第6章　多态性

北京理工大學 BEIJING INSTITUTE OF TECHNOLOGY

- 多态性
  - 继承层次结构中子类对象和父类对象间的关系
  - 抽象类和抽象方法的声明和使用
  - 接口的声明和实现
  - 实现多态性的编程技术

- 先来看一个例子，预测下程序运行的结果。

- 多态性(Polymorphism)
- **Polymorphism（多态）** 是面向对象程序设计语言中继数据抽象和继承之后第三种基本特性
  - 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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 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。

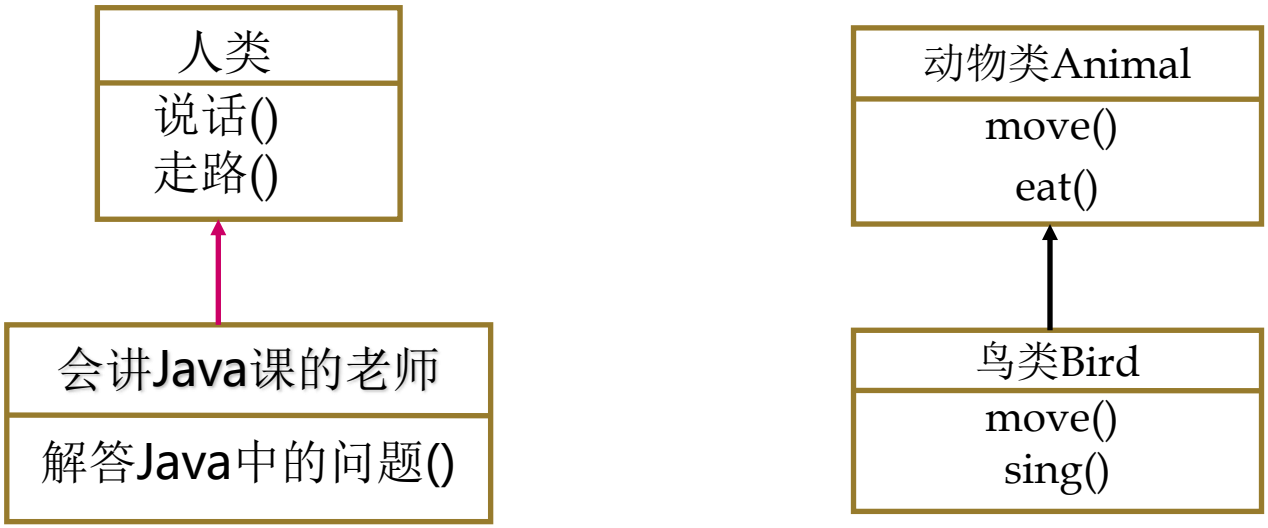
```
public class Test {
    public static void main(String[] args) {
        //子类对象送给父类引用
        Animal a = new Bird();
        ……
    }
}
```

- 向上转型：将子类对象直接赋值给父类引用的过程，不需要强制类型转换，由系统自动完成（已存在“is a”关系）

Java语言中Object是所有类的直接或间接父类，也就是说，

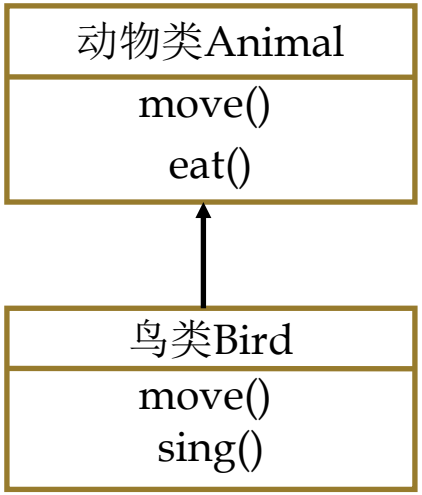
任何类型的对象都可以赋值给Object引用。

【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?

【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?

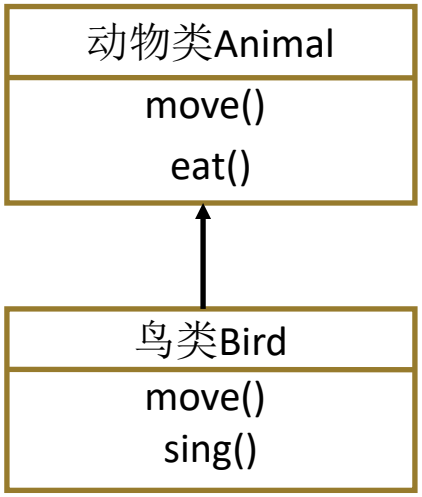
6.1.1 多态性

【画龙点睛 】在继承层次中，把子类对象赋给父类引用后：父类中没有的方法不能调用；子类没有重写的方法，执行父类方法行为；子类重写的方法，执行子类的方法行为。

6.1.1 多态性

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 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。

1 继承

2 向上转型

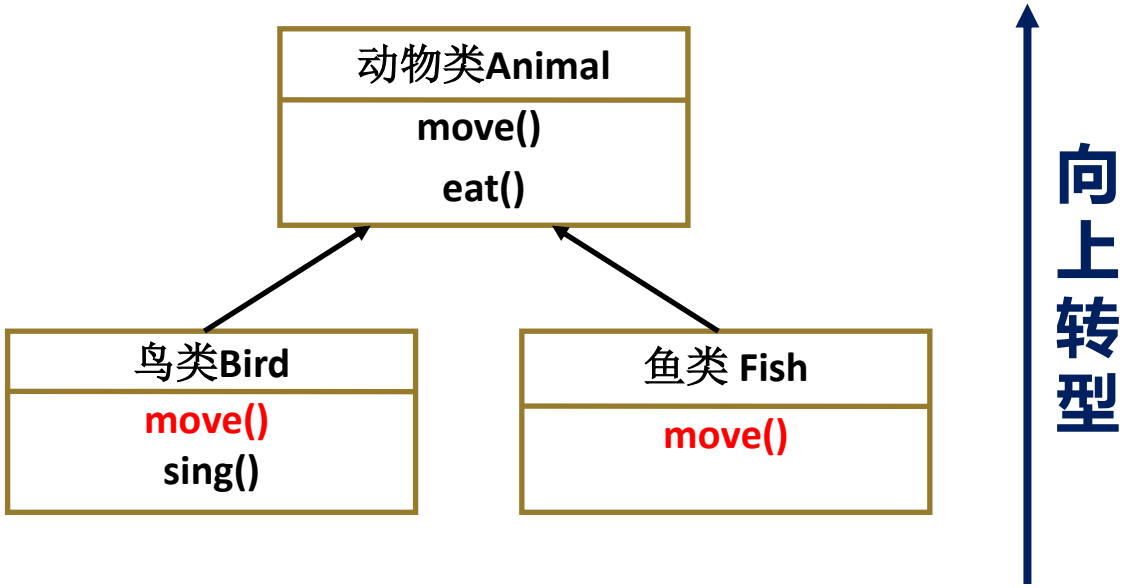
3 子类对父类中的方法进行重写

多态三要素

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

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



多态三要素

1 继承

2 向上转型

3 子类对父类中的方法进行重写

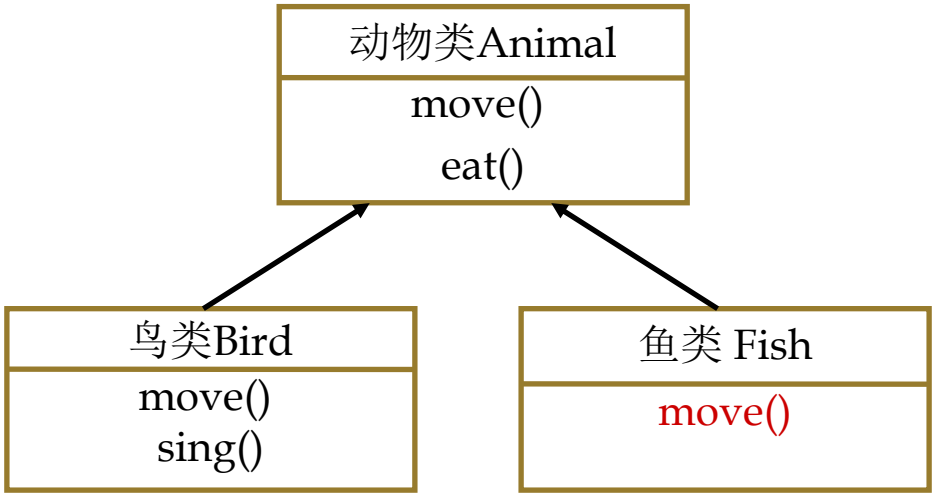
```
Animal[] a = new Animal[2];
a[0] = new Bird("bird",4);
a[1] = new Fish("fish",1);
for(int i=0; i<a.length; i++){
   a[i].move();
}
```

6.1.2 静态绑定和动态绑定

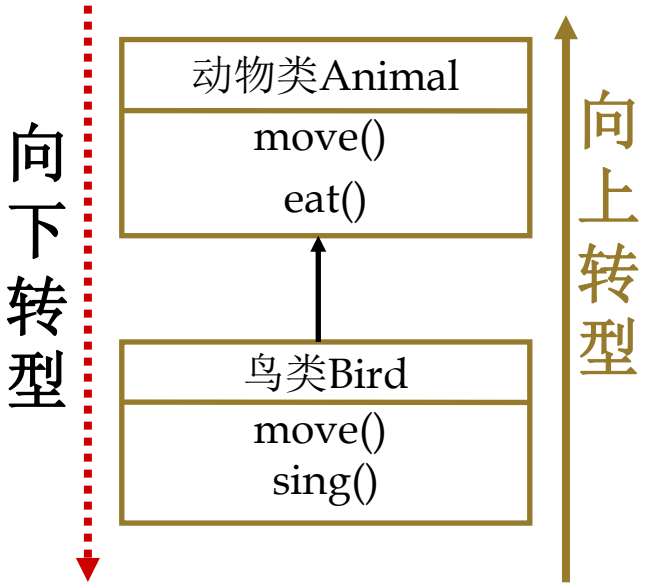
- 动态绑定技术： Java虚拟机实现，多态的基础
- 静态绑定技术：编译器

- JVM调用方法的过程
  1. 编译器查看对象的声明类型和方法名，找出所有可能被调用的候选方法。
  2. 编译器查看调用方法时提供的参数类型，如果存在一个参数类型和个数完全匹配的方法，则调用该方法---重载解析。
  3. 如果方法是private、static、final、构造方法，则编译器可以准确地获悉应该调用哪个方法并进行调用---静态绑定。否则，将进行动态绑定。
  4. 当程序运行、并采用动态绑定调用方法时，JVM一定调用与当前对象的实际类型最合适的那个类的方法。

- 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



- 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)



- 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。

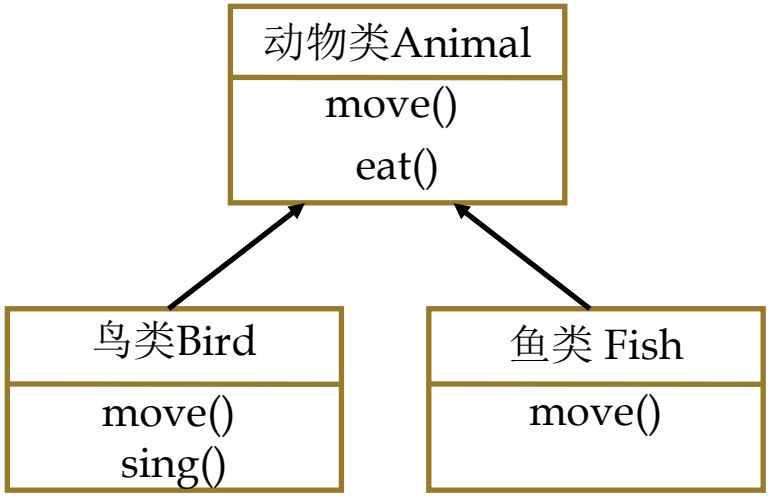
```
Animal  animal = new Animal();
Bird bird = (Bird)animal;
```

编译无错，运行出错， ClassCastException异常：JVM在检测到两个类型间转换不兼容时引发的运行时异常。

- 向下转型
  - 在做强制类型转换前，应使用instanceof运算判断引用变量的类型！

```
Animal   animal = new Bird();
if(animal instanceof Bird) {
      Bird bird=(Bird) xxx;
}
```

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) ?

【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。

向上转型和向下转型的应用

```
public class EqualsTest {
    public static void main(String[] args) {
        Employee e1=new Employee();
        e1.setId("001");
        e1.setName("zhang");
        Employee e2= e1;

        Employee e3 = new Employee ("001","zhang");
        System.out.println("e1:"+e1);
        System.out.println("e2:"+e2);

        System.out.println("e3:"+e3);
        System.out.println("e1==e3?"+e1.equals(e3));
    }
}
```

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

如何利用equals()方法比较对象的内容？

向上转型和向下转型的应用

- Emp

```
public bo
    retu
}
```

```
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));
    }
}
```

向上转型

```
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

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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- **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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 抽象方法的现实意义(用处)
  - 类在抽象层次上是比较高的，虽然有某些行为，但行为方式不能确定，或者有很多表现形式。
  - 例如：Animal类中的move() 方法，在Animal层次上move是一个抽象的行为，不能确定。如果定义Bird类，则move方法是具体的。

```
public class Bird extends Animal{
        public void move() {  //子类实现父类的抽象方法
                System.out.println("我可以在天空飞翔……");
        }
}
```

6.2.1 抽象类及抽象方法的定义

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 抽象方法
  - 抽象方法不提供实现。
  - 包含抽象方法的类必须声明为抽象类。
  - 抽象父类的所有具体子类都必须为父类的抽象方法提供具体实现；否则其仍然为抽象类。
  - 静态方法、私有方法、final方法、构造方法不能被声明为抽象方法。

6.2.1 抽象类及抽象方法的定义

- 抽象类中不一定所有方法都是抽象方法，可以在抽象类中定义非抽象方法。
- 尽管抽象类不能实例化，但是抽象类可以有构造方法，为其子类的创建而准备。

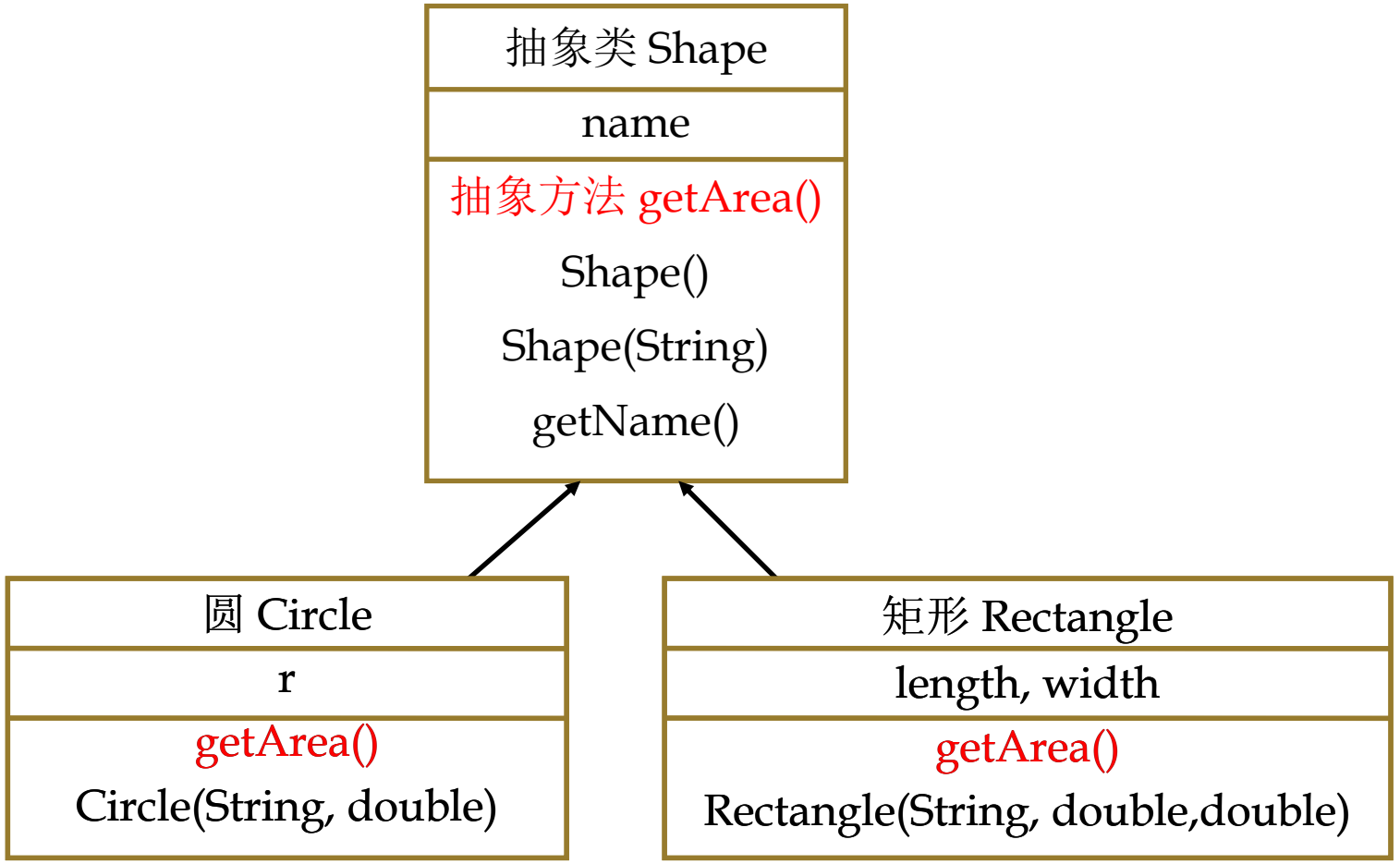
6.2.1 抽象类及抽象方法的定义

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 抽象类与多态
  - 抽象父类不能实例化，但可以使用抽象父类来声明引用变量，用以保存抽象类所派生的任何具体类的对象。程序通常使用这种变量来多态地操作子类对象。

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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

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

        public Shape() {
        }
        public Shape(String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }

        public abstract double getArea();   //抽象方法
}
```

6.2.1 抽象类及抽象方法的定义

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

```
public class Circle  extends  Shape{
    private double r;

    publi
        sup
        thi
    }

    publi
        r
    }
}
```

```
public class Rectangle  extends  Shape{
    private double length, width;

    public Rectangle(String name, double length, double width) {
        super(name);
```

```
public double getArea() { //实现抽象方法
    return Math.PI*r*r;
}
```

```
public double getArea() {  //实现抽象方法
    return length*width;
}
```

```
}
```

6.2.1 抽象类及抽象方法的定义

北京理工大學 BEIJING INSTITUTE OF TECHNOLOGY

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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 抽象类通常用来表征对问题领域进行分析、设计后得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象。
- 比如说，如果进行一个图形编辑软件的开发，就会发现问题领域存在着圆、三角形这样一些具体概念，它们是不同的，但是它们又都属于“形状”这样一个概念，形状这个概念在问题领域是不存在的，它就是一个抽象概念。正因为抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是不能够实例化的。

6.2.2 为什么设计抽象类

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 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

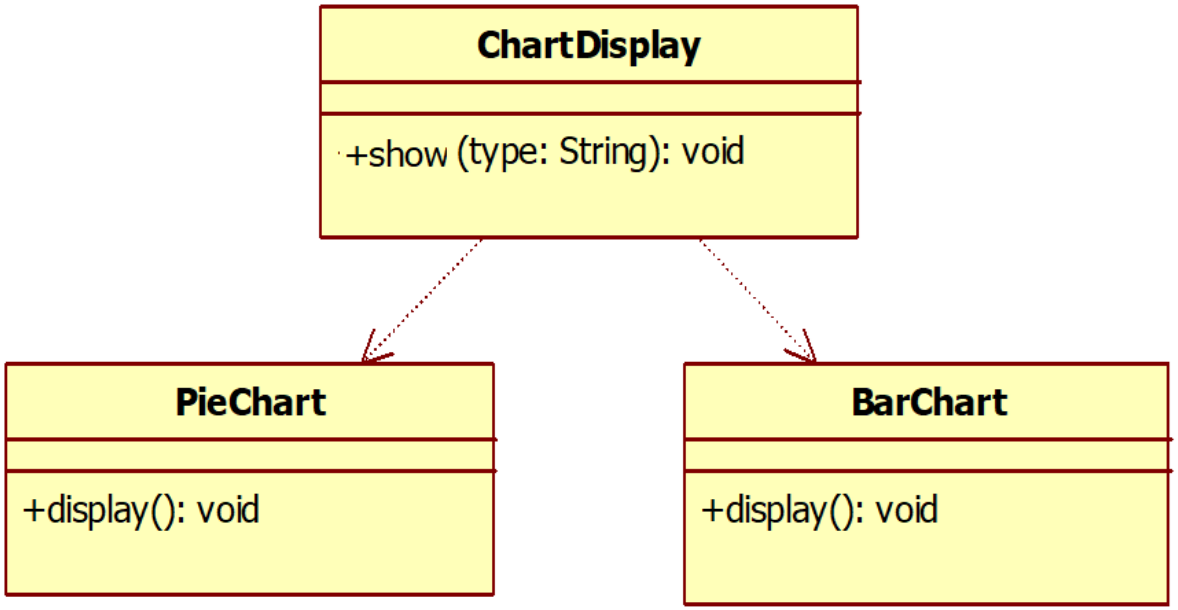
在面向对象编程领域中，开闭原则规定“软件中的对象（类，模块，函数等等）应该对于扩展是开放的，但是对于修改是封闭的”，这意味着一个实体是允许在不改变它的源代码的前提下变更它的行为。

6.2.3 开闭原则

- 任何软件产品只要在生命期内，都需要面临一个很重要的问题，即它们的需求会随时间的推移而发生变化。当软件系统面对新的需求时，应该尽量保证系统的设计框架是稳定的。
- 如果一个软件设计符合开闭原则，那么可以非常方便地对系统进行扩展，而且在扩展时无须修改现有代码，使得软件系统在拥有适应性和灵活性的同时具备较好的稳定性和延续性。
- 随着软件规模越来越大，软件寿命越来越长，软件维护成本越来越高，设计满足开闭原则的软件系统也变得越来越重要。

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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

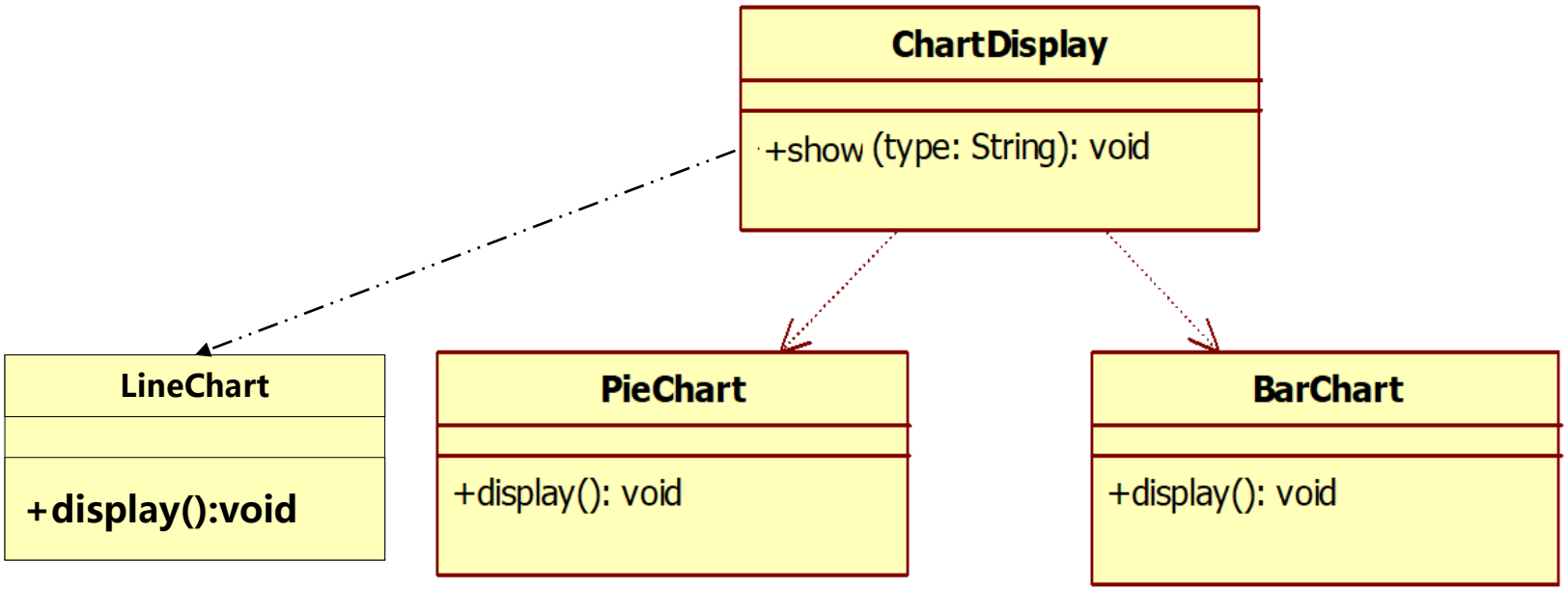
- 在ChartDisplay类的show()方法中会存在如下代码：

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

6.2.3 开闭原则

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 新的需求：增加显示一种新的图表--折线图。



6.2.3 开闭原则

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 方案一在设计好折线图类LineChart后，需要修改ChartDisplay类的show()方法的源代码，增加新的判断逻辑：

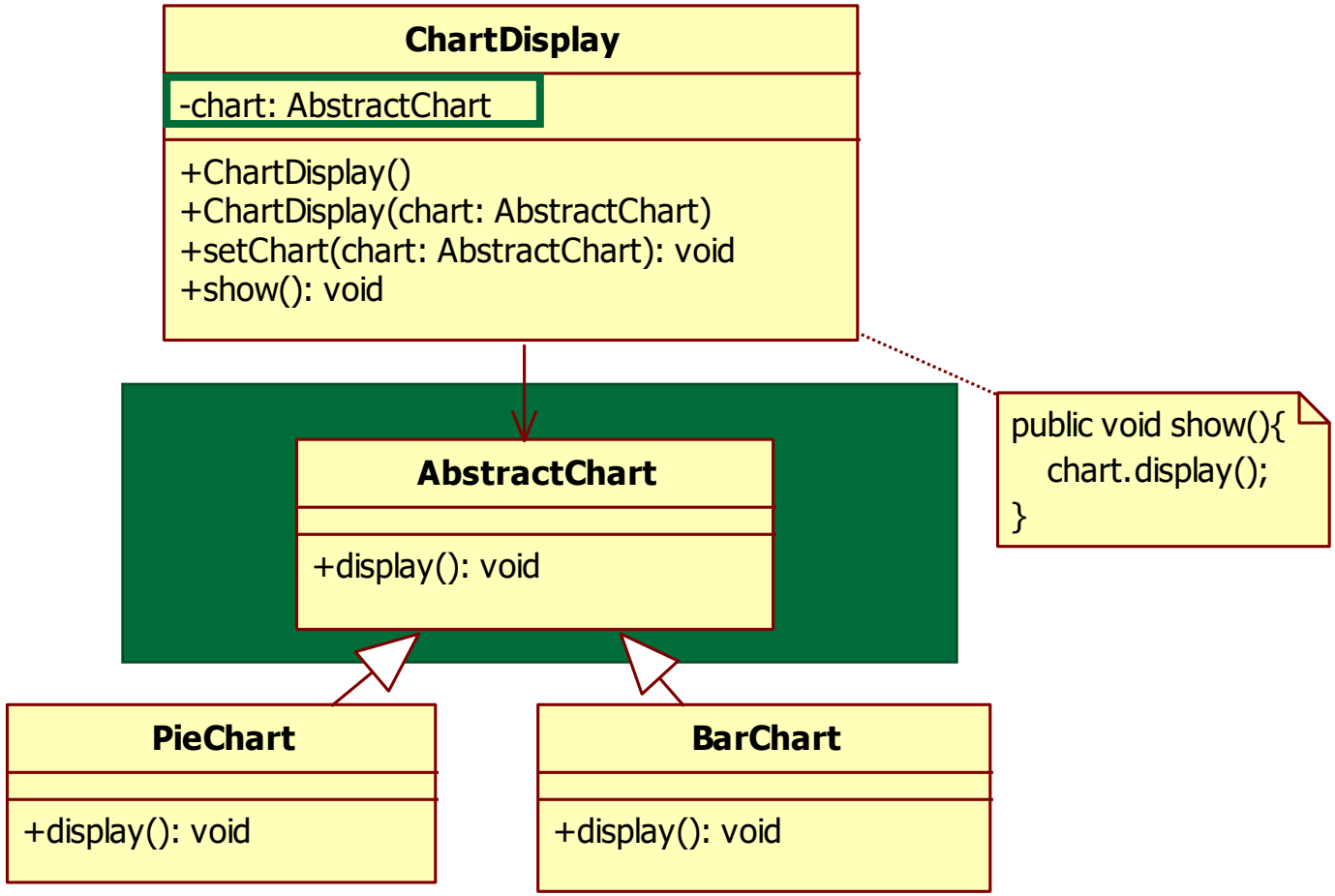
违反了开闭原则，没有实现对修改是关闭的

```
if(type.equalsIgnoreCase("pie")){
        PieChart chart = new PieChart();
        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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 方案二

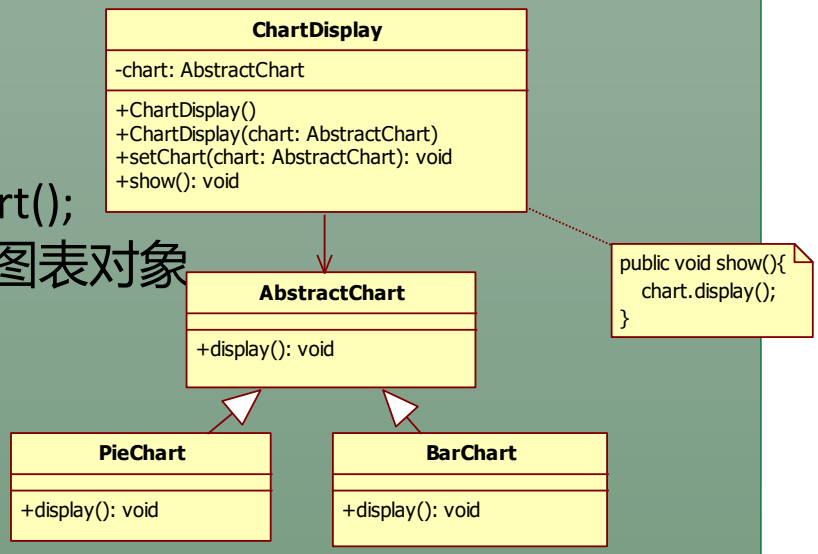


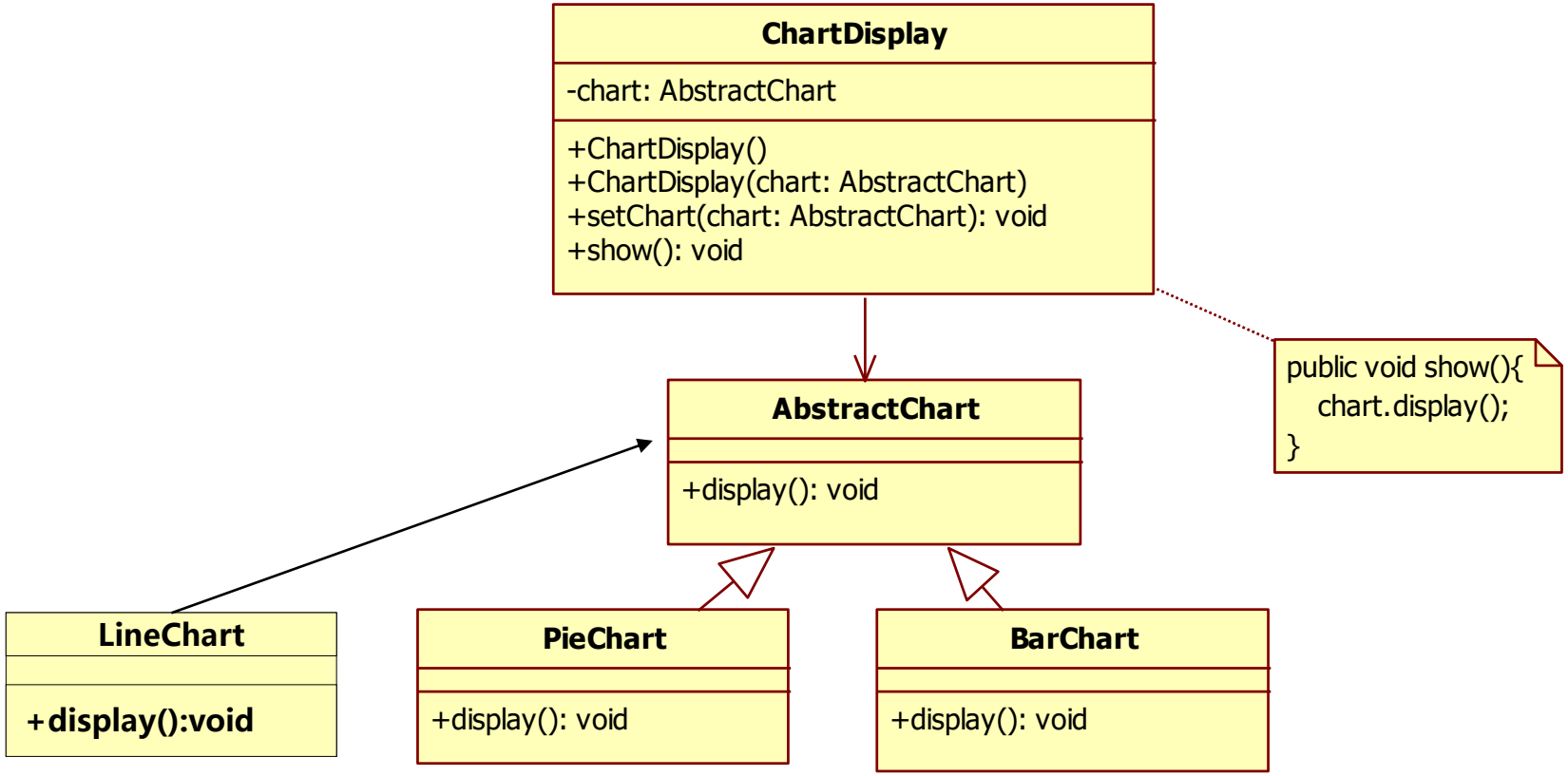
6.2.3 开闭原则

```
public class Client {
        public static void main(String[] args){
                ChartDisplay chartDisplay = new ChartDisplay();

                PieChart pie = new PieChart();
                //向ChartDisplay注入具体图表对象
                chartDisplay.setChart(pie);
                chartDisplay.show();

                BarChart bar = new BarChart();
                //向ChartDisplay注入具体图表对象
                chartDisplay.setChart(bar);
                chartDisplay.show();
        }
}
```




ChartDisplay
-chart: AbstractChart
+ChartDisplay()
+ChartDisplay(chart: AbstractChart)
+setChart(chart: AbstractChart): void
+show(): void
public void show(){
chart.display();
}
AbstractChart
+display(): void
LineChart
+display():void
PieChart
+display(): void
BarChart
+display(): void

6.2.3 开闭原则

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 方案二的特点
  - 在方案二的设计架构中引入了抽象类作为中间层，通过扩展它的子类使系统适应了需求的变化，保持了历史代码的不变，从而提高了系统的稳定性。
  - 如果需要增加一种新的图表，如折线图LineChart，只需要将LineChart也作为AbstractChart的子类，在客户端向ChartDisplay中注入一个LineChart对象即可，无需修改现有源代码，方案二符合了开闭原则。

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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- **接口**：特殊的抽象类，接口比抽象的概念更向前迈了一步，接口可以看成是“纯粹”的抽象类。
  - 关键字interface标识，由常量和一组抽象方法组成。
  - 接口中的所有方法都必须是公开的抽象方法。系统默认方法为public abstract。

```
void move();    //默认是public abstract
```

  - 接口中的所有属性都是公开静态常量。系统默认属性为public static final。

```
int a = 10;  //public static final int a=10; 必须赋值
```

  - 接口没有构造方法（抽象类有构造方法，即使不写，系统自动加上一个无参构造方法）。

6.3.1 接口的定义和实现

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- **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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 接口的实现
  - 类对接口予以实现。
  - 为了声明一个类要实现接口，在类的声明中要包括一条implements语句。一个类可以实现多个接口，因此可以在implements后面可列出类要实现的接口系列，这些接口以逗号分隔。

```
class 类名 extends 父类   implements  接口1, 接口2 {
      ……
     //实现接口中所有的方法
}
```

6.3.1 接口的定义和实现

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 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	

6.3.1 接口的定义和实现

【例6-3】为一个用户管理系统的前台管理设计抽象接口层，并写一个实现类。

- 用户管理系统的前台向用户提供注册、登录功能。
- 用户类User包含userName、password等数据成员。

6.3.1 接口的定义和实现

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 按照开闭原则的实现方式，此处利用接口为系统定义一个抽象层，描述用户管理系统中应具有的行为。比如，注册对应着向系统中添加用户的行为，登录对应着在系统用户中按照用户名和密码进行查询的行为。

```
public interface UserDao {
        public boolean addUser(User user);
        public User getUser(String userName,String password);

}
```

6.3.1 接口的定义和实现

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 用数组保存所有的用户信息，并用count成员记录当前用户的数量。

```
public class UserDaoForArray implements UserDao{
        private User[] data;
        private int count=0;

        public UserDaoForArray(){
                data=new User[10];
        }
```

6.3.1 接口的定义和实现

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

```
public boolean addUser(User user) {
        if(count==data.length){ //用户上限已达到
                return false;
        }
        //查找用户名是否已存在
        for(int i=0; i<count; i++){
                if(data[i].getUserName().equals(user.getUserName())){
                        return false;
                }
        }
        //添加新用户
        data[count]=user;
        count++;
        return true;
}
```

6.3.1 接口的定义和实现

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

```
        public User getUser(String userName, String password) {
                for(int i=0;i<count;i++){
                        if(data[i].getUserName().equals(userName)
&& data[i].getPassword().equals(password) ){
                                return new User(userName,
password);
                        }
                }
                return null;  //用户名，密码不匹配返回null表示失败
        }
}
```

6.3.2 接口与抽象类的区别

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 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。

6.3.2 接口与抽象类的区别

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 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。
- 抽象类在Java语言中体现的是继承关系，要想使继承关系合理，父类和子类之间必须存在“is a”的关系，即父类和子类在概念本质上应该是相同的。
- 接口则不然，它不要求接口的实现类与接口在概念本质上是一致的，实现类与接口间仅是一种契约关系，实现类按接口的规定兑现契约，是一种“like a”关系的体现。

6.3.2 接口与抽象类的区别

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

【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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

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。

6.3.2 接口与抽象类的区别

- 改进方案
  - 既然open、close和alarm是属于两个不同概念的行为方式，那么就应该把它们分别定义在代表这两个概念的抽象类或接口中。
  - 因为Java语言不支持多重继承，所以或者两个概念都使用接口定义，或者一个概念使用抽象类定义、另一个概念使用接口定义。

6.3.2 接口与抽象类的区别

- 问题领域： “带有报警功能的门”
  - AlarmDoor在概念本质上是（is a）Door，同时它有具有（like a）报警的功能。
  - Door应该使用抽象类方式定义，Alarm这个概念应该通过接口方式定义，AlarmDoor继承Door，并实现Alarm接口。

6.3.2 接口与抽象类的区别

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

```
public abstract class Door {
        public abstract void open();
        public abstract void close();
}
interface Alarm{
        void alarm();
}
class AlarmDoor extends Door implements Alarm{
        public void open(){
        }
        public void close(){
        }
        public void alarm(){
        }
}
```

6.3.2 接口与抽象类的区别

- （1）抽象层到底使用抽象类还是使用接口，本质上取决于我们对于问题领域的理解，抽象类表示的是“is a”关系，接口表示的是“like a”关系，这一点可以作为选择的依据。
- （2）使用抽象类主要是为了代码的复用，并能够保证父类和子类间的层次关系。
- （3）系统中的行为模型（描述具有什么功能）应该总是通过接口而不是抽象类定义（属于like a的关系），如例6-3中UserDao对前台行为的抽象就是使用的接口。通过接口定义行为能够更有效地分离行为与实现，为代码的维护和修改带来方便。

6.4 面向接口编程

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 6.4.1 案例分析
- 6.4.2 面向接口编程的代码组织

面向接口的编程

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

【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

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

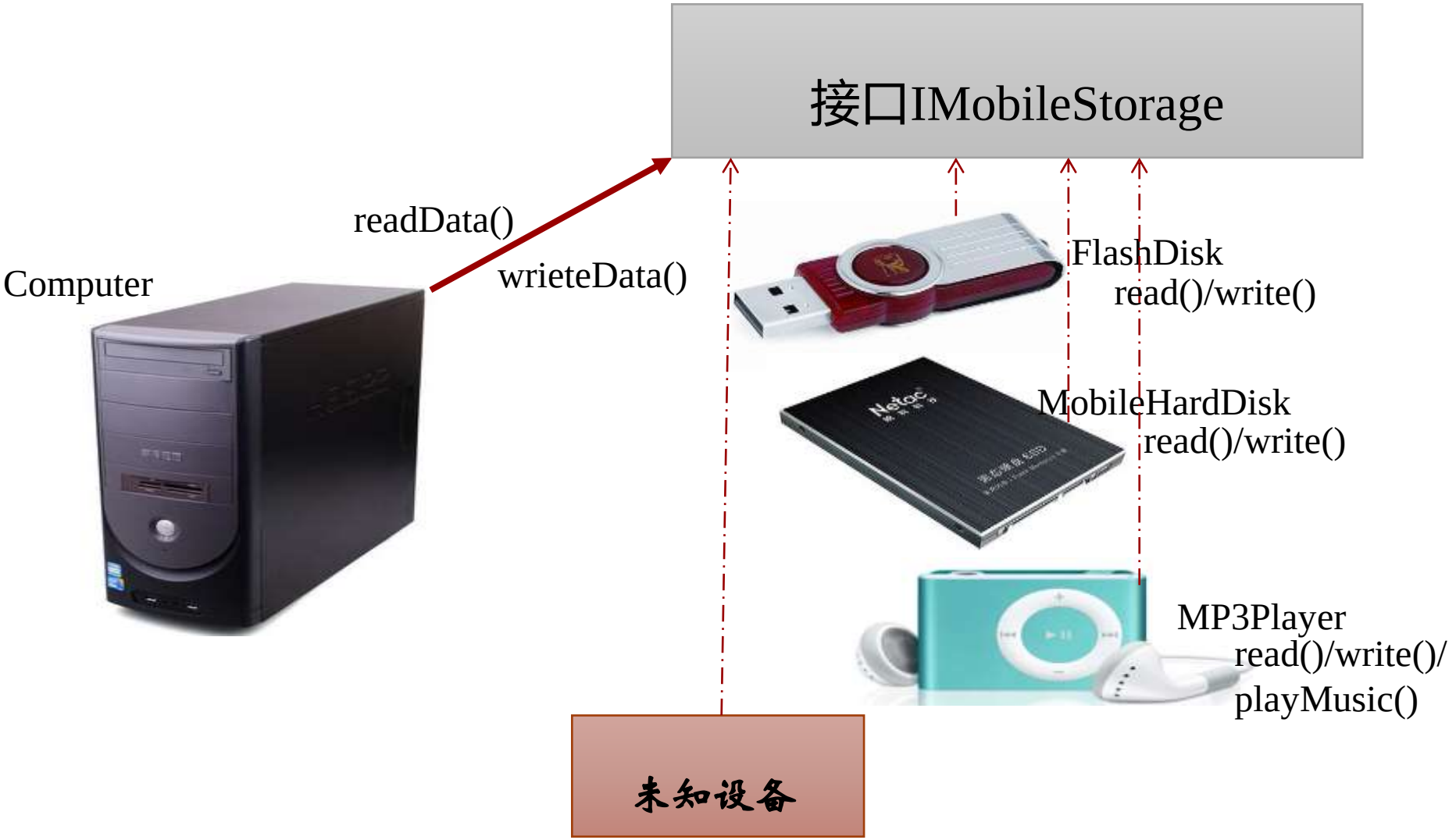
- 方案一缺陷
  - 可扩展性差，不符合“开闭原则”（为对扩展开放，对修改关闭）。
  - 未来有了新的移动存储设备时，必须对Computer的源代码进行修改。这就如在一个真实的计算机上，为每一种移动存储设备实现一个不同的插口、并分别有各自的驱动程序。当有了一种新的移动存储设备后，我们就要将计算机大卸八块，然后增加一个新的插口，再编写一套针对此新设备的驱动程序。这种设计显然不可取。

6.4.1 案例分析

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 方案二:
  - 采取满足开闭原则的设计方案，把Computer的行为read()和write()组织为一个中间的抽象层IMobileStorage，描述Computer对移动设备的读写。如前节的分析，系统中的行为模型应该总是通过接口定义，所以抽象层IMobileStorage用接口实现。三个存储设备分别实现该接口。
  - Computer类引入一个IMobileStorage类型的成员变量，并为其提供set方法，Computer中的readData()和writeData()方法通过该成员调用IMobileStorage实现类（三种存储设备）对象的读写方法。Computer类利用接口IMobileStorage实现了多态。

6.4.1 案例分析


接口IMobileStorage
readData()
wrieteData()
Computer
FlashDisk
read()/write()
MobileHardDisk
read()/write()
MP3Player
read()/write()/
playMusic()
未知设备

6.4.1 案例分析

北京理工大學 BEIJING INSTITUTE OF TECHNOLOGY

- 方案二的优势
- 具有良好的可扩展性。
- 有新的移动存储设备要接入Computer，只要令其实现IMobileStorage，就可以接进去运行。这就是所谓的“面向接口的编程”。

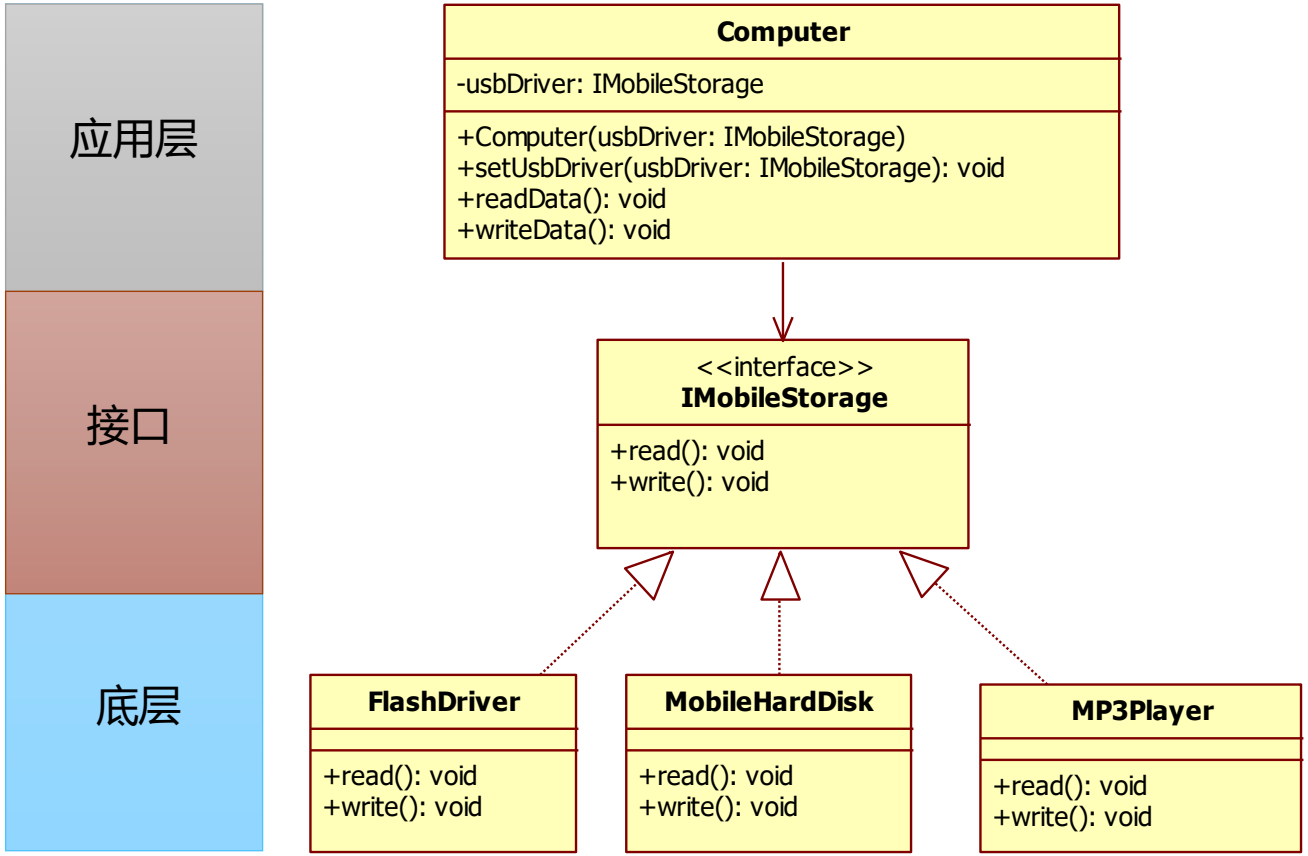
6.4.1 案例分析

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

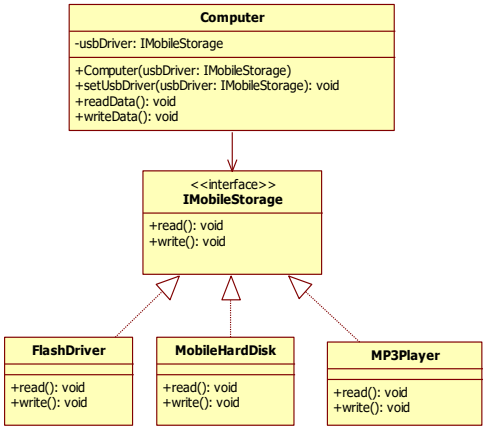
- 面向接口的编程
  - 对系统架构进行分层
  - 底层不是直接向顶层提供服务，即不是将自己直接实例化在顶层中
  - 定义一个中间的接口层，仅向顶层暴露接口层的功能
  - 顶层仅是接口依赖，而不依赖于底层具体类

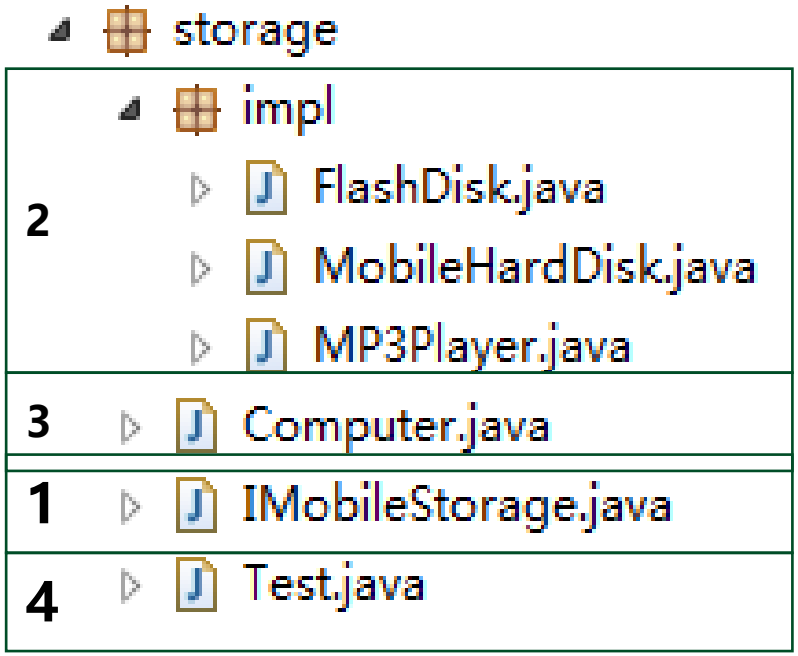
- 面向接口的编程增加了系统的稳定性和灵活性，当底层需要改变时，只要接口及接口功能不变，则顶层不用做任何修改，符合开闭原则。

6.4.2 面向接口编程的代码组织



- 包结构





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。

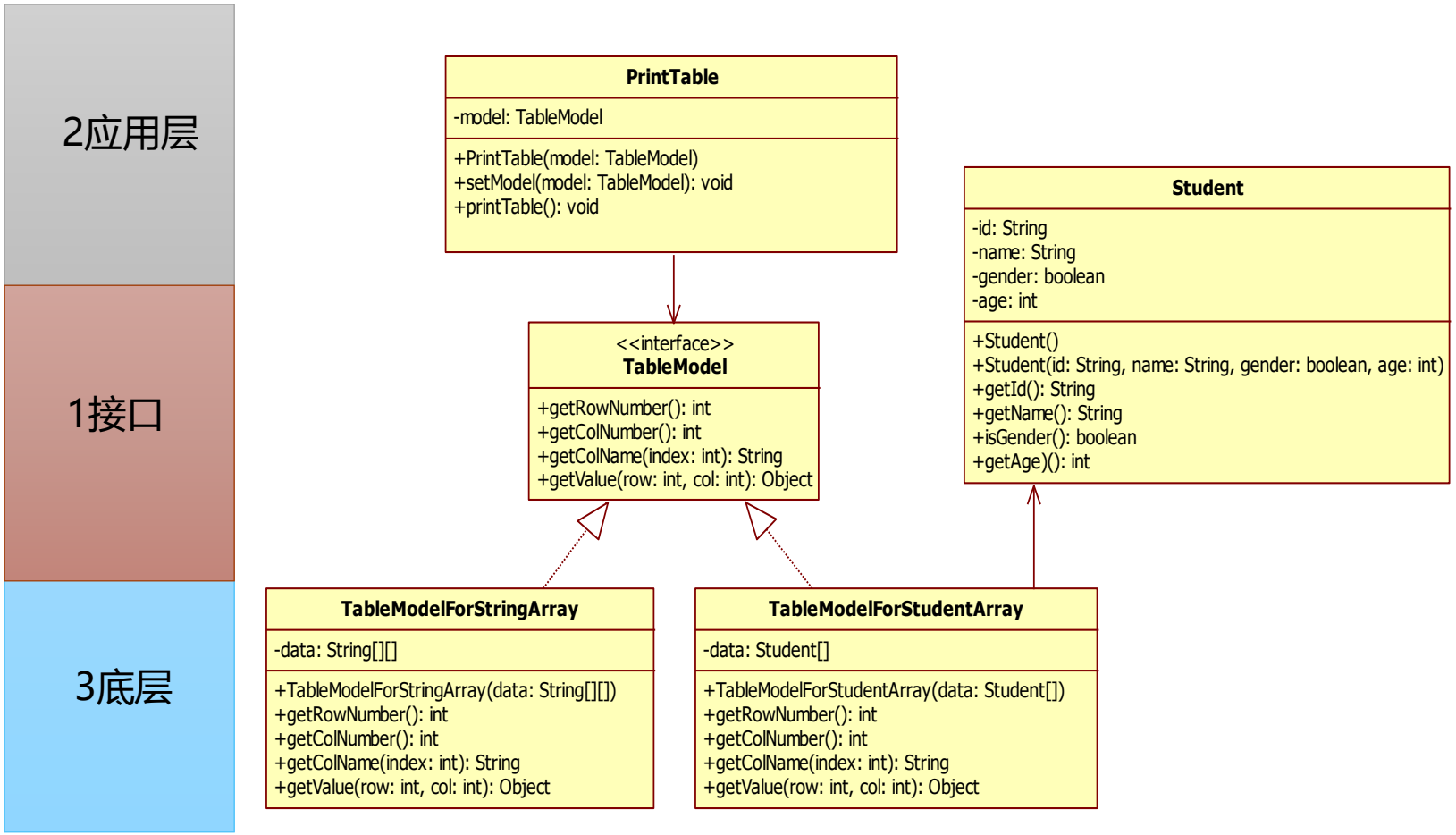
6.5 综合实践--格式化输出学生对象数据

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

- 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	qinggi	男	21

6.5.1 系统架构


2应用层
1接口
3底层
PrintTable
-model: TableModel
+PrintTable(model: TableModel)
+setModel(model: TableModel): void
+printTable(): void
<<interface>>
TableModel
+getRowNumber(): int
+getColNumber(): int
+getColName(index: int): String
+getValue(row: int, col: int): Object
Student
-id: String
-name: String
-gender: boolean
-age: int
+Student()
+Student(id: String, name: String, gender: boolean, age: int)
+getId(): String
+getName(): String
+isGender(): boolean
+getAge)(): int
TableModelForStringArray
-data: String[][]
+TableModelForStringArray(data: String[][])
+getRowNumber(): int
+getColNumber(): int
+getColName(index: int): String
+getValue(row: int, col: int): Object
TableModelForStudentArray
-data: Student[]
+TableModelForStudentArray(data: Student[])
+getRowNumber(): int
+getColNumber(): int
+getColName(index: int): String
+getValue(row: int, col: int): Object

6.5.1 系统架构

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

1、接口层TableModel

- 面向接口的设计中，接口层负责抽象出系统所有的行为方法。输出一个二维表格所需的方法包括：计算表格的行数、计算表格的列数、获取每列的表头名称、获取每行每列元素的取值。

6.5.1 系统架构

2、应用层PrintTable类

- 面向接口的编程中，应用层要依赖接口成员变量完成业务逻辑。
- PrintTable类将TableModel接口的引用变量model作为数据成员；定义构造方法或set方法对接口成员初始化；将打印表格的业务封装在printTable()方法中，printTable()方法通过model调用接口中的方法实现打印表格。

6.5.1 系统架构

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

3、实现类TableModelForStringArray和TableModelForStudentArray

- TableModelForStringArray类基于字符串数组的存储结构实现接口中的所有方法。
- TableModelForStudentArray基于Student对象数组的存储结构实现接口中的所有方法。

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。

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"}
};
```

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;
        }
```

```
public void printTable(){
                for(int i=0; i<model.getColNumber(); i++){  //输出表头
                        System.out.print(model.getColName(i)+"\t");
                }
                System.out.println();
                System.out.println("--------------------------");

                //输出表格内容
                for(int i=0; i<model.getRowNumber(); i++){
                        for(int j=0; j<model.getColNumber(); j++){
                                System.out.print(model.getValue(i, j)+"\t");
                        }
                        System.out.println();
                }
        }
}
```

6.5.2 向接口编程的代码

```
    public int getRowNumber(){//行数：数组元素的个数
                    return data.length;
            }
    public Object getValue(int row, int col){//row为数组下标
            switch(col){//由col定位获取对象的哪个数据成员
                    case 0: return data[row].getId();
                    case 1: return data[row].getName();
                    case 2: return data[row].isGender()?"男":"女";
                    case 3: return data[row].getAge();
            }
            return null;
    }
}
```

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),
};
```

6.5.2 向接口编程的代码

```
public class TableModelForStudentArray implements TableModel{
        private Student[] data;
        public TableModelForStudentArray(Student[] data){
                this.data=data;
        }
        public String getColName(int index) {
                switch(index){       //指定每列名称
                        case 0: return "ID";
                        case 1: return "NAME";
                        case 2: return "GENDER";
                        case 3: return "AGE";
                }
                return null;
        }
        public int getColNumber(){//指定列数
                return 4;
        }
```

6.5.2 向接口编程的代码

```
public class TableModelForStringArray implements TableModel{
        private String[][] data;

        public TableModelForStringArray(String[][] data){
                this.data=data;
        }
        //列名：每列的名字在二维数组的第一行中
        public String getColName(int index){
                return data[0][index];
        }
        public int getColNumber(){ //列数：二维数组每行的长度
                return data[0].length;
        }
        public int getRowNumber(){//行数：二维数组的总行数-1（去掉标题行）
                return data.length-1;
        }
        //数据：在row+1行(越过标题行),col列
        public Object getValue(int row, int col) {
                return data[row+1][col];
        }
}
```

面向对象思想总结

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

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

- 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。

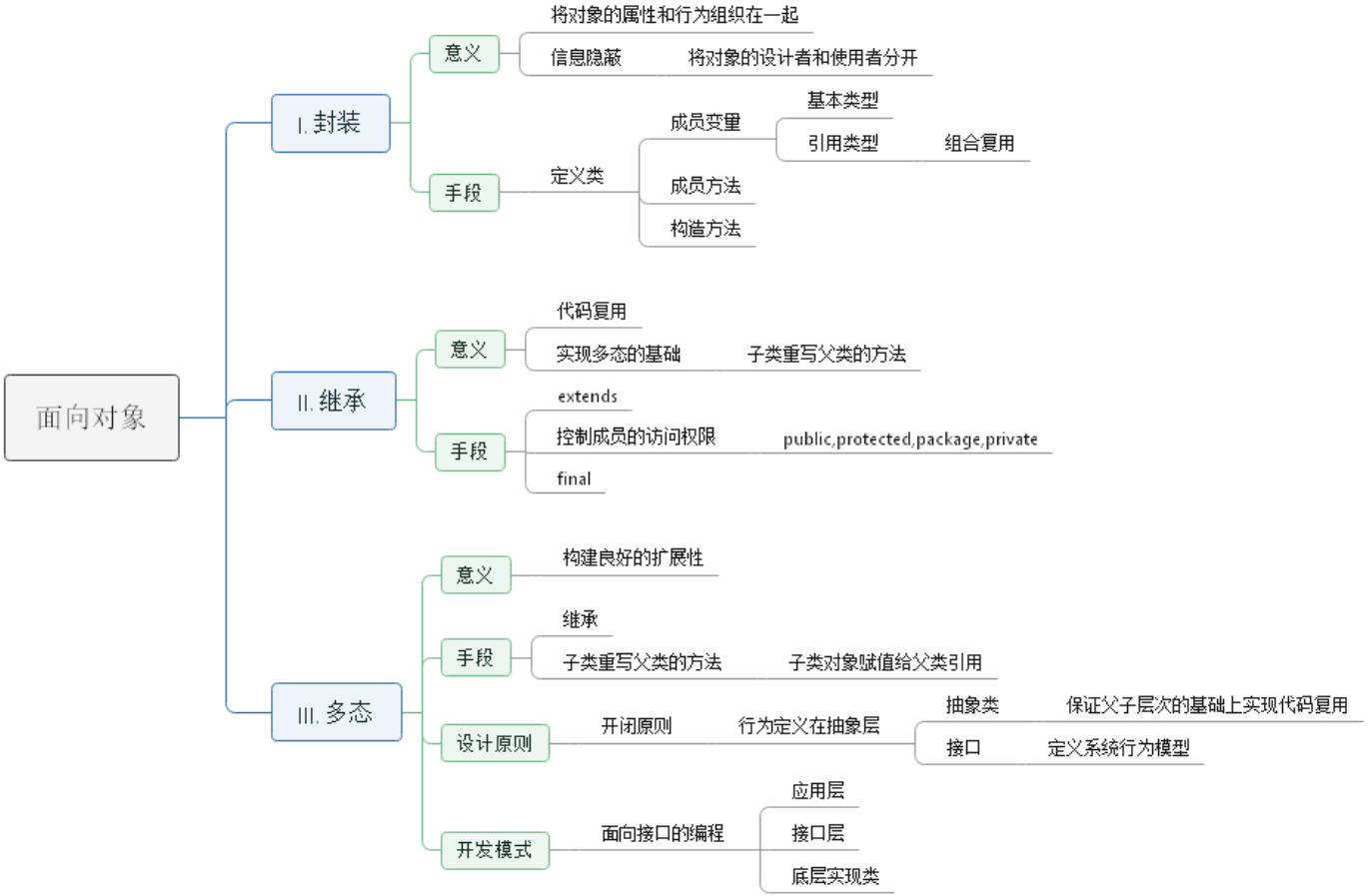
- 子类对象赋给父类引用后的3个层次
  - (1) 父类中没有的方法子类对象不能调用。
  - (2) 如果子类没有覆盖父类的方法则调用父类的方法。
  - (3) 如果子类覆盖父类的方法则调用子类的方法。
- 父类对象转换为子类对象
  - 必须使用强制类型转换。
  - 必须在有意义的情况下才能进行。
  - 在做强制类型转换前，应使用instanceof运算判断引用变量的类型。

- 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


多态性
I. 多态的意义
构建良好的扩展性
II. 多态的原理
继承
子类覆盖父类方法
编译器静态绑定技术
重载解析
private、static、final、构造方法
向下转型
instanceof
判断引用所指对象类型
向上转型
子类对象赋值给父类引用
父类引用只能调用父类中的方法
父类引用调用被子类覆盖的方法，执行子类方法功能
Java虚拟机动态绑定技术
III. 开闭原则
对扩展开放，对修改关闭
实现方式
将行为定义在抽象层
抽象类
abstract
抽象方法
abstract
方法体为空
由子类覆盖实现
非抽象方法、构造方法
成员变量
接口
interface
抽象方法
public、abstract
成员变量
public、static、final
default方法
Java SE 8.0
抽象类和接口的应用
类间的关系不同
抽象类：is a
接口：like a
应用不同
抽象类：保证父子类关的基础上，实现代码复用
接口：定义系统行为模型
IV. 案例
多态的应用
向上转型
向下转型
【例6-1】重写Object类的equals()方法
开闭原则
抽象类
【例6-2】显示各种类型图表
开闭原则
接口
【例6-3】设计用户管理系统的前台管理功能
开闭原则
抽象类和接口的区别
【例6-4】报警门的设计
开闭原则
面向接口的编程
【例6-5】模拟存储设备的读写
开闭原则
面向接口的编程
【综合实践】格式化输出学生对象数据
V. 面向接口的编程
int方法赋值
构造方法赋值
接口引用变量
成员变量
组织接口中的方法
业务行为方法
应用层
行为模型
接口层
定义数据的数据结构
实现接口中的方法
基层

面向对象思维导图


面向对象
I. 封装
意义
将对象的属性和行为组织在一起
信息隐蔽
将对象的设计者和使用者分开
手段
定义类
成员变量
基本类型
引用类型
组合复用
成员方法
构造方法
II. 继承
意义
代码复用
实现多态的基础
子类重写父类的方法
手段
extends
控制成员的访问权限
public,protected,package,private
final
III. 多态
意义
构建良好的扩展性
手段
继承
子类重写父类的方法
子类对象赋值给父类引用
设计原则
开闭原则
行为定义在抽象层
抽象类
保证父子层次的基础上实现代码复用
接口
定义系统行为模型
开发模式
面向接口的编程
应用层
接口层
底层实现类