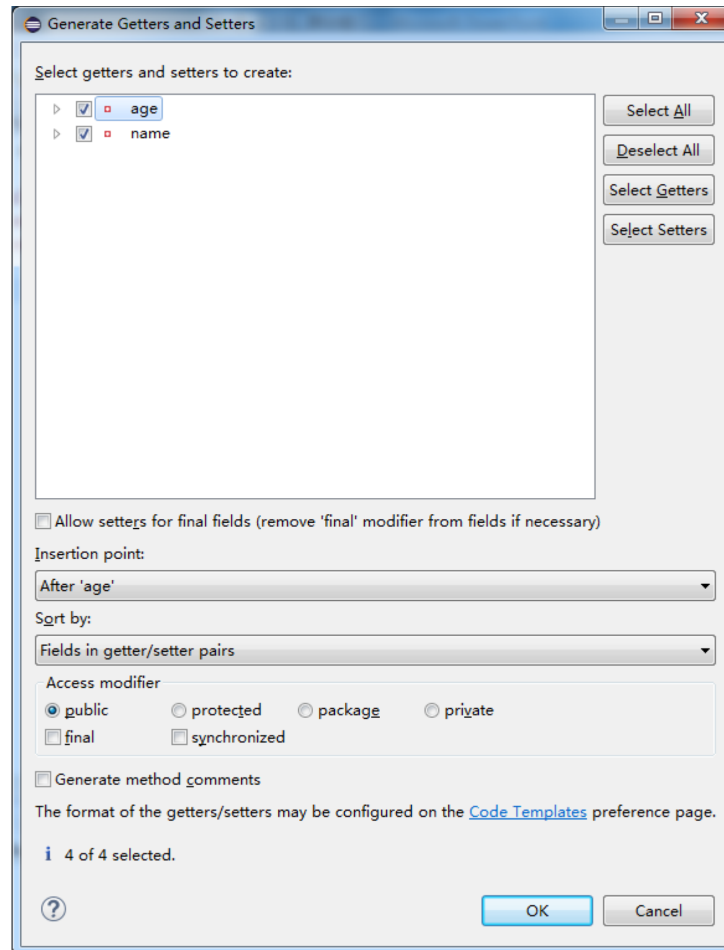
6.1 多态

```
public class Person {
private String name;
private int age;
//alt+s 生成get、set方法以及有参五参构造


}
```

6.1 多态

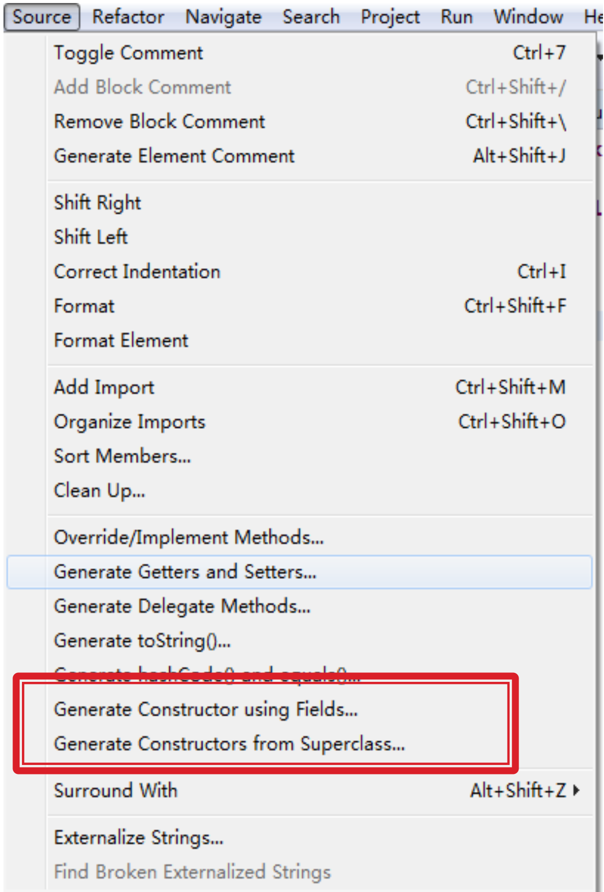


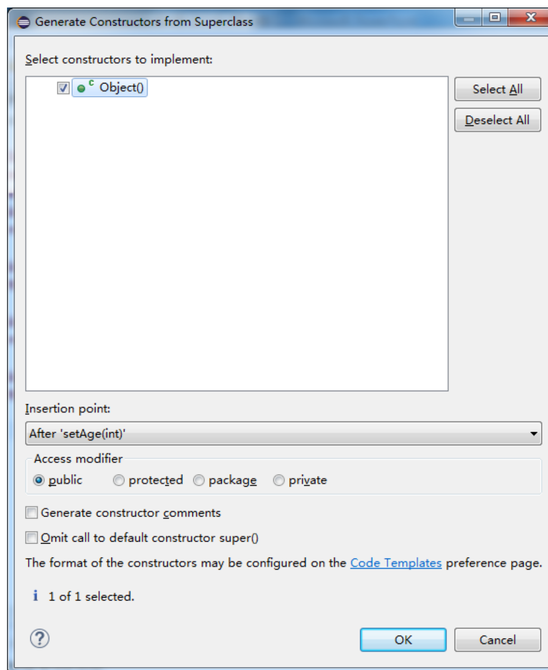


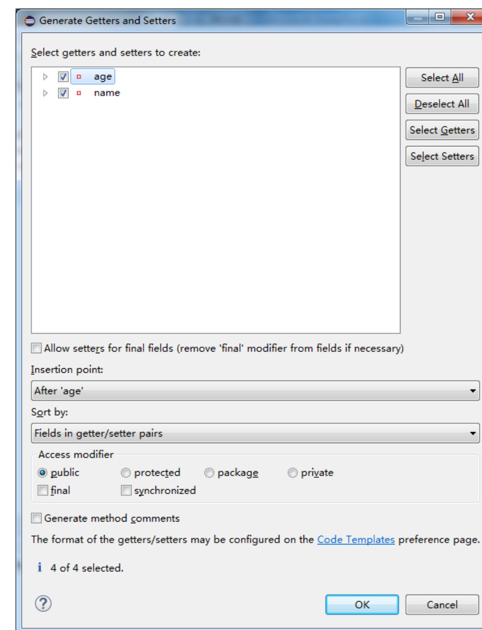
6.1 多态

```
public class Person {
private String name;
private int age;
public String getName() {
return name;}
public void setName(String name) {
this.name = name;}
public int getAge() {
return age;}
public void setAge(int age) {
this.age = age;}}
```

6.1 多态







```
public class Person {
private String name;
private int age;
public String getName() {return name;}
public void setName(String name) {
this.name = name;}
public int getAge() {return age;}
public void setAge(int age) {
this.age = age;}
public Person() {}
public Person(String name, int age) {super();
this.name = name;
this.age = age;
}  }
```

```
public class Person {---Person.java
private String name;
private int age;
public String getName() {return name;}
public void setName(String name) {
this.name = name;}
public int getAge() {return age;}
public void setAge(int age)
  { if(age>0&&age<120){this.age =
 age;}else{System.out.println("年龄不合理");}}
public Person() {}
public Person(String name, int age) {super();
this.name = name;this.age = age;}
public void show(){ System.out.println("我是
 "+getName()+",今年"+getAge()+"岁了！");}
}
```

```
public class Student extends Person{ --Student.java
private int id;
public Student() {super();}//生成无参构造
public Student(String name, int age) {
super(name, age);}
public Student(int id) {super(); this.id = id;}//生产有参构造

public int getId() {return id;}//生成get和set方法
public void setId(int id) {this.id=id;}
}
```

```
public class Student extends Person{ --Student.java
private int id;
public Student() {super();}//生成无参构造
public Student(String name, int age,int id) {
super(name, age);  setId(id);}
public Student(int id) {super(); this.id = id;}//生产有参构造

public int getId() {return id;}//生成get和set方法
public void setId(int id) {if(id>0){
        this.id=id;   }else{System.out.println("学号不合理！！");}}
}
```

```
public class testPersonStudent {
	public static void main(String[] args){
		//使用父类的引用指向父类自己的对象
		Person p =new Person();
		//调用Person类自己的show()方法
		p.show();
		System.out.println("-------------");
		//使用子类的引用指向子类自己的对象
		Student s =new Student();
//当子类中没有show()方法时，则调用父类Person的show()方法
		s.show(); }
	}
```

```java
public class testPersonStudent {
	public static void main(String[] args){
	//使用父类的引用指向父类自己的对象
	Person p =new Person();
	//调用Person类自己的show()方法
	p.show();
	System.out.println("-------------");
	//使用子类的引用指向子类自己的对象
	Student s =new Student();
//当子类中没有show()方法时，则调用父类Person的show()方法

		s.show();
	}
}
```

```
我是null,今年0岁了！
-----------------
我是null,今年0岁了！
```

```
public class testPersonStudent {
    public static void main(String[] args){
    //使用父类的引用指向父类自己的对象
    Person p =new Person();
    //调用Person类自己的show()方法
    p.show();
    System.out.println("-------------");
    //使用子类的引用指向子类自己的对象
    Student s =new Student();
//当子类中没有show()方法时，则调用父类Person的show()方法
//当子类中重写show()方法后，则调用子类Student自己的show()方法
        s.show();
    }
}
```

```
public class Student extends Person{ --Student.java
private int id;
public Student() {super();}//生成无参构造
public Student(String name, int age,int id) {
super(name, age);  setId(id);}
public Student(int id) {super(); this.id = id;}//生产有参构造

public int getId() {return id;}//生成get和set方法
public void setId(int id) {if(id>0){
        this.id=id;   }else{System.out.println("学号不合理！！");}}
 public void show(){super.show();
   System.out.println("学号: "+getId());}
}
```

```
public class testPersonStudent {
	public static void main(String[] args){
	//使用父类的引用指向父类自己的对象
	Person p =new Person();
	//调用Person类自己的show()方法
	p.show();
	System.out.println("-------------");
	//使用子类的引用指向子类自己的对象
	Student s =new Student();
//当子类中没有show()方法时，则调用父类Person的show()方法
//当子类中重写show()方法后，则调用子类Student自己的show()方法
		s.show();
	}
}
```

```
我是null,今年0岁了!
------------------
我是null,今年0岁了!
学号: 0
```

```
public class testPersonStudent {
        public static void main(String[] args){
            //使用父类的引用指向父类自己的对象
            Person p =new Person();
            //调用Person类自己的show()方法
            p.show();       System.out.println("----            ");
            //使用子类的引用指向子类自己的对象
            Student s =new Student();
//当子类中没有show()方法时，则调用父类Person的show()方法
//当子类中重写show()方法后，则调用子类Student自己的show()方法
            s.show();       System.out.println("--------------");
//使用父类引用指向子类对象，形成多态
  Person ps=new Student("张三",18,1010);
            ps.show();
        }     }
```

思考：ps调用的show()方法到底是Person类的还是Student类的???

```
我是null,今年0岁了！
--------------------
我是null,今年0岁了！
学号: 0
--------------------
我是张三,今年18岁了！
学号: 1010
```

请注意：观察变化

现在我们把父类中show()方法去掉

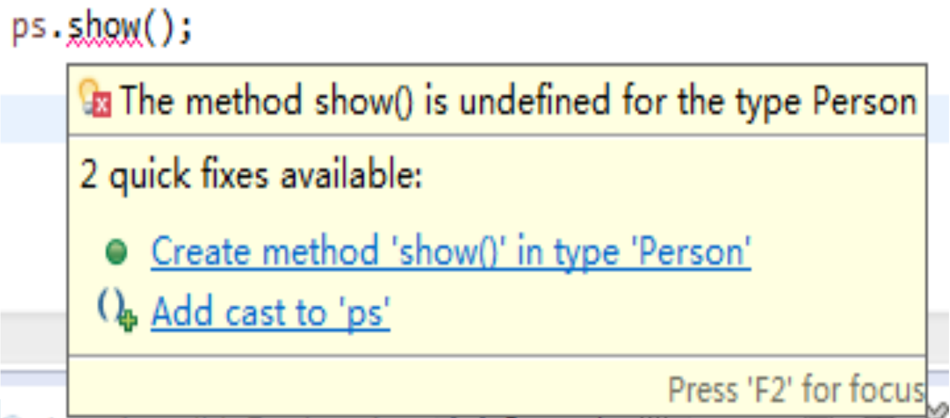
```
public class Person {---Person.java
private String name;
private int age;
public String getName() {return name;}
public void setName(String name) {
this.name = name;}
public int getAge() {return age;}
public void setAge(int age) { if(age>0&&age<120){this.age
  = age;}else{System.out.println("年龄不合理");}}
public Person() {}
public Person(String name, int age) {super();
this.name = name;this.age = age;}
//public void show(){ System.out.println("我是
  //"+getName()+",今年"+getAge()+"岁了！");}
}
```

```
public class Student extends Person{ --Student.java
private int id;
public Student() {super();}//生成无参构造
public Student(String name, int age,int id) {
super(name, age);  setId(id);}
public Student(int id) {super(); this.id = id;}//生产有参构造

public int getId() {return id;}//生成get和set方法
public void setId(int id) {if(id>0){
        this.id=id;   }else{System.out.println("学号不合理！！
  ");}}
 public void show(){  //super.show();
   System.out.println("学号: "+getId());}
}
```

```
public class testPersonStudent {
        public static void main(String[] args){
        Person p =new Person();//使用父类的引用指向父类自己的对象
        p.show();//调用Person类自己的show()方法
        System.out.println("------------");
    Student s =new Student();//使用子类的引用指向子类自己的对象

//当子类中没有show()方法时，则调用父类Person的show()方法
//当子类中重写show()方法后，则调用子类Student自己的show()方法
        s.show();
       System.out.println("------------");
Person ps=new Student("张三",18,1010);//使用父类引用指向子类对象，形成多态
            ps.show(); } }
```



```
public class testPersonStudent {
        public static void main(String[] args){
        Person p =new Person();//使用父类的引用指向父类自己的对象
        p.show();//调用Person类自己的show()方法
        System.out.println("-------------");
    Student s =new Student();//使用子类的引用指向子类自己的对象
//当子类中没有show()方法时，则调用父类Person的show()方法
//当子类中重写show()方法后，则调用子类Student自己的show()方法
        s.show();
       System.out.println("-------------");
Person ps=new Student("张三",18,1010);//使用父类引用指向子类对象，形成多态
              ps.show(); } }
```

思考：ps调用的show()方法到底是Person类的还是Student类的???

解析：在编译期间调用Person类的show()方法，在运行阶段调用Student类的show()方法

```
public class testPersonStudent {
        public static void main(String[] args){
        Person p =new Person();//使用父类的引用指向父类自己的对象
        p.show();//调用Person类自己的show()方法
        System.out.println("-------------");
    Student s =new Student();//使用子类的引用指向子类自己的对象
//当子类中没有show()方法时，则调用父类Person的show()方法
//当子类中重写show()方法后，则调用子类Student自己的show()方法
        s.show();
       System.out.println("-------------");
Person ps=new Student("张三",18,1010);//使用父类引用指向子类对象，形成多态
            ps.show(); } }
```

思考：ps调用的show()方法到底是Person类的还是Student类的?
解析：在编译期间调用Person类的show()方法，在运行阶段调用Student类的show()方法。
发生：Student类型向Person类型的转换，小范围向大范围的转换，属于自动类型转换。

6.1 多态

- 先来看一个例子，预测下程序运行的结果。
- 多态性(Polymorphism)
  - 在父类中定义的行为，被子类继承之后，表现出不同的行为。
  - 效果：同一行为在父类及其各个子类中具有不同的语义。

6.1.1 多态性

- 引用变量既可以指向相同类型的类的对象，也可以指向该类的任何一个子类的对象。

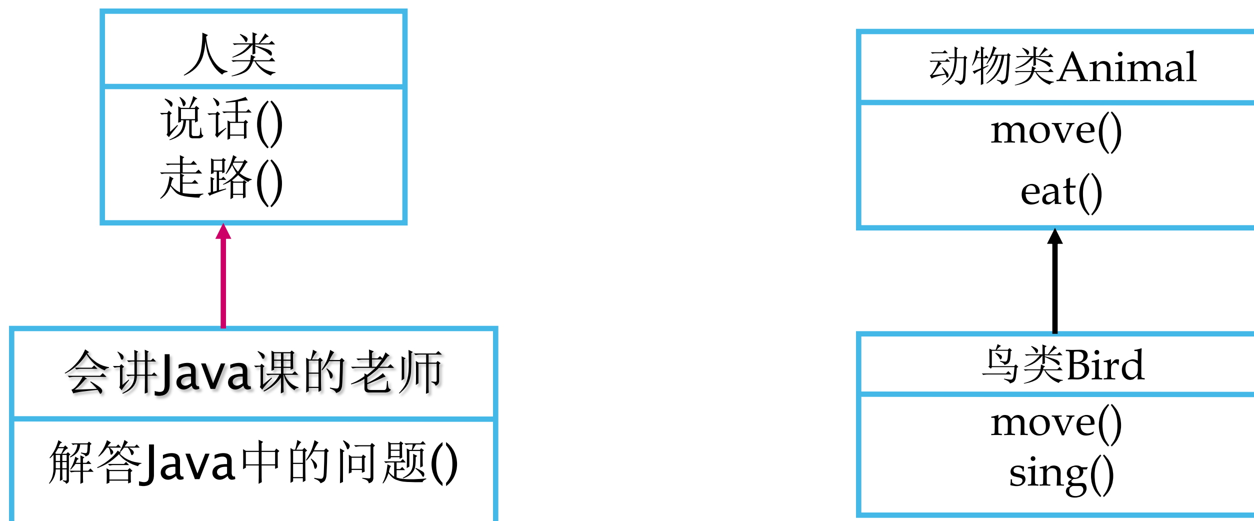
```
public class Test {
    public static void main(String[] args) {
        //子类对象送给父类引用
        Animal a = new Bird();
        ……
    }
}
```

Java语言中Object是所有类的直接或间接父类，也就是说，

任何类型的对象都可以赋值给Object引用。

6.1.1 多态性

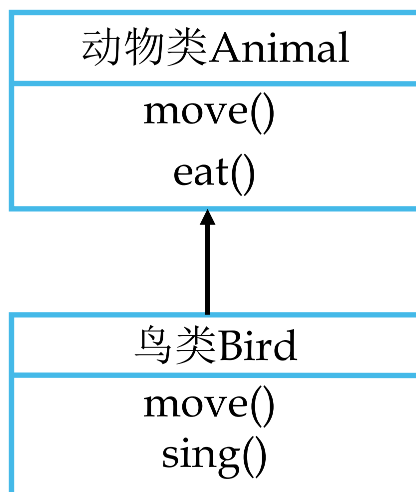
【说明 】如果把子类对象赋给父类引用（将子类对象当作父类对象看待），那么就只能调用父类中已有的方法。



Animal a = new Bird();
引用变量a可以调用的方法有哪些？

6.1.1 多态性

【说明 】如果子类把父类方法覆盖了，再把子类对象赋给父类引用，通过父类引用调用该方法时，调用的是子类重写之后的方法。

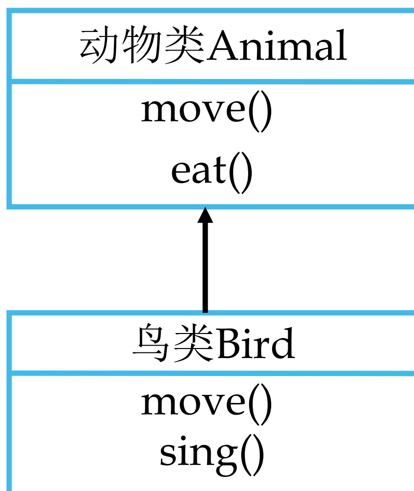



```
public class Test {
    public static void main(String[] args) {
        //子类对象送给父类引用
        Animal a = new Bird();
        a.move();
    }
}
```

Animal a = new Bird();
a.move()执行的是谁的move()方法？

6.1.1 多态性

- 举例



- 子类对象赋给父类引用后的3个层次

(1)父类中没有的方法(如sing()方法)不能调用。

(2)如果子类没有覆盖父类的方法(如eat()方法)，则调用父类的方法。

(3)如果子类覆盖父类的方法(如move()方法)，则调用子类的方法。

```
Bird b =new Bird("bird1", 4);
b.move();
b.sing();
b.eat();
```

```
Animal a1=new Animal("animal1",20);
a1.move();
a1.eat();
```

```
Animal a2=new Bird("bird2",4);
a2.sing();    //出错，父类中没有该方法
a2.eat();     //执行父类eat方法
a2.move();    //执行子类move方法
```

6.1.2 静态绑定和动态绑定

Person ps=new Student("张三",18,1010);

ps 非静态方法的调用

```
public class testPersonStudent {
        public static void main(String[] args){
        Person p =new Person();//使用父类自
        p.show();//调用Person类自己的show
        System.out.println("------------")
    Student s =new Student();//使用子类的
//当子类中没有show()方法时，则调用父类Pe
//当子类中重写show()方法后，则调用子类St
        s.show();     System.out.println("---
   Person ps=new Student("张三",18,1010);//使用父类引用指向子类对象，形成多态
        ps.show();   System.out.println("------------");
     String str=ps.getName();//使用ps调用非静态方法测试
System.out.println("获取的姓名是："+str); } }
```

```
我是null,今年0岁了！
----------------
我是null,今年0岁了！
学号: 0
----------------
我是张三,今年18岁了！
学号: 1010
----------------
获取的姓名是：张三
```

Ps是Person类型的引用，因此可以调用Person类自己的方法

```
public class testPersonStudent {
        public static void main(String[] args){
        Person p =new Person();//使用父类的
        p.show();//调用Person类自己的show
        System.out.println("-------------")
    Student s =new Student();//使用子类的
//当子类中没有show()方法时，则调用父类Pe
//当子类中重写show()方法后，则调用子类St
        s.show();     System.out.println("---                    ");
   Person ps=new Student("张三",18,1010);//使用父类引用指向子类对象，形成多态
        ps.show();   System.out.println("-------------");
    String str=ps.getName();//使用ps调用非静态方法测试
    System.out.println("获取的姓名是："+str);
```

```
     ps.getId();// } }
```

```
我是null,今年0岁了！
------------------
我是null,今年0岁了！
学号: 0
------------------
我是张三,今年18岁了！
学号: 1010
------------------
获取的姓名是：张三
```

ps是Person类型的引用，因此可以调用Person类自己的方法

6.1.2 静态绑定和动态绑定

Person ps=new Student("张三",18,1010);

ps 静态方法的调用

```
public class Person {---Person.java
private String name;
private int age;
public String getName() {return name;}
public void setName(String name) {
this.name = name;}
public int getAge() {return age;}
public void setAge(int age) { if(age>0&&age<120){this.age = age;}else{System.out.println("年龄不合理");}}
public Person() {}
public Person(String name, int age) {super();
this.name = name;this.age = age;}
public void show(){ System.out.println("我是"+getName()+",今年"+getAge()+"岁了！");}
public static void test(){System.out.println("Person类的静态方法");}        }
```

```
public class Student extends Person{ --Student.java
private int id;
public Student() {super();}//生成无参构造
public Student(String name, int age,int id) {
super(name, age);  setId(id);}
public Student(int id) {super(); this.id = id;}//生产有参构造

public int getId() {return id;}//生成get和set方法
public void setId(int id) {if(id>0){
        this.id=id;   }else{System.out.println("学号部合理！！
   ");}}
 public void show(){  super.show();
    System.out.println("学号: "+getId());}
   @override
```

```
   public static void test(){System.out.println("Student类
                          的静态方法");} //静态方法不能重写 }
```

```
public class Student extends Person{ --Student.java
private int id;
public Student() {super();}//生成无参构造
public Student(String name, int age,int id) {
super(name, age);  setId(id);}
public Student(int id) {super(); this.id = id;}//生产有参构造

public int getId() {return id;}//生成get和set方法
public void setId(int id) {if(id>0){
        this.id=id;   }else{System.out.println("学号部合理！！
  ");}}
 public void show(){  super.show();
   System.out.println("学号: "+getId());}
   //@override
   public static void test(){System.out.println("Student类
                         的静态方法");} //静态方法不能重写 }
```

```
public class testPersonStudent {
        public static void main(String[] args){
        Person p =new Person();//使用父类的引 ... 象
        p.show();//调用Person类自己的show()方
        System.out.println("-------------");
    Student s =new Student();//使用子类的引
//当子类中没有show()方法时，则调用父类Pers
//当子类中重写show()方法后，则调用子类Stuc
        s.show();     System.out.println("-----
   Person ps=new Student("张三",18,1010);/ ... 类
对象，形成多态
        ps.show();   System.out.println("-------------");
     String str=ps.getName();//使用ps调用非静态方法测试
     System.out.println("获取的姓名是："+str);
      //ps.getId();
     System.out.println("-------------");
      ps.test();//出现黄色警告，使用ps调用静态方法进行测试，静态的成员推
荐使用类名.的方式      }}
```

```
我是null,今年0岁了！
-------------------
我是null,今年0岁了！
学号：0
-------------------
我是张三,今年18岁了！
学号：1010
-------------------
获取的姓名是：张三
-------------------
Person类的静态方法
```

```
public static void main(String[] args){
    Person p =new Person();//使用父类的引 象
    p.show();//调用Person类自己的show()方
    System.out.println("-------------");
  Student s =new Student();//使用子类的引用
//当子类中没有show()方法时，则调用父类Perso
//当子类中重写show()方法后，则调用子类Stude
    s.show();    System.out.println("------
 Person ps=new Student("张三",18,1010);//  类
对象，形成多态
    ps.show();   System.out.println("------
   String str=ps.getName();//使用ps调用非静态方法测试
   System.out.println("获取的姓名是："+str);
    //ps.getId();
   System.out.println("-------------");
    ps.test();//出现黄色警告，使用ps调用静态方法进行测试，静态的成员推
荐使用类名.的方式
              Person.test(); } }
```

```
我是null,今年0岁了！
------------------
我是null,今年0岁了！
学号：0
------------------
我是张三,今年18岁了！
学号：1010
------------------
获取的姓名是：张三
------------------
Person类的静态方法
Person类的静态方法
```

总结：

如：Person ps=new Student();

ps.show();

解析：

在编译阶段ps是Person类型，因此调用Person类自己的show()方法，若没有编译时报错。

在运行阶段ps真正指向的对象是Student类型，因此最终调用的是Student类中自己的show()方法。

当使用多态方式调用方法的时候，首先会检查父类中是否有该方法，没有则编译报错。如果有再去调用子类的同名方法。（注意:静态的static方法属于特殊情况，静态方法只能继承，不能被重写override，如果子类定义了同名同形式的静态方法，他对父类方法只能起到隐藏的作用。调用的时候用谁的引用，则调用谁的版本）

多态存在的三个必要条件

- 要有继承
- 要有重写
- 父类引用指向子类对象

如：Person ps=new Student();

ps.show();

如何去实现子类中方法的调用呢？

ps.getId();//error

```
public class testPersonStudent {
        public static void main(String[] args){
        Person p =new Person();//使用父类的引用
        p.show();//调用Person类自己的show()方法
        System.out.println("--------------");
      Student s =new Student();//使用子类的引用
        s.show();     System.out.println("--------
    Person ps=new Student("张三",18,1010);//使
对象，形成多态
        ps.show();   System.out.println("--------
    String str=ps.getName();//使用ps调用非静态方法测试
    System.out.println(“获取的姓名是：”+str);
     //ps.getId();
    System.out.println("--------------");
     ps.test();//出现黄色警告，使用ps调用静态方法进行测试，静态的成员推
荐使用类名.的方式
                              Person.test(); } }
```

```
我是null,今年0岁了！
--------------------
我是null,今年0岁了！
学号: 0
--------------------
我是张三,今年18岁了！
学号: 1010
--------------------
获取的姓名是：张三
--------------------
Person类的静态方法
Person类的静态方法
```

```
public class testPersonStudent {
    public static void main(String[] args){
    Person p =new Person();//使用父类的引用指
    p.show();//调用Person类自己的show()方法
    System.out.println("-------------");
  Student s =new Student();//使用子类的引用指
    s.show();    System.out.println("---------
 Person ps=new Student("张三",18,1010);//使用
对象，形成多态
    ps.show();  System.out.println("---------
   String str=ps.getName();//使用ps调用非静态方法测试
   System.out.println("获取的姓名是："+str);
    //ps.getId();//Person类型向Student类型转换
   System.out.println("-------------");
    ps.test();//出现黄色警告，使用ps调用静态方法进行测试，静态的成员推
荐使用类名.的方式
                    Person.test(); } }
```

```
我是null,今年0岁了！
-------------------
我是null,今年0岁了！
学号: 0
-------------------
我是张三,今年18岁了！
学号: 1010
-------------------
获取的姓名是：张三
-------------------
Person类的静态方法
Person类的静态方法
```

```
public class testPersonStudent {
        public static void main(String[] args){
        Person p =new Person();//使用父类的引用
        p.show();//调用Person类自己的show()方
        System.out.println("-------------");
    Student s =new Student();//使用子类的引用
       s.show();     System.out.println("-------
   Person ps=new Student("张三",18,1010);//使
象，形成多态
        ps.show();   System.out.println("-------
     String str=ps.getName();//使用ps调用非静态方法测试
     System.out.println(“获取的姓名是：”+str);
Student st=(Studnet) ps；int rs=st.getId();
System.out.println(“id=”+rs);
System.out.println("-------------"); ps.test();//出现黄色警告，使用ps
用静态方法进行测试，静态的成员推荐使用类名.的方式
                              Person.test(); } }
```

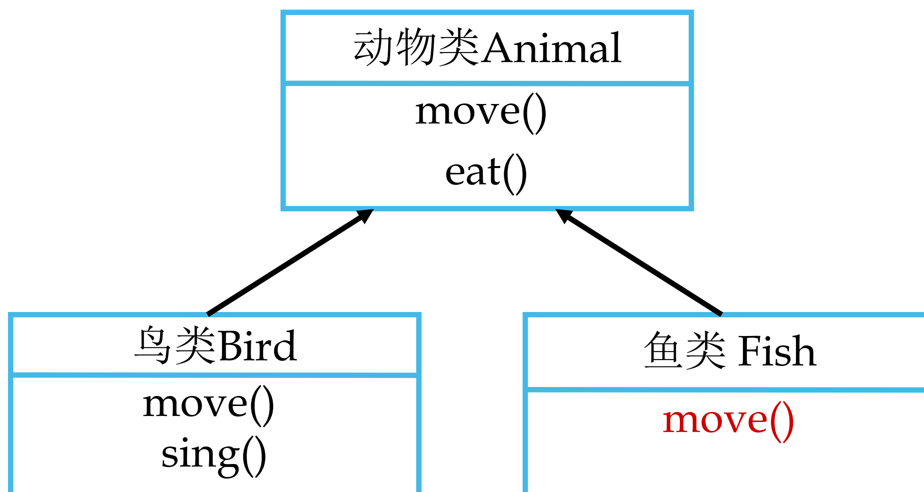
```
我是null,今年0岁了！
-------------------
我是null,今年0岁了！
学号：0
-------------------
我是张三,今年18岁了！
学号：1010
-------------------
获取的姓名是：张三
学号：1010
-------------------
Person类的静态方法
Person类的静态方法
```

建立Teacher类并继承Person

Teacher t=(Teacher)ps;//编译没有保持，但运行的时候报错。

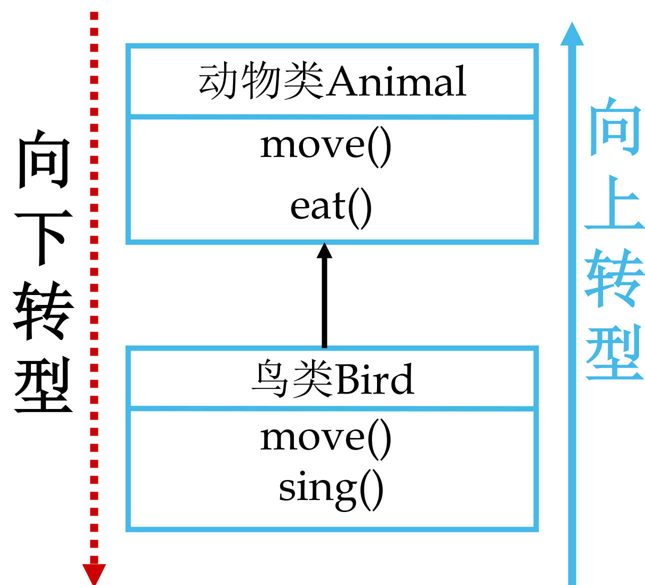
6.1.2 静态绑定和动态绑定

- 动态绑定的作用：无需对现存的代码进行修改，就可以对程序进行扩展。
- 举例



6.1.3 instanceof运算符

- Java编译器允许在具有直接或间接继承关系的类之间进行类型转换。
  - 子类对象->父类引用(向上转型)
  - 父类引用->子类引用(向下转型)



6.1.3 instanceof运算符

- 向下转型
  - 必须使用强制类型转换
  - 必须在有意义的情况下进行，即强转必须是合理的
  - 编译器对于强制类型转换采取的是一律放行的原则（只检查语义）

```
Bird bird = (Bird) xxx;
```

无论xxx是哪种类型，编译器都不会报错。

```
Animal   animal = new Animal();
Bird bird = (Bird)animal;
```

编译无错，运行出错， ClassCastException异常

6.1.3 instanceof运算符

```
if (ps instanceof Teacher){
System.out.println( “可以放心的进行强制转换了…”);
else{System.out.println( “不能进行强制转换”);
}

}
```

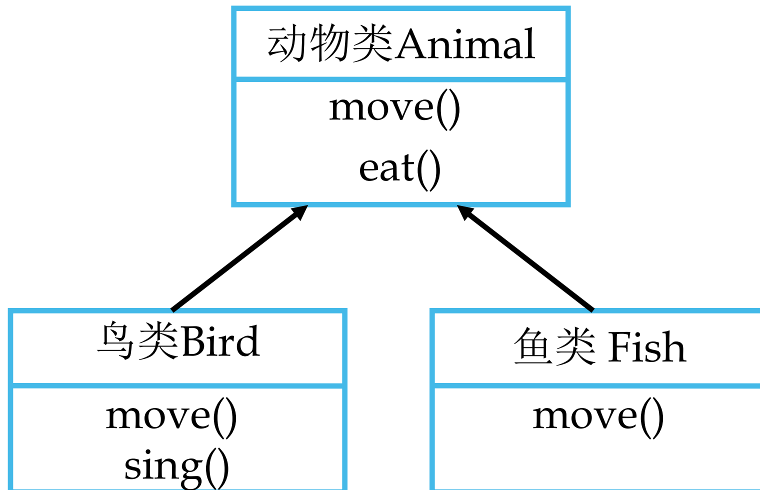
```
public class testPersonStudent {
        public static void main(String[] args){
        Person p =new Person();//使用父类的引
        p.show();//调用Person类自己的show()方
        System.out.println("-------------");
    Student s =new Student();//使用子类的引用
       s.show();     System.out.println("------
   Person ps=new Student("张三",18,1010);//
象，形成多态
        ps.show();   System.out.println("------
     String str=ps.getName();//使用ps调用非静态方法测试
     System.out.println("获取的姓名是：" +str);
Student st=(Studnet) ps；int rs=st.getId();
System.out.println("id="+rs);
System.out.println("-------------"); ps.test();//出现黄色警告，使用ps
用静态方法进行测试，静态的成员推荐使用类名.的方式
                              Person.test();
                    if (ps instanceof Teacher){…}
}}
```

```
我是null,今年0岁了！
------------------
我是null,今年0岁了！
学号: 0
------------------
我是张三,今年18岁了！
学号: 1010
------------------
获取的姓名是：张三
学号: 1010
------------------
Person类的静态方法
Person类的静态方法
不能进行强制类型转换
```

```
public class testPersonStudent {
        public static void main(String[] args){
        Person p =new Person();//使用父类的引用
        p.show();//调用Person类自己的show()方法
        System.out.println("--------------");
    Student s =new Student();//使用子类的引用指
       s.show();      System.out.println("--------
   Person ps=new Student("张三",18,1010);//使
象，形成多态
        ps.show();   System.out.println("--------
     String str=ps.getName();//使用ps调用非静态
     System.out.println("获取的姓名是："+str);
Student st=(Studnet) ps；int rs=st.getId();
System.out.println("id="+rs);
System.out.println("--------------"); ps.test();//出现黄色警告，使用ps
用静态方法进行测试，静态的成员推荐使用类名.的方式
                              Person.test();
                    if (ps instanceof Student){...}
```

```
我是null,今年0岁了！
--------------------
我是null,今年0岁了！
学号：0
--------------------
我是张三,今年18岁了！
学号：1010
--------------------
获取的姓名是：张三
学号：1010
--------------------
Person类的静态方法
Person类的静态方法
可以放心的进行强制类
型转换了...
```

6.1.3 instanceof运算符



- Animal a=new Fish();
- Animal b=new Bird();
- Animal c=new Animal();

- if(a instanceof Animal) ?
- if(b instanceof Animal) ?
- if(c instanceof Animal) ?
- if(a instanceof Bird) ?
- if(b instanceof Fish) ?
- if(c instanceof Fish) ?

6.1.3 instanceof运算符

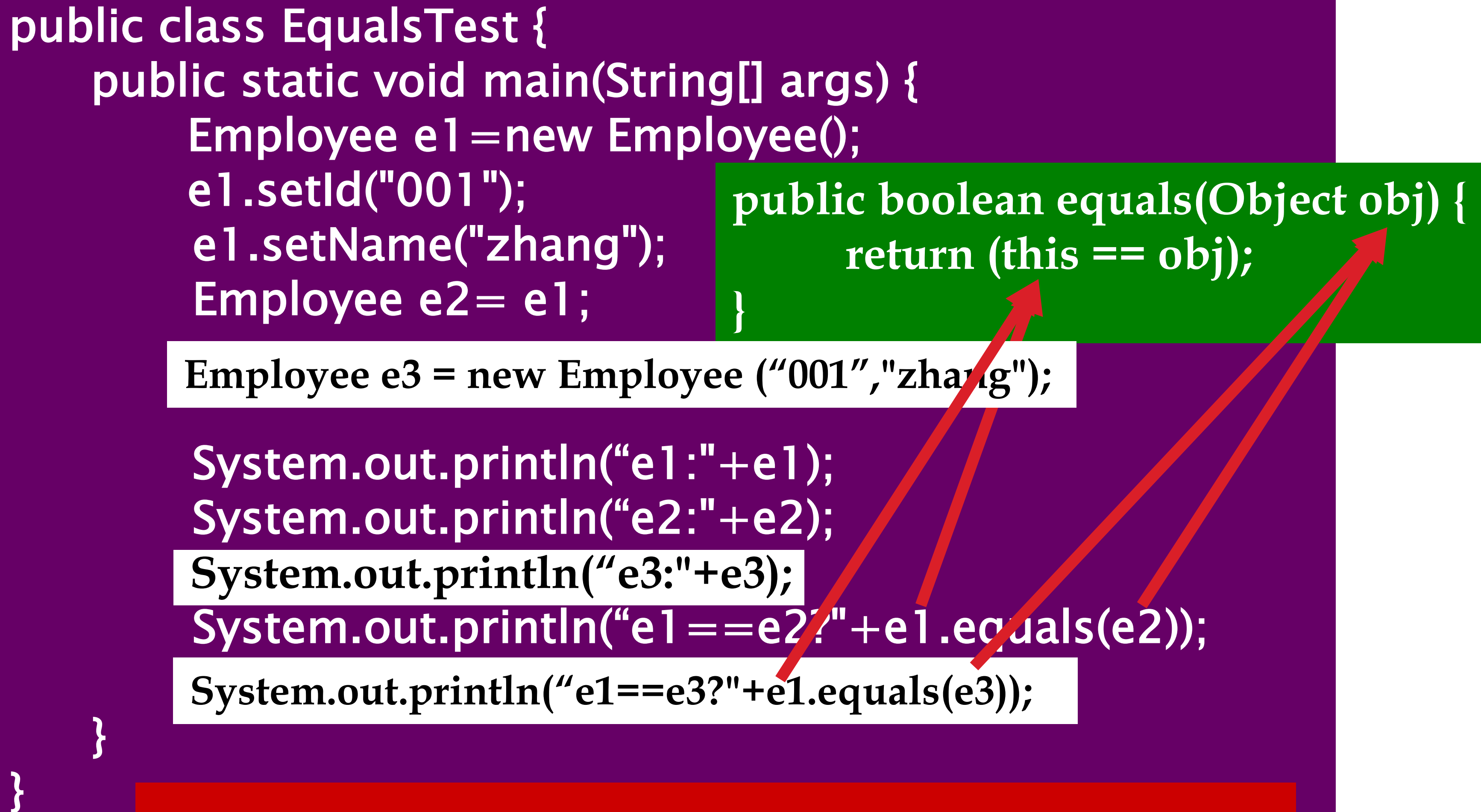
【例6-1】在Employee类中重写java.lang.Object中的equals()方法。设有员工Employee类，包含工号id、姓名name、工资salary等属性。当工号id与姓名name均相同时，两个对象相等。

Object类中的equals()方法

```
public boolean equals(Object obj){
        return this==obj;
}
```

功能：比较参数所指定的对象是否与当前对象“相等”。对于任何非空引用变量x 和 y，当且仅当 x 和 y 引用同一个对象时，此方法返回 true。

6.1.3 instanceof运算符



6.1.3 instanceof运算符

- Employee类对象相等的条件是Id和name都相同。

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

6.1.3 instanceof运算符

- Emp

```
public b
        ret
}
```

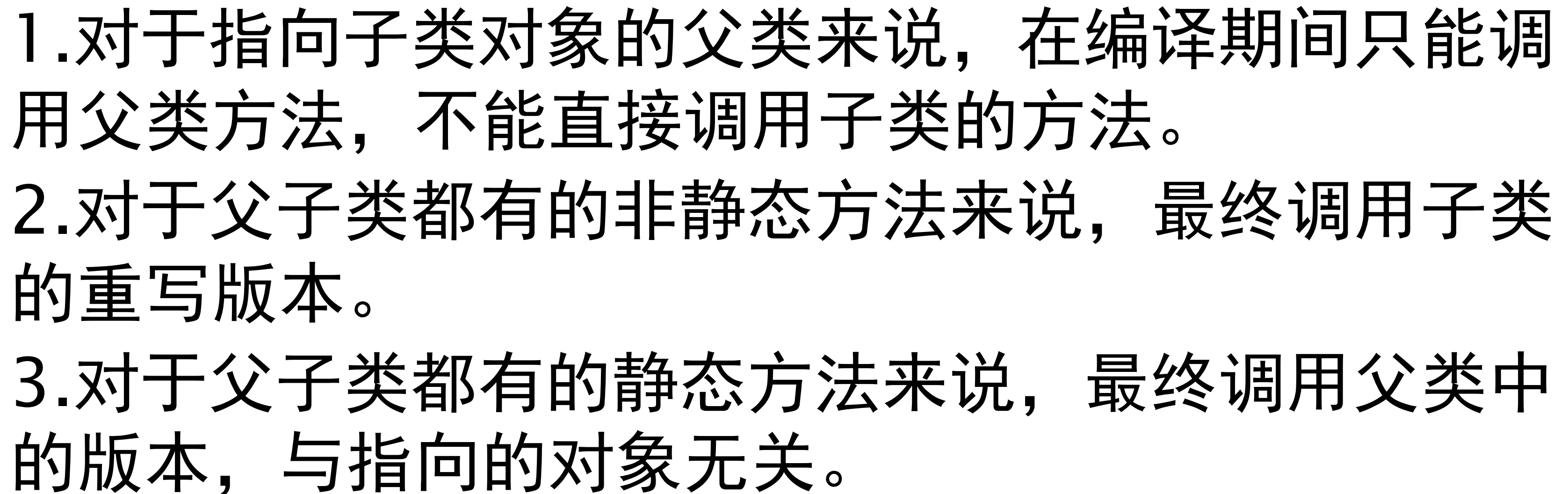
```
public class EqualsTest {
    public static void main(String[] args) {
        Employee e1=new Employee ("001","zhang");
        Employee e2=new Employee ("001","zhang");
        System.out.println("e1==e2?"+e1.equals(e2));
    }
}
```

动态绑定

```
public boolean equals(Object obj){
    if( obj instanceof Employee ){
        Employee e = (Employee )obj;
        return this.name.equals(e.name) && this.id.equals(e.id);
    }
    return false;
}
```

向下转型

多态的效果



多态的实际意义

- 多态的意义在于：可以屏蔽不同子类的差异性编写通用代码，从而产生不同的效果。

案例：

矩形：横坐标、纵坐标、长度、宽度

圆形：横坐标、纵坐标、半径

写一个方法要求既能打印矩形又能打印圆形

多态的实际意义

- 多态的意义在于：可以屏蔽不同子类的差异性编写通用代码，从而产生不同的效果。

案例：

形状：横坐标、纵坐标

矩形：横坐标、纵坐标、长度、宽度

圆形：横坐标、纵坐标、半径

写一个方法要求既能打印矩形又能打印圆形

```
public class Shape {
private int x;
private int y;
public Shape() {}
public Shape(int x, int y) {setX(x);  setY(y);}
public int getX() {return x;}
public void setX(int x) {this.x = x;}
public int getY() {return y;}
public void setY(int y) {this.y = y;}
public void show(){
System.out.println("x="+x+",y="+y);}
}
```

```
public class Rect extends Shape{
private int len;
private int vid;
public Rect() {super();}
public Rect(int len, int vid,int x,int y) {
super(x,y); setLen(len);setVid(vid);}
public int getLen() {return len;}
public void setLen(int len) {this.len = len;}
public int getVid() {return vid;}
public void setVid(int vid) {this.vid = vid;}
@Override
public void show() {super.show();
System.out.println("len="+getLen()+",vid="+getVid());}
```

```
public class Cricle extends Shape {
private int r;
public Cricle() {super();}
public Cricle(int x, int y,int r)
{super(x,y);setR(r);}
public int getR() {return r;}
public void setR(int r) {this.r = r;}
@Override
public void show() {super.show();
System.out.println("r="+getR());}
}
```

```
public class Testa {
public static void main(String[] args){
Rect r=new Rect(1,2,3,4);
r.show(r);
Cricle c=new Cricle(5,6,7);
c.show(c);}
}
```

```
x=3,y=4
len=1,vid=2
x=5,y=6
r=7
```

多态的实际意义

- 多态的意义在于：可以屏蔽不同子类的差异性编写通用代码，从而产生不同的效果。

案例：

形状：横坐标、纵坐标

矩形：横坐标、纵坐标、长度、宽度

圆形：横坐标、纵坐标、半径

写一个方法要求既能打印矩形又能打印圆形

```
public class Testa {
public static void draw(Rect r){
r.show();}
public static void main(String[] args){
Rect r=new Rect(1,2,3,4);
Testa.draw(r);
Cricle c=new Cricle(5,6,7);
  }
}
```

```
public class Testa {
public static void draw(Rect r){
r.show();}
public static void draw2(Cricle c){
c.show();}
public static void main(String[] args){
Rect r=new Rect(1,2,3,4);
Testa.draw(r);
Cricle c=new Cricle(5,6,7);
Testa.draw2(c);}
}
```

```
x=3,y=4
len=1,vid=2
x=5,y=6
r=7
```

```
public class Testa {
public static void draw(Shape s){//当形参类型为Shape
//时，既可以接收矩形对象，同时又可以接收圆形对象
s.show();}//在编辑阶段调用的Shape类的show()方法，在运
//行阶段调用子类重写的方
public static void main(String[] args){
Rect r=new Rect(1,2,3,4);
Testa.draw(r);
Cricle c=new Cricle(5,6,7);
Testa.draw(c);
  }
}
```

```
x=3,y=4
len=1,vid=2
x=5,y=6
r=7
```

一个方法可以接收多个对象的时候，使用多态。

6.2 抽象类和抽象方法

抽象类—定义

- 用abstract关键字修饰的类称为抽象类。

```
[权限修饰符] abstract class  类名
{  成员变量;
   成员方法;
   构造方法;
   抽象方法  }
```

- 抽象类不能实例化。
- 抽象类的意义在于“被继承”，抽象类为其子类“抽象”出了公共部分，通常也定义了子类所必须具体实现的抽象方法。

抽象类

- 抽象方法

  不能被具体实现的方法，也就是没有方法体，并且使用抽abstract关键字修饰；

  语法格式：

  [访问修饰符] abstract 返回值类型 方法名（参数表）；

  例如： public abstract void show();

- 抽象类

  使用abstract关键字修饰的类，抽象类不能被实例化。

TestAbstract.java

```
public class TestAbstract {
private int num;
public TestAbstract(int num) {this.num = num;}
public TestAbstract() {}
public void show(){
System.out.println("show方法");
}
public static void main(String[] args){
TestAbstract t=new TestAbstract();
t.show();
}}
```

```
show方法
```

TestAbstract.java

```
public  abstract  class TestAbstract {
private int num;
public TestAbstract(int num) {this.num = num;}
public TestAbstract() {}
public void show(){
System.out.println("show方法");
}
public static void main(String[] args){
TestAbstract t=new TestAbstract();
t.show();
}}
```

Cannot instantiate the type TestAbstract

TestAbstract.java

```
public  abstract  class TestAbstract {
private int num;
public TestAbstract(int num) {this.num = num;}
public TestAbstract() {}
public void show(){
System.out.println("show方法");
}
public static void main(String[] args){
TestAbstract t=new TestAbstract();
t.show();
}}
```

TestAbstract.java

```
public abstract class TestAbstract {
private int num;
public TestAbstract(int num) {this.num = num;}
public TestAbstract() {}
public abstract void show();
public void show(){
System.out.println("show方法");
}
public static void main(String[] args){
TestAbstract t=new TestAbstract();
t.show();
}}
```

The abstract method show in type TestAbstract can only be defined by an abstract class

TestAbstract.java

```
public  abstract  class TestAbstract {
private int num;
public TestAbstract(int num) {this.num = num;}
public TestAbstract() {}
public abstract void show();
public void show(){
System.out.println("show方法");
}
public static void main(String[] args){
TestAbstract t=new TestAbstract();
t.show();
}}
```

抽象类—定义

- 用abstract关键字修饰的类称为抽象类。
- 抽象类不能实例化。
- 抽象类的意义在于“被继承”，抽象类为其子类“抽象”出了公共部分，通常也定义了子类所必须具体实现的抽象方法。

```
public class TestSubAbstract extends TestAbstract {

}
```

The type TestSubAbstract must implement the inherited abstract method TestAbstract.show()

```java
public class TestSubAbstract extends TestAbstract {
@Override
public void show() {System.out.println("抽象类果然不同");}
}
```

```
public class TestSubAbstract extends TestAbstract {
@Override
public void show() {System.out.println("抽象类果然不同");}
public static void main(String[] args){
TestSubAbstract ts= new TestSubAbstract();
ts.show();//子类的引用只能调用自己的show方法
System.out.println("---------------");
TestAbstract ta=new TestSubAbstract();//父类的引用指向子类的对象，形成多态
ta.show();//在编译阶段调用父类的show()，在运行阶段调用的子类重写父类以后的方法}
```

```
抽象类果然不同
---------------
抽象类果然不同
```

抽象类的注意事项

1.抽象类可以有成员变量、成员方法以及构造方法；
2.抽象类可以有抽象方法，也可以没有；
3.拥有抽象方法的类必须是抽象类，因此通常情况下认为拥有抽象方法并且有abstract，abstract关键字修饰的类才认为是真正的抽象类；

抽象类的意义

抽象类的意义不在于实例化而在于被继承，若一个类继承自抽象类必须重写抽象方法，否则该类也得变成抽象类。

因此抽象类对子类具有强制性和规范性，也叫做模板设计模式。

抽象类的意义

经验分享：

在以后的开发中推荐使用父类引用指向子类对象形式，因为父类引用直接调用的方法一定是父类拥有的方法，当我们需要更换指向子类对象的时候，只需要将new后面的该方式类型更改一下就可以了，其他的代码无需改动，因此提高了代码的可维护性以及以后的可扩展性。

该方式的缺点在于：父类引用不能直接调用子类独有的方法，若调用则需要强制类型转换。

练习

- 自定义Account类实现封装，特征：账户余额，提供计算利息并返回的抽象方法。
- 自定义FixAccount类继承Account类，实现抽象方法的重写。
- 自定义Test FixAccount,在main()方法中使用多态的语法创建对象并调用计算利息的方法。

抽象类的总结

抽象关键字 abstract

当abstract修饰类，就是抽象类，抽象类不能实例化；当abstract修饰方法时称方法为抽象方法，该方法没有方法体，继承抽象类，必然要重写抽象方法。

6.3 接口

接口的定义

- 接口可以看成是特殊的抽象类，即只包含有抽象方法的抽象类，例如：

通过关键字interface定义接口

```
Interface Runner{
        public static final int DEF_SPEED=100;
        void run();
}
```

接口中不可以定义成员变量，但可以定义常量

接口中只可以定义没有实现的方法（可以省略 public abstract）

接口基本概述

- 接口就是一种比抽象类还抽象的类，该类型不能实例化
- 定义类的关键字是class，而定义接口的关键字是interface
- 继承类的关键字是extends，而实现接口的关键字是implements
- 当多个类型之间具有相同行为能力的时候，java中就可以通过接口来进行类型之间的联系。

  通过接口可以解决java中单继承所带来的一些类型无法共享的问题。

接口基本概述

- 接口定义了某一些规范，并且需要遵守

  接口不关心类的内部数据和信息，也不关心这些类里方法的实现细节，它只规定这些类必须提供这些方法。

接口的语法格式

修饰符 interface 接口名称 [extends 父接口1，父类接口2….]{

零个到多个常量的定义……

零个到多个抽象方法的定义……

零个到多个默认方法的定义……（jdk1.8新特性）

零个到多个静态方法的定义…… （jdk1.8新特性）

}

接口

案例：

黄金

行为：买东西、发光

金属

行为：发光

货币

行为：买东西

接口

案例：

黄金

行为：买东西、发光

金属

行为：发光

货币

行为：买东西

```
public interface Money {
//自定义抽象方法用于描述买东西的行为
    public abstract void buy();
}
```

```
public interface Metal {
//自定抽象方法用来描述金属发光的行为
    public abstract  void shine();
}
```

```
public class Gold implements Money,Metal {
@Override
public void shine() {
System.out.println("发出了金黄色的光芒");
    }
@Override
public void buy() {
System.out.println("买了好多好吃的……");
    }
  }
```

TestGold.java

```
public class TestGold {
public static void main(String[] args){
//接口类型的引用指向了实现类的对象，形成多态
Money mn = new Gold();//
mn.buy();

	}
}
```

买了好多好吃的......

TestGold.java

```
public class TestGold {
public static void main(String[] args){
//接口类型的引用指向了实现类的对象，形成多态
Money mn = new Gold();//
mn.buy();
Metal m= new Gold();
m.shine();
	}
}
```

买了好多好吃的……
发出了金黄色的光芒

接口的注意事项

1.接口可以实现多继承，也就是一个接口可以同时继承多个父接口。

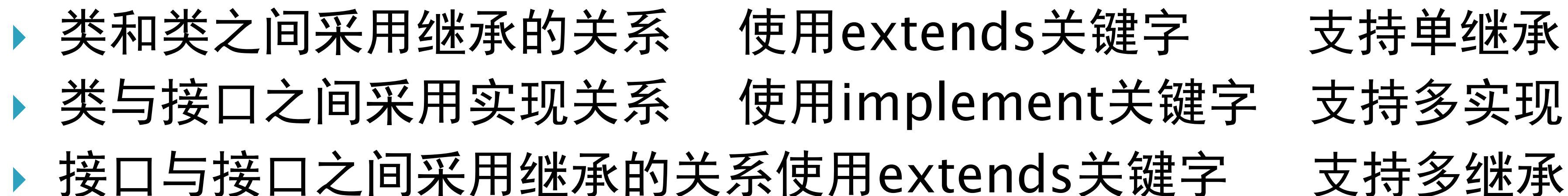
2.实现接口的类如果不能实现所有接口中待重写的方法，则必须设置为抽象类。

3.一个类可以继承一个父类，同时实现多个接口。

4.接口中的所有成员变量必须由public static final共同修饰，也就是常量。

5.接口中所有成员方法必须由public abstract共同修饰，也就是抽象方法。

类和接口之间的关系



抽象类和接口的主要区别

1.定义抽象类的关键字class，而定义接口的关键字interface。

2.继承抽象类关键字extends，而实现接口关键字implement。

3.继承抽象类支持单继承，而实现接口支持多实现

4.抽象类有构造方法，但接口没有。

5.接口中所有的成员变量都必须是常量，而抽象类中可以是变量。

6.接口中所有成员方法都必须是抽象方法，而抽象类中可以是普通方法。

7.接口中增加方法一定影响子类，而抽象类中可以不影响。

接口

- 接口（英文：Interface），在JAVA编程语言中是一个抽象类型，是抽象方法的集合，接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。
- 接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。
- 除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。
- 接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

接口与类相似点

- 一个接口可以有多个方法。
- 接口文件保存在 .java 结尾的文件中，文件名使用接口名。
- 接口的字节码文件保存在 .class 结尾的文件中。
- 接口相应的字节码文件必须在与包名称相匹配的目录结构中。

接口与类的区别

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了 static 和 final 变量。
- 接口不是被类继承了，而是要被类实现。
- 接口支持多继承。

接口特性

- 接口中每一个方法也是隐式抽象的,接口中的方法会被隐式的指定为 **public abstract**（只能是 public abstract，其他修饰符都会报错）。
- 接口中可以含有变量，但是接口中的变量会被隐式的指定为 **public static final** 变量（并且只能是 public，用 private 修饰会报编译错误）。
- 接口中的方法是不能在接口中实现的，只能由实现接口的类来实现接口中的方法。

抽象类和接口的区别

1. 抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行。
2. 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 **public static final** 类型的。
3. 接口中不能含有静态代码块以及静态方法(用 static 修饰的方法)，而抽象类是可以有静态代码块和静态方法。
4. 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

接口的声明

- **接口的声明**
- 接口的声明语法格式如下：

```
[可见度] interface 接口名称 [extends 其他的类名]
{
// 声明变量
// 抽象方法
}
```

简单例子

interface关键字用来声明一个接口。下面是接口声明的一个简单例子。

```
public interface NameOfInterface
{
 //任何类型 final, static 字段
 //抽象方法
 }
```

接口有以下特性：
接口是隐式抽象的，当声明一个接口的时候，不必使用abstract关键字。
接口中每一个方法也是隐式抽象的，声明时同样不需要abstract关键字。
接口中的方法都是公有的。

实例

Animal.java 文件代码：

```
interface Animal
{
public void eat();
public void travel();
}
```

接口的实现

- 当类实现接口的时候，类要实现接口中所有的方法。否则，类必须声明为抽象的类。
- 类使用implements关键字实现接口。在类声明中，Implements关键字放在class声明后面。

实现一个接口的语法，可以使用下面格式：

...implements 接口名称[, 其他接口名称, 其他接口名称..., ...] ...

实例

```
public class MammalInt implements Animal{
   public void eat(){
      System.out.println("Mammal eats");}
   public void travel(){
    System.out.println("Mammal travels");}
   public int noOfLegs(){
      return 0;}
   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();   }}
```

以上实例编译运行结果如下:

Mammal eats
Mammal travels

重写接口中声明的方法时，需要注意以下规则

类在实现接口的方法时，不能抛出强制性异常，只能在接口中，或者继承接口的抽象类中抛出该强制性异常。

类在重写方法时要保持一致的方法名，并且应该保持相同或者相兼容的返回值类型。

如果实现接口的类是抽象类，那么就没必要实现该接口的方法。

在实现接口的时候，也要注意一些规则：

一个类可以同时实现多个接口。

一个类只能继承一个类，但是能实现多个接口。

一个接口能继承另一个接口，这和类之间的继承比较相似。

接口的继承

一个接口能继承另一个接口，和类之间的继承方式比较相似。接口的继承使用extends关键字，子接口继承父接口的方法。

下面的Sports接口被Hockey和Football接口继承：

// 文件名: Sports.java

```
public interface Sports
{  public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}
```

// 文件名: Football.java

```
public interface Football extends Sports
{  public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}
```

// 文件名: Hockey.java

```
public interface Hockey extends Sports
{  public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

Hockey接口自己声明了四个方法，从Sports接口继承了两个方法，这样，实现Hockey接口的类需要实现六个方法。

相似的，实现Football接口的类需要实现五个方法，其中两个来自于Sports接口。

接口的多继承

在Java中，类的多继承是不合法，但接口允许多继承，。

在接口的多继承中extends关键字只需要使用一次，在其后跟着继承接口。 如下所示：

public interface Hockey extends Sports, Event

以上的程序片段是合法定义的子接口，与类不同的是，接口允许多继承，而 Sports及 Event 可能定义或是继承相同的方法

标记接口

最常用的继承接口是没有包含任何方法的接口。

标记接口是没有任何方法和属性的接口.它仅仅表明它的类属于一个特定的类型,供其他代码来测试允许做一些事情。

标记接口作用：简单形象的说就是给某个对象打个标（盖个戳），使对象拥有某个或某些特权。

例如：java.awt.event 包中的 MouseListener 接口继承的 java.util.EventListener 接口定义如下：

```
package java.util;
public interface EventListener
{}
```

标记接口

没有任何方法的接口被称为标记接口。标记接口主要用于以下两种目的：

建立一个公共的父接口：

正如EventListener接口，这是由几十个其他接口扩展的Java API，你可以使用一个标记接口来建立一组接口的父接口。例如：当一个接口继承了EventListener接口，Java虚拟机(JVM)就知道该接口将要被用于一个事件的代理方案。

向一个类添加数据类型：

这种情况是标记接口最初的目的，实现标记接口的类不需要定义任何接口方法(因为标记接口根本就没有方法)，但是该类通过多态性变成一个接口类型。

6.2 抽象类和抽象方法

6.2.1 抽象类及抽象方法的定义
6.2.2 为什么设计抽象类
6.2.3 开闭原则

6.2.1 抽象类及抽象方法的定义

- 抽象类：至少包含一个抽象方法的类。
- 抽象方法：没有实现的方法，由abstract修饰。

它的实现交给子类根据自己的情况去实现。

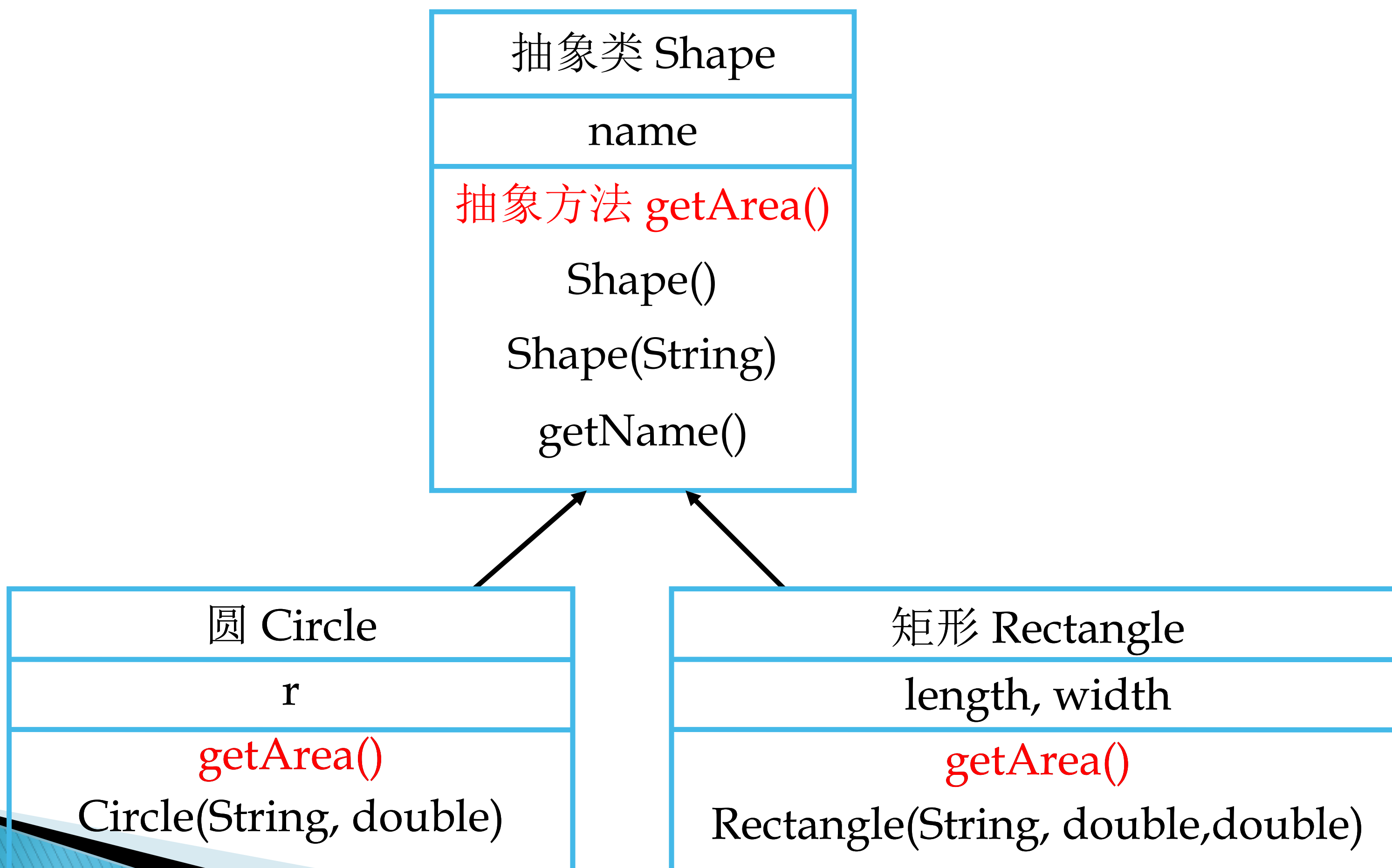
```
public abstract class Animal {
        private String name;

        public abstract void move();   //抽象方法

        public Animal() {   //构造方法，抽象类中可以有构造方法
        }
        public String getName(){  //非抽象方法，抽象类中可以有非抽象方法
                return this.name;
        }
}
```

6.2.1 抽象类及抽象方法的定义

【练习】完成如下代码的设计。



6.2.1 抽象类及抽象方法的定义

向上转型

```
public class Test {
   public static void main(String[] args) {
      Shape shape;
      shape = new Circle("circle", 5);
      System.out.println(shape.getName()+":"+shape. getArea());
      shape = new Rectangle("rect",10,8);
      System.out.println(shape.getName()+":"+shape. getArea());
   }
}
```

多态

6.2.2 为什么设计抽象类

- 我们可以构造出一组行为的抽象描述，但是这组行为却能够有任意个可能的具体实现方式。这个抽象描述就是抽象类，而这一组任意个可能的具体实现则由所有可能的派生类表现。

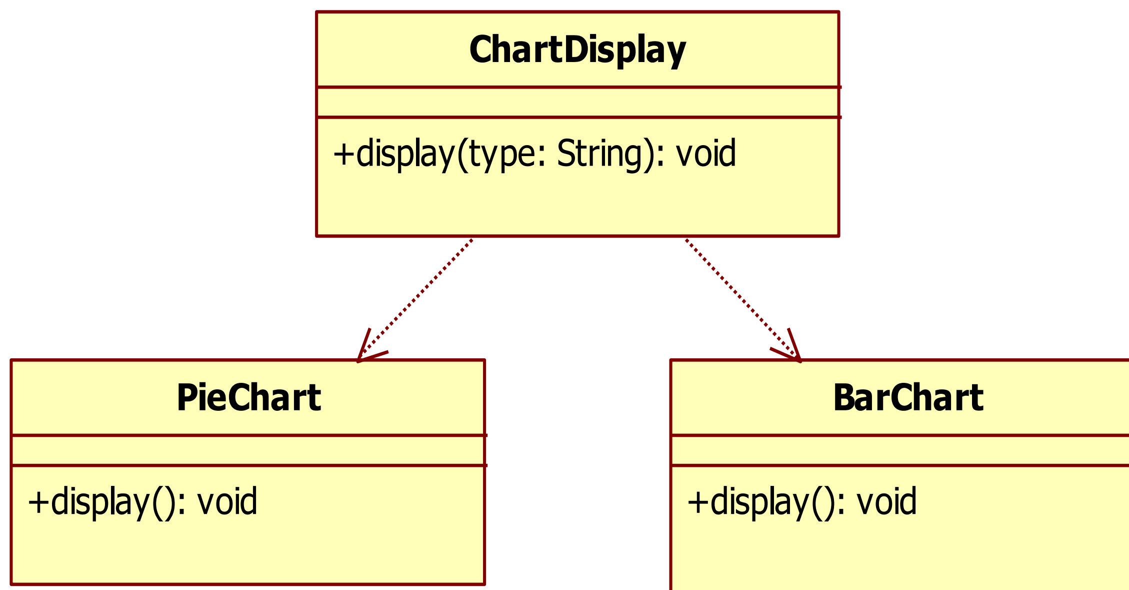
6.2.3 开闭原则

- 开闭原则OCP（Open-Closed Principle）
  - 面向对象设计的一个最核心的原则
  - 对于扩展是开放的，对于修改是关闭的

一个程序员累昏倒了，在医院昏迷了好几天，家人哭的稀里哗啦的，老婆孩子在旁边怎么叫就是不醒。
一天他同事来看他，第一句话就是对着躺在病床上的他说：需求又变了。
奇迹发生了，那个程序员一下从病床上做起来了。

6.2.3 开闭原则

【例6-2】为某个系统设计方案，要求能显示各种类型的图表，如饼图和柱状图等。



6.2.3 开闭原则

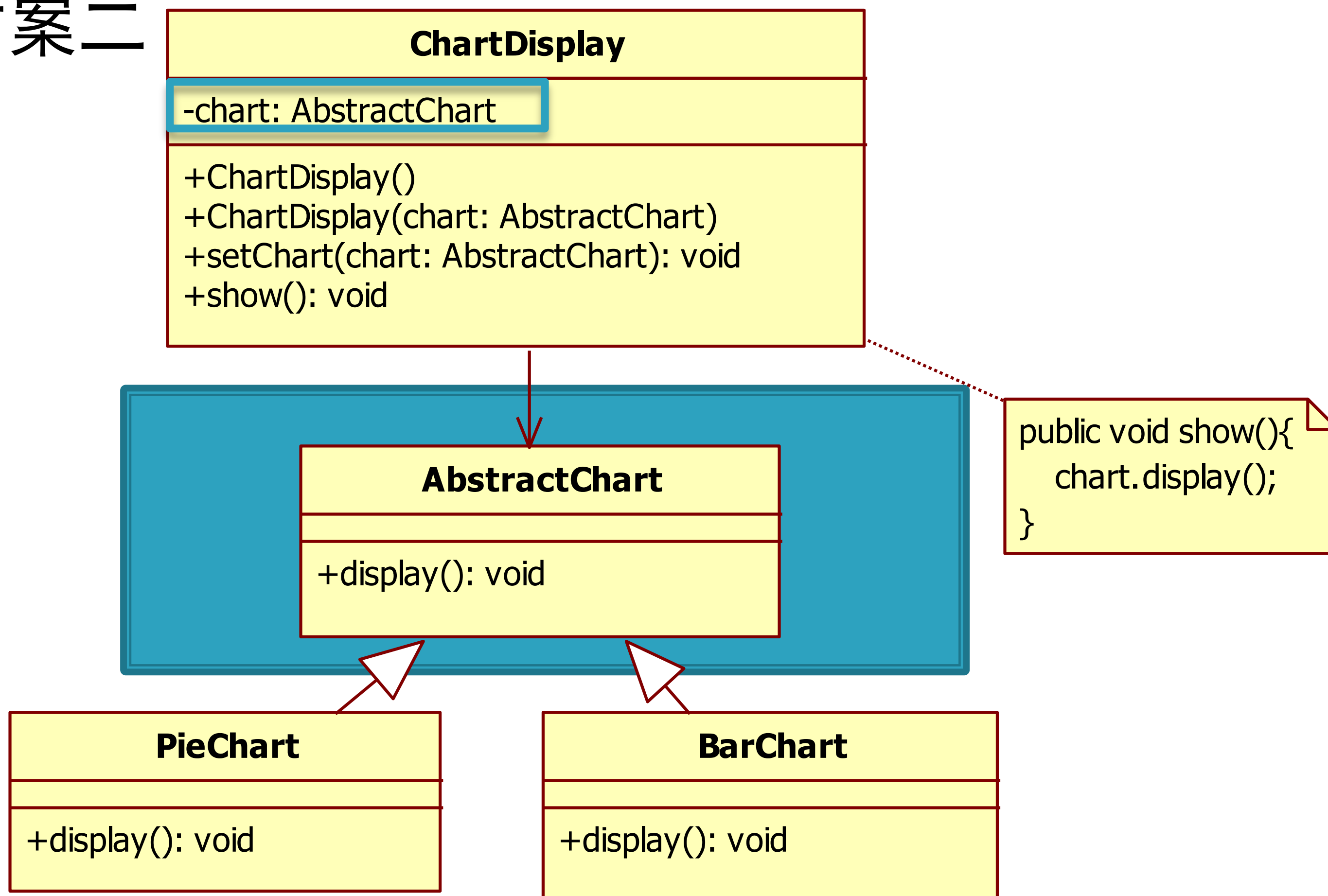
- 新的需求：增加显示一种新的图表--折线图。
- 方案一在设计好折线图类LineChart后，需要修改ChartDisplay类的display()方法的源代码，增加新

违反了开闭原则，没有实现对修改是关闭的

```
        chart.display();
}else if(type.equalsIgnoreCase("bar")){
        BarChart chart = new BarChart();
        chart.display();
} else if(type.equalsIgnoreCase("line")){
        LineChart chart = new LineChart();
        chart.display();
}
```

6.2.3 开闭原则

- 方案二



6.3 接口

6.3.1 接口的定义和实现
6.3.2 接口与抽象类的区别

6.3.1 接口的定义和实现

- **接口**
  - 接口由常量和一组抽象方法组成。
  - 接口支持多重继承。
    - 一个类可以同时实现多个接口。
    - 一个接口可以同时继承自多个接口(不会产生二义性)。

6.3.1 接口的定义和实现

- 指出下面代码中错误的部分。

```
public  interface  Introduce {
    public String detail();

    public void introduction(){
      detail();
    }

    private void showMessage();

    void speak();
}
```

Java接口中不能有方法实现

Java接口中的方法必须是public

6.3.1 接口的定义和实现

- 定义接口的一般格式

```
[public] interface 接口名 [extends 父接口名列表]{

      [public] [final] [static] 类型  常量名=常量值;

      [public ] [abstract] 返回类型 方法名(参数列表) ;
}
```

```
public interface  ChineseEmployee {
    String nationality="Chinese";     //public static final
    double pay();       //abstract
}
```

6.3.1 接口的定义和实现

- 如果一个类实现一个接口，且实现接口中声明的所有方法时，那么这个类才是具体的类；否则它还是一个抽象的类。

父	子	关键字	关系
类	类	extends	单一
接口	类	implements	多重
接口	接口	extends	多重
类	接口	不存在	

6.3.2 接口与抽象类的区别

- 抽象类和接口是支持开闭原则中抽象层定义的两种机制
- 区别1（次要）：从语法层面上抽象类和接口的区别很明显，抽象类可以有非常量的数据成员，也可以有非抽象的方法，甚至它可以有构造方法（虽然抽象类不能创建实例，但是构造方法为其子类对象的创建做好准备）；而接口只能有静态、常量的数据成员，只能有抽象方法，不能有构造方法。抽象类支持单继承；接口支持多继承。

6.3.2 接口与抽象类的区别

- 区别2（次要）：从编程的角度看，抽象类中的非抽象方法可以定义对象的默认行为方式，而接口中的方法永远只有一个驱壳，没有行为方式。
- 区别3（主要）：面向对象的设计实际是看世界的一个过程，所以设计理念上的区别才是抽象类和接口的本质不同。我们应该在对问题领域的本质的理解，以及对设计意图的理解的基础上正确地选择它们。

6.3.2 接口与抽象类的区别

【例6-4】门和报警门的设计。

- 假设在问题领域中有一个关于门Door的抽象概念，该Door具有两个动作open和close。
- 使用抽象类作为中间层
- 或者使用接口作为中间层

```
public abstract class Door {
        public abstract void open();
        public abstract void close();
}
```

```
public interface Door {
        public  void open();
        public  void close();
}
```

6.3.2 接口与抽象类的区别

需求变化：

要求Door还要具有报警的功能，该如何设计类结构呢？

```
public abstract class Door {
        public abstract void open();
        public abstract void close();
        public abstract void alarm();
}
```

```
public interface Door {
        public  void open();
        public  void close();
        public void alarm();
}
```

在Door的定义中把Door概念本身固有的行为方法（open()和close()）和另外一个概念“报警器”的行为方法（alarm()）混在了一起，使那些仅仅依赖于Door这个概念的模块会因为“报警器”的改变（例如修改alarm()方法的参数）而改变。

6.4 面向接口编程

- 6.4.1 案例分析
- 6.4.2面向接口编程的代码组织

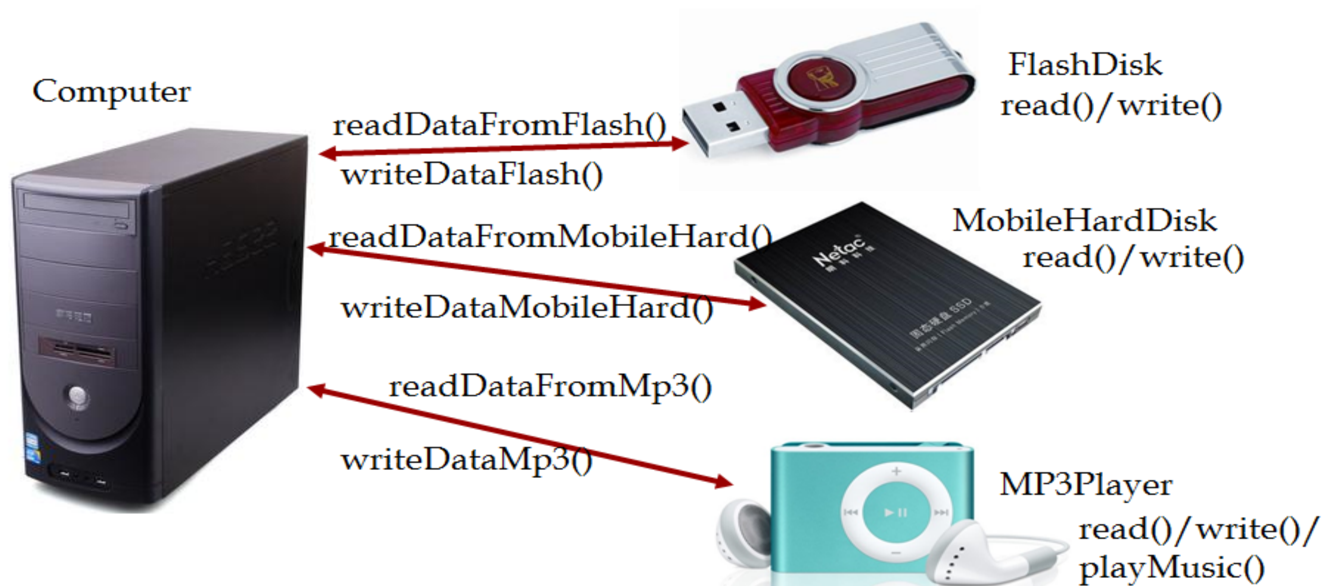
6.4.1 案例分析

【例6-5】现要开发一个应用，模拟移动存储设备的读写，即模拟计算机与U盘、移动硬盘、MP3等设备间的数据交换。

- 现已确定有U盘、移动硬盘、MP3播放器三种设备，但以后可能会有新的移动存储设备出现，所以数据交换必须有扩展性，保证计算机能与目前未知、而以后可能会出现的存储设备进行数据交换。

6.4.1 案例分析

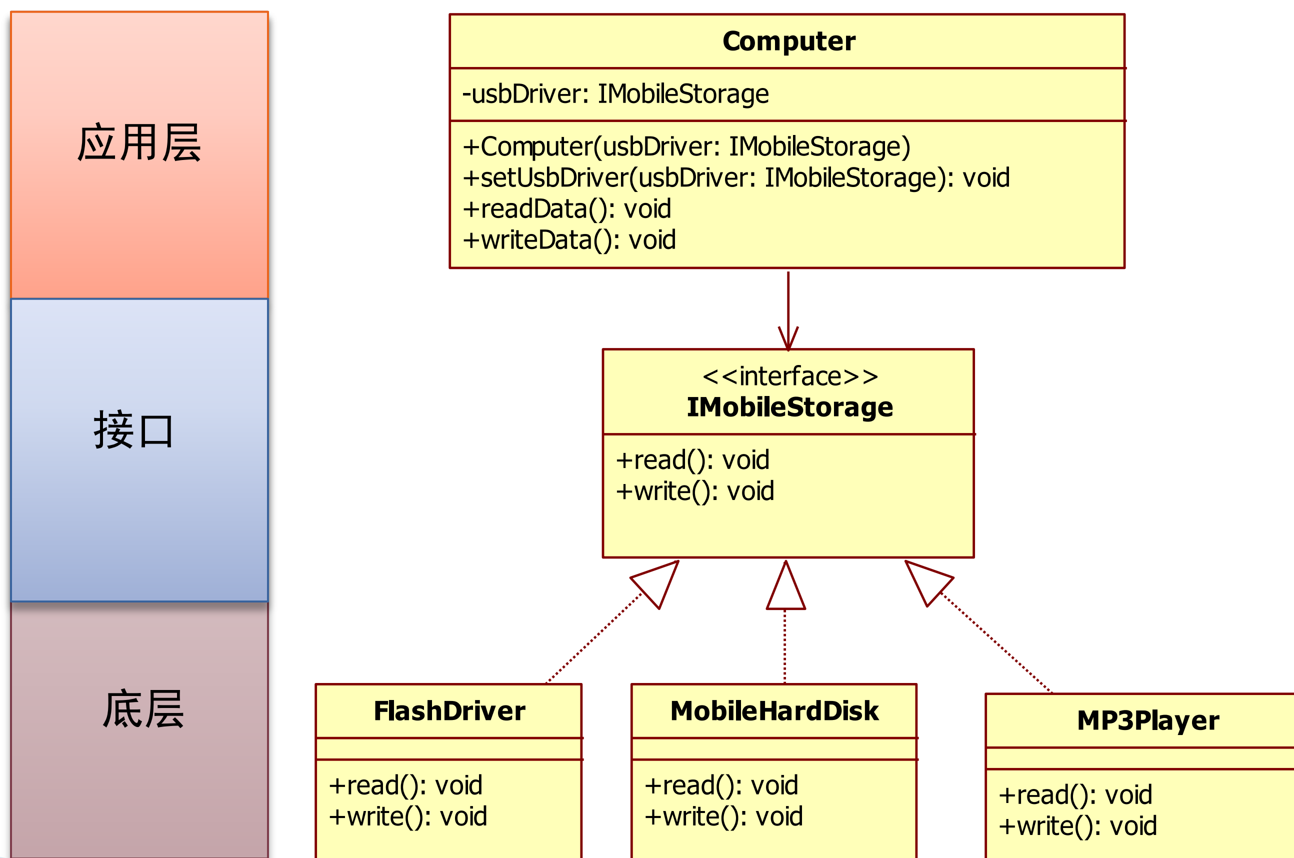
- **方案一：**分别定义U盘FlashDisk类、移动硬盘MobileHardDisk类、MP3播放器MP3Player类，实现各自的read()和write()方法。然后在Computer类中实例化上述三个类，为每个类分别定义读、写方法。



6.4.1 案例分析

接口IMobileStorage

6.4.2 面向接口编程的代码组织



6.4.2 面向接口编程的代码组织

- 包结构

storage
- impl
  - FlashDisk.java
  - MobileHardDisk.java
  - MP3Player.java
- Computer.java
- IMobileStorage.java
- Test.java

2 impl（FlashDisk.java、MobileHardDisk.java、MP3Player.java）
3 Computer.java
1 IMobileStorage.java
4 Test.java

第二步，编写底层各实现类，实现接口中的方法，直接操作底层数据（接口的实现类通常放在接口的子包中，命名为“impl”）。

第三步，编写应用层Computer类，依赖接口完成业务逻辑，屏蔽底层的实现。
（1）将接口引用变量作为应用层的数据成员。
（2）定义构造方法或set方法对接口数据成员初始化。
（3）封装应用层的业务行为方法，通过接口成员调用接口中的方法实现业务逻辑。

第一步，编写中间层接口IMobileStorage，定义方法，即系统的行为模型。

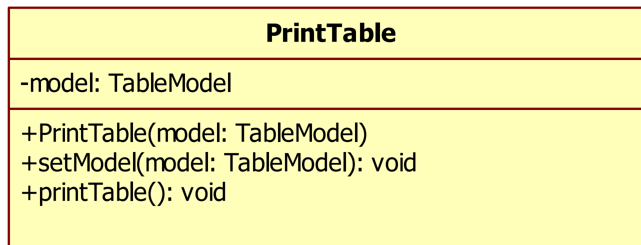
第四步，编写测试类Test，在main()方法中创建接口的实现类对象，传递给应用层实例，应用层实例调用应用层业务方法完成任务。

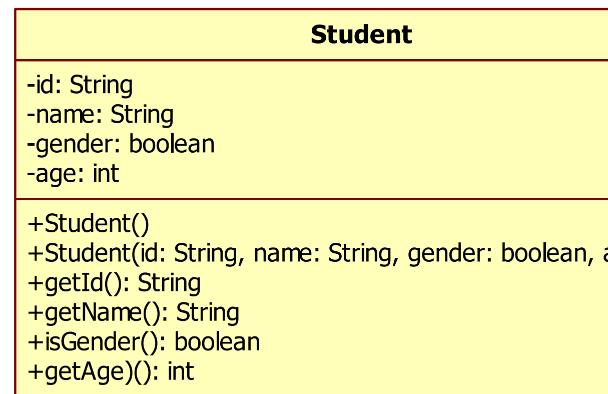
6.5 综合实践--格式化输出学生对象数据

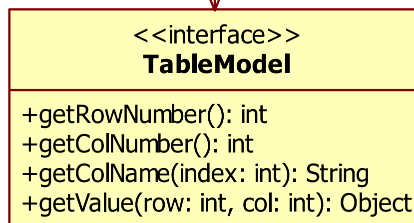
- 将学生（Student类）数据按照表格的方式输出。学生数据存储在不同的结构中，一种是二维字符串数组，每一行代表一个学生的数据；另一种是一维Student类型数组，每个元素代表一个学生。要求采用面向接口的方式设计架构，在接口层抽象出输出一个二维表格所需的所有方法，并在底层用两种存储方式分别予以实现。

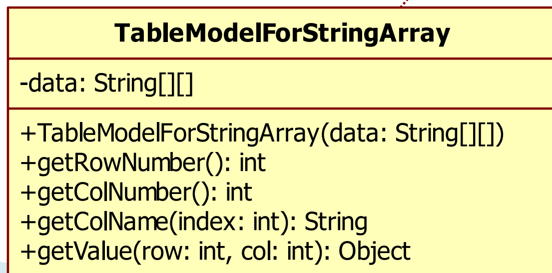
```
ID        NAME      GENDER   AGE
----------------------------
1001      zhangs    男        21
1002      lis       男        23
1003      wangwu    女        21
1004      zhangs    男        24
1005      zhaol     女        25
1006      qingqi    男        21
```

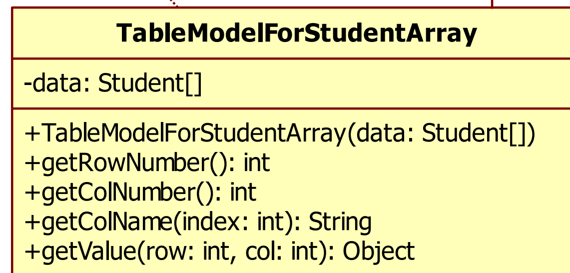
6.5.1 系统架构











6.5.2 向接口编程的代码

1．接口层TableModel

```
public interface TableModel {
        public int getRowNumber();        //获取表格的行数
        public int getColNumber();        //获取表格的列数
        public String getColName(int index);        //获取表头名称
        public Object getValue(int row,int col);      //获取 row行col
列的数据
}
```

getValue()方法获取row行col列的表格数据，因为表格数据的类型并不统一，所以用最高类型Object作为返回值类型，允许该方法返回任何类型的数据。

6.5.2 向接口编程的代码

2．底层实现类TableModelForStringArray

```
String[][] str={
        {"ID","NAME","GENDER","AGE"},
        {"1001","zhangs","男","21"},
        {"1002","lis","男","23"},
        {"1003","wangwu","女","21"},
        {"1004","zhangs","男","24"},
        {"1005","zhaol","女","25"},
        {"1006","qingqi","男","21"}
};
```

6.5.2 向接口编程的代码

2．应用层PrintTable类

```
public class PrintTable {
		private TableModel model;		//接口成员

		public PrintTable(){
		}
		public PrintTable(TableModel model) {	//构造方法初始化接口成员变量
				this.model = model;
		}
		public void setModel(TableModel model) {		//set方法初始化接口成员变量
				this.model = model;
		}
```

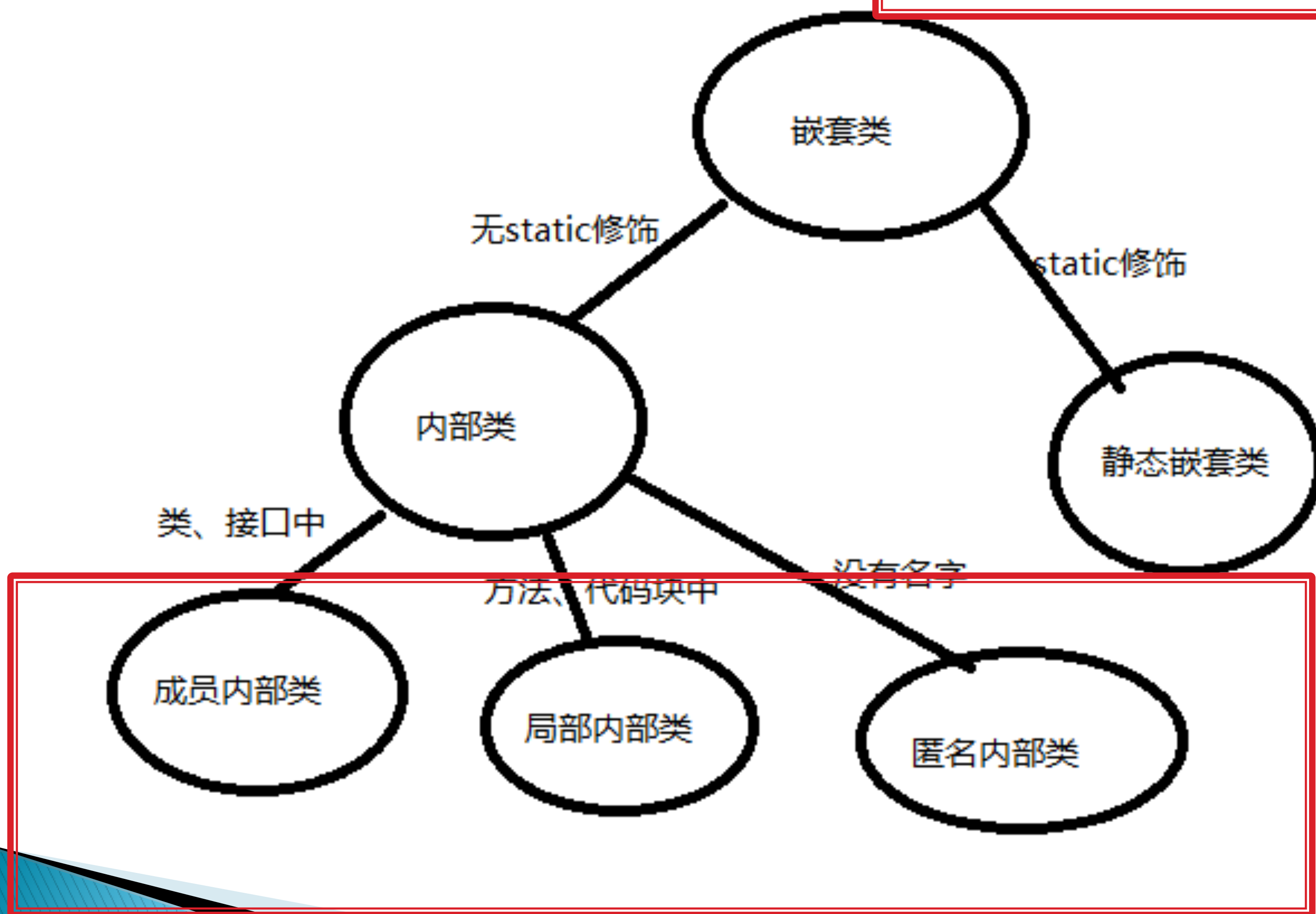
6.5.2 向接口编程的代码

3．底层实现类TableModelForStudentArray

```
Student[] s={
        new Student(1001,"zhangs",true,21),
        new Student(1002,"lisi",true,24),
        new Student(1003,"wangw",false,23),
        new Student(1004,"zhaol",true,25),
        new Student(1005,"qianqi",false,20),
        new Student(1006,"liuba",true,22),
};
```

内部类（了解）

为什么使用嵌套类？



内部类（了解）

- 当一个类的定义放在另一个类的实体时，则该类叫做内部类，该类所在的类叫做外部类。
- 在一个类体中可以出现内容：成员变量、成员方法、构造方法、静态语句块、静态变量、方法、内部类

内部类（了解）

- 语法格式

```
class 外部类类名{
        class 内部类类名{
            内部类类体
        }
    }
```

内部类（了解）

- 成员内部类

　　定义：成员内部类是定义在另一个类或接口中的内部类

Outer.java

```
public class Outer {
private String str1="Outer类的str1";
private String str2="Outer类的str2";
public class Inner{//内部类
    private String str1="Inner类的str1";
    private String str2="Inner类的str2";
    public void show(){
        System.out.println(str1);
        System.out.println(str2);  }   }    }
```

```
import aa.Outer.Inner;
public class TestOuter {
public static void main(String[] args){
//如果要创建内部类，那么在此之前需要先建外部类对象

//创建外部类对象
 Outer o =new Outer();
//创建内部类
 Inner inner = o.new Inner();
 inner.show();}   }
```

```
Inner类的str1
Inner类的str2
```

Outer.java

```
public class Outer {
private String str1="Outer类的str1";
private String str2="Outer类的str2";
public class Inner{//内部类
    private String str1="Inner类的str1";
    private String str2="Inner类的str2";
    public void show(){
      System.out.println(str1);
      System.out.println(str2);  }   }    }
```

内部类的成员优先于外部类成员

使用“外部类名.this.成员”访问外部类成员

Outer.java

内部类的成员优先于外部类成员

```
public class Outer {
private String str1="Outer类的str1";
private String str2="Outer类的str2";
public class Inner{//内部类
    private String str1="Inner类的str1";
    private String str2="Inner类的str2";
    public void show(){
      System.out.println(str1);
      System.out.println(str2);
      System.out.println(Outer.this.str2);}   }   }
```

```
Inner类的str1
Inner类的str2
Outer类的str2
```

使用“外部类名.this.成员”访问外部类成员

内部类（了解）

- 成员内部类

  定义：成员内部类是定义在另一个类或接口中的内部类

注意事项：

1.内部类名称不能于外部类重名。

2.可以使用final访问修饰符（注意类的修饰符）。

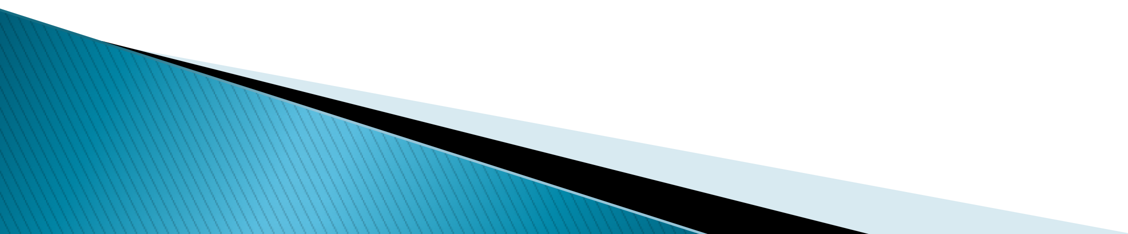
3.如果要创建内部类，那么在此之前需要先建外部类对象。

4.不能含有静态变量，静态代码块、静态方法（除了静态常量）。

5.外部类可以通过成员内部类的对象调用内部类私有成员。

6.成员内部类是一个独立的类，编译成独立的class文件。

内部类（了解）



面向对象思想总结

OO基本特征	定义	具体实现方式	优势
封装	隐藏实现细节，对外提供公共的访问接口	属性私有化、添加公有的setter、getter方法	增强代码的可维护性
继承	从一个已有的类派生出新的类，子类具有父类的一般特性，以及自身特殊的特性	继承需要符合的关系：is-a	1、实现抽象（抽出像的部分） 2、增强代码的可复用性
多态	同一个实现接口，使用不同的实例而执行不同操作	通过Java接口/继承来定义统一的实现接口；通过方法重写为不同的实现类/子类定义不同的操作	增强代码的可扩展性、可维护性

本章小结

- 利用多态性面向接口(抽象类)编程
  - 定义类继承自抽象类，并覆盖抽象方法；或者实现接口，实现接口中的方法。
  - 将子类对象赋值给抽象父类引用或接口引用。
  - 利用父类的这些引用调用子类中的同名方法。
- 子类对象赋给父类引用后的3个层次

  (1) 父类中没有的方法子类对象不能调用。

  (2) 如果子类没有覆盖父类的方法则调用父类的方法。

  (3) 如果子类覆盖父类的方法则调用子类的方法。
- 父类对象转换为子类对象

本章小结

- 抽象类
  - 抽象类
  - 抽象方法
- 接口
  - 特殊的抽象类，由常量和抽象方法组成。
  - 接口中的所有方法默认为公开抽象方法(public abstract)，在类中实现接口的方法时，方法必须是public修饰。
  - 接口中的所有属性默认为公开静态常量(public static final)。
- 接口与抽象类的区别

本章思维导图