



第7章 继承



第7章 继承

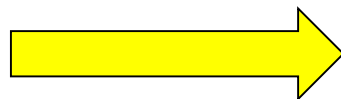
- ◆ 继承的定义与实现
- ◆ 方法重写
- ◆ 抽象类
- ◆ 接口
- ◆ final修饰符
- ◆ Object类
- ◆ 类的关系

第7章 继承

人

属性： 姓名、年龄等

方法： 吃饭、睡觉等



继承

学生

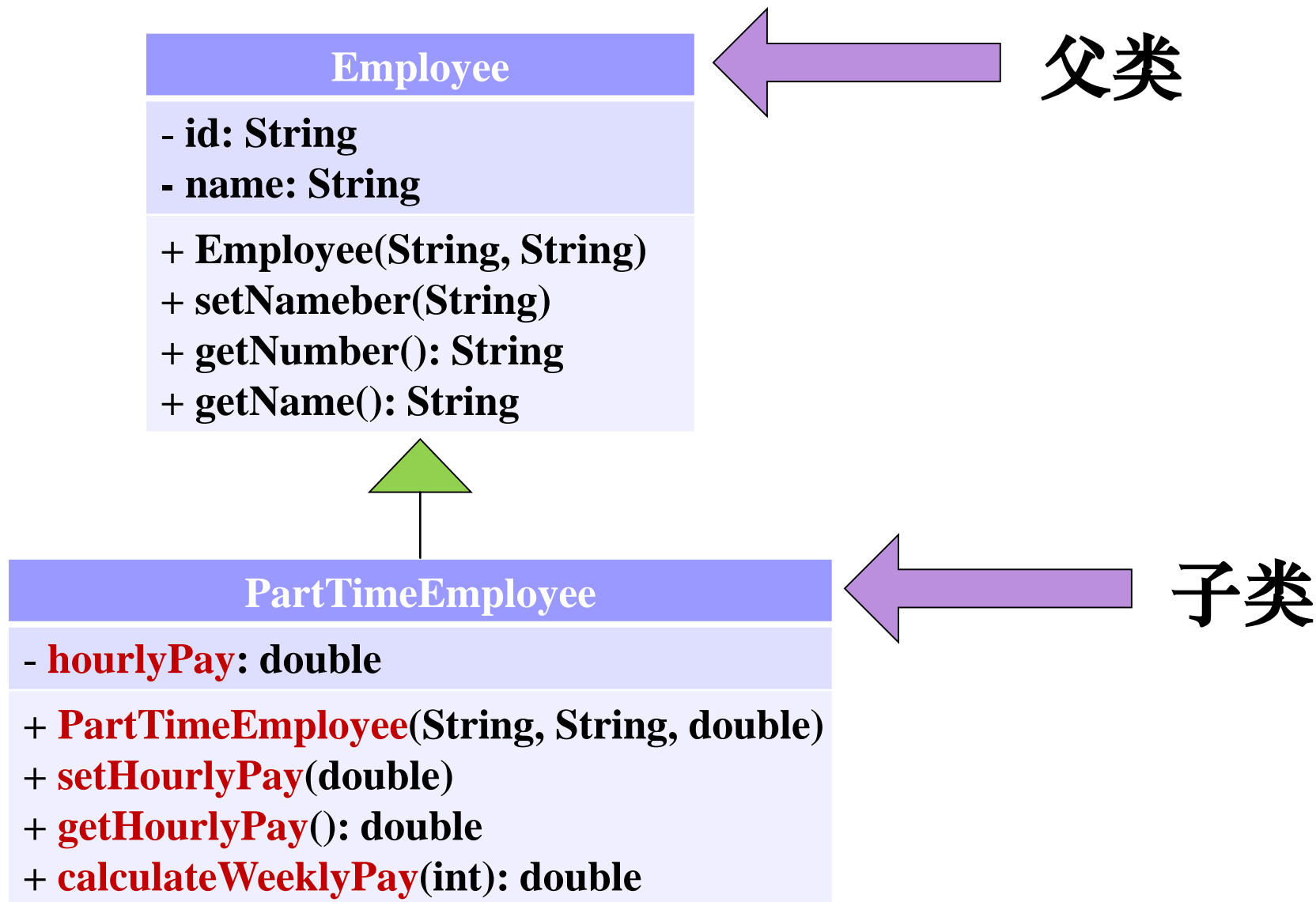


小徐，19岁
方法： 学习、
街舞

7.1 继承的定义与实现

- ◆ 继承可实现类之间共享属性和方法。
- ◆ 继承是在已有类的基础上定义新的类、新的类继承了已有类的属性和方法，又可修改或扩展。

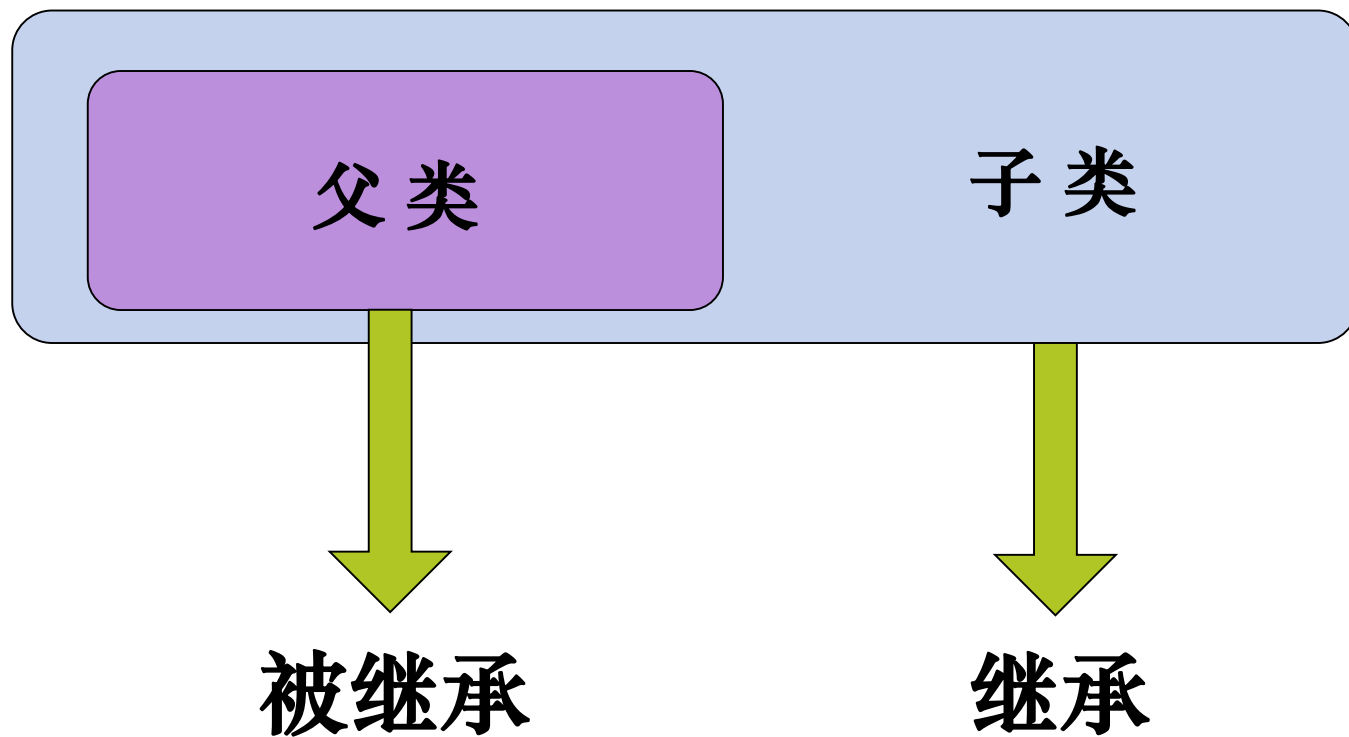
7.1 继承的定义与实现



7.1 继承的定义与实现

- ◆ 继承就是在已有类的基础上构建新的类。
- ◆ 一个类继承一个已有类后，可以对被继承类中的属性和方法进行重用。
- ◆ Java中，一个类只允许有一个父类，不支持多重继承。
- ◆ 多重继承可以通过接口实现。

7.1 继承的定义与实现





```
// Employee.java
```

```
public class Employee{  
    private String id;  
    private String name;  
    public Employee(String idIn, String nameIn){  
        id=idIn;  
        name=nameIn;  
    }  
    public void setName(String nameIn){  
        name=nameIn;  
    }  
    public String getId(){  
        return id;  
    }  
    public String getName(){  
        return name;  
    }  
}
```


// PartTimeEmployee.java

```
public class PartTimeEmployee extends Employee {  
    private double hourlyPay;    //子类新增的属性  
    public PartTimeEmployee(String idIn, String nameIn,  
                             double hourlyPayIn) {  
        super(idIn, nameIn); // 调用父类构造方法  
        hourlyPay = hourlyPayIn;  
    }  
    public double getHourlyPay(){  
        return hourlyPay;  
    }  
    public void setHourlyPay(double hourlyPayIn){  
        hourlyPay=hourlyPayIn;  
    }  
    public double calculateWeeklyPay(int noOfHoursIn){  
        return noOfHoursIn * hourlyPay;  
    }  
}
```

存在的问题

```
// Employee.java
public class Employee{
    private String id;
    private String name;
    .
    .
    .
}
```

```
// PartTimeEmployee.java
public class PartTimeEmployee extends Employee {
    private double hourlyPay;
    .
    .
    .
}
```

解决方法

(1) 可以将原始属性声明为**public**。

```
private String id;  
private String name;
```



```
public String id;  
public String name;
```

违反了封装的原则。

(2) 使用关键字**protected**替换**private**。

```
private String id;  
private String name;
```



```
protected String id;  
protected String name;
```

减弱了信息的封装性。

解决方法

(3) 保持属性为**private**，同时提供这些属性的**public**的get和set方法。

```
private String id;  
private String name;
```



```
public void setName(String nameIn){  
    name = nameIn;  
}
```

```
public String getName(){  
    return name;  
}
```

既保存了数据了封装性，
又允许子类使用。

子类的属性

```
// Employee.java
public class Employee{
    private String id;
    private String name;
    .
    .
    .
}
```

```
// PartTimeEmployee.java
public class PartTimeEmployee extends Employee {
    private double hourlyPay;
    .
    .
    .
}
```

调用父类的构造方法

// Employee.java

```
public class Employee{  
    private String id;  
    private String name;  
    public Employee(String idIn, String nameIn){  
        id=idIn;  
        name=nameIn;  
    }  
}
```

用子类创建对象时，希望给属性id和name赋值。

// PartTimeEmployee.java

```
public class PartTimeEmployee extends Employee {  
    public PartTimeEmployee(String idIn, String nameIn,  
        double hourlyPayIn) {  
        super(idIn, nameIn); // 调用父类构造方法  
    }  
}
```

```
// PartTimeEmployeeTest.java
import java.util.*;
public class PartTimeEmployeeTest{
    public static void main(String args[]){
        String id, name;
        double pay;
        int hours;
        PartTimeEmployee emp;
        Scanner sc=new Scanner(System.in);
        System.out.print("Employee id? ");
        id=sc.next();
        System.out.print("Employee name? ");
        name=sc.next();
        System.out.print("Hourly pay? ");
        pay=sc.nextDouble();
        System.out.print("Hours worked this week? ");
        hous=sc.nextInt();
        emp=new PartTimeEmployee(id, name, pay);
        System.out.println(emp.getName());
        System.out.println(emp.getId());
        System.out.println(emp.calculateWeeklyPay(hous));
    }
}
```

继承

```
class A{  
}
```

```
class B{  
}
```

```
class C extends A{  
}
```

```
class C extends B{  
}
```

```
class A{  
}
```

```
class B extends A{  
}
```

```
class C extends B{  
}
```


继承

```
class A{  
}
```

```
class B extends A{  
}
```

```
class C extends A{  
}
```

继承的结果

◆ 子类（用extends继承）

- 子类自动继承父类的属性和方法
- 子类对成员的修改不会影响父类
- 子类的成员称为复合成员
 - 继承的成员
 - 子类的成员

◆ 父类

- 子类依赖于父类的实现
- 父类进行了修改，可能导致已定义的子类不能正常工作

继承的结果

◆ 超级父类 (Object类)

- 如果一个类没有明确使用extends继承某个类，它将会默认人为Object类的子类。

继承的结果

子类继承父类，子类是否可以直接使用父类中的所有属性和方法？

不可以。

- 父类的构造方法不会被子类继承。
- 子类必须有自己的构造方法。
- 子类可以使用`super(参数列表)`语句调用父类的构造方法。
- 父类中被修饰为`private`的属性和方法，子类也不能直接访问。

7.2方法重写

在**同一个类**中，具有**相同名称**、**不同参数**的方法称之为**方法重载**。

```
public void Person(String nameIn){  
    name = nameIn;  
}  
public void Person(String nameIn, int  
    ageIn){  
    name = nameIn;  
    age = ageIn;  
}
```

7.2方法重写

顾客

只有在消费者缴纳商品货款之后，商品才能被分配给消费者

`dispatchGoods(double) :`
`boolean`

金牌顾客

具有额外的特权，有信用额度。购买商品是允许在信用额度方位内透支

`dispatchGoods(double) :`
`boolean`



7.2 方法重写

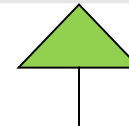
◆ 在方法重载中，不同的方法通过不同的参数列表相互区分。

◆ 在方法重写中，方法名和参数相同，是通过方法隶属的对象不同相互区分。

Customer

Name : String
totalMoneyPaid : double
totalGoodsReceived : double

Customer(String)
getName() : String
getTotalMoneyPaid() : double
getTotalGoodsReceived() : double
calculateBalance() : double
recordPayment(double)
dispatchGoods(double) : boolean



GoldCustomer

creditLimit : double

GoldCustomer(String, double)
getCreditLimit() : double
setCreditLimit(double)
dispatchGoods(double) : boolean

【例7.4】 Customer类

// Customer.java

```
public class Customer{  
    protected String name;  
    protected double totalMoneyPaid;  
    protected double totalGoodsReceived;  
    public Customer(String nameIn){  
        name=nameIn;  
        totalMoneyPaid=0;  
        totalGoodsReceived=0;  
    }  
    public String getName(){  
        return name;  
    }  
    public double getTotalMoneyPaid(){  
        return totalMoneyPaid;  
    }  
}
```



```
public double getTotalGoodsReceived(){  
    return totalGoodsReceived;  
}
```

//计算消费者账户的当前余额

```
public double calculateBalance(){  
    return totalMoneyPaid-totalGoodsReceived;  
}
```

//记录消费者充值操作

```
public void recordPayment(double paymentIn){  
    totalMoneyPaid=totalMoneyPaid+paymentIn;  
}
```


```
public boolean dispatchGoods(double goodsIn){  
    if(calculateBalance()>=goodsIn){  
        totalGoodsReceived=totalGoodsReceived+goodsIn;  
        return true;  
    }else  
        return false;  
}
```

```
}
```

【例7.5】 GoldCustomer子类

// GoldCustomer.java

```
public class GoldCustomer extends Customer{  
    private double creditLimit;  
    public GoldCustomer(String nameIn,double limitIn){  
        super(nameIn);  
        creditLimit=limitIn;  
    }  
    public void setCreditLimit(double limitIn){  
        creditLimit=limitIn;  
    }  
    public double getCreditLimit(){  
        return creditLimit;  
    }  
}
```



```
public boolean dispatchGoods(double goodsIn){  
    if(calculateBalance() + creditLimit >= goodsIn){  
        totalGoodsReceived=totalGoodsReceived  
                                +goodsIn;  
        return true;  
    }else  
        return false;  
}  
}
```

7.2方法重写

◆ 在方法重写中，方法名和参数相同，是通过方法隶属的对象不同相互区分。

```
Customer firstCustomer = new Customer("Mike");
```

```
GoldCustomer secondCustomer =
```

```
    new GoldCustomer("Jean", 1000);
```

```
firstCustomer.dispatchGoods(200);
```

```
secondCustomer.dispatchGoods(370);
```

方法重写时注意的几点

- ◆ 子类方法的名称、参数签名、返回类型必须与父类一致。
- ◆ 子类的方法不能缩小父类方法的访问权限。
- ◆ 子类方法不能抛出比父类方法更多的异常。
- ◆ 方法的重写只能在与子类和父类之间。

7.3 super关键字

- ◆ 调用父类的成员变量或成员方法

```
super . 成员名称;
```

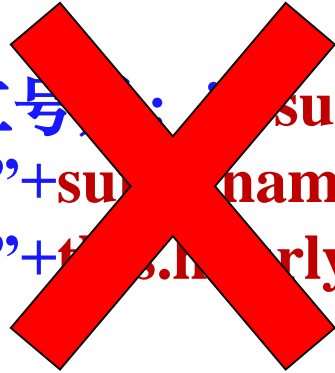
- ◆ 调用父类的构造方法


```
super(参数列表);
```

注意：使用super调用父类构造方法，必须是子类构造方法的第一条语句，因此最多只能调用一次。

```
class Employee{
    private String id;
    private String name;
    .
    .
    public print(){
        System.out.println("该员工的工号是: "+this.id);
        System.out.println("\t姓名是: "+this.name);
    }
}
```

```
class PartTimeEmployee extends Employee() {
    public print(){
        System.out.println("该员工的工号: "+super.id);
        System.out.println("\t姓名是: "+super.name);
        System.out.println("\t姓名是: "+this.monthlyPay);
    }
}
```





```
class Employee{  
    private String id;  
    private String name;  
    .  
    .  
    public print(){  
        System.out.println("该员工的工号是: "+this.id);  
        System.out.println("\t姓名是: "+this.name);  
    }  
}
```

```
class PartTimeEmployee extends Employee() {  
    public print(){  
        super.print();  
        System.out.println("\t姓名是: "+this.hourlyPay);  
    }  
}
```


super与 this

	this	super
访问属性	访问本类中的属性，如果本类中没有此属性，则从父类中继续查找	访问父类中的属性
访问方法	访问本类中的方法，如果本类中没有此方法，则从父类中继续查找	直接访问父类中的方法
调用构造方法	调用本类的构造方法，必须放在构造方法的首行	调用父类构造方法，必须放在子类构造方法的首行

7.4 抽象类

用**abstract**修饰类，称为**抽象类**。

- ◆ 抽象类可以包含抽象方法、也可以没有抽象方法
- ◆ 抽象方法只有方法的声明，没有方法的实现
- ◆ 方法的实现在子类中完成
- ◆ 可以包含非抽象方法
- ◆ 抽象方法只能在抽象类中声明，不可以出现在非抽象类中

7.4 抽象类

- ◆ 因为抽象类不能实例化对象，所以必须要有子类来实现它之后才能使用。这样就可以把一些具有相同属性和方法的组件进行抽象，这样更有利于代码和程序的维护。
- ◆ 当又有一个具有相似的组件产生时，只需要实现该抽象类就可以获得该抽象类的那些属性和方法。

【例7.6】 Employee抽象类

// Employee.java

```
public abstract class Employee {           // 抽象类
    private String id;
    private String name;
    public Employee(String idIn, String nameIn){
        id=idIn;
        name=nameIn;
    }
    public void setName(String nameIn){
        name=nameIn;
    }
    public String getId(){
        return id;
    }
    public String getName(){
        return name;
    }
    abstract public String getStatus();    //抽象方法
}
```

【例7.7】 FullTimeEmployee子类

// FullTimeEmployee.java

```
public class FullTimeEmployee extends Employee{
    private double monthlyPay;
    public FullTimeEmployee(String idIn, String nameIn,
double hourlyPayIn){
        super(idIn, nameIn);
        hourlyPay=hourlyPayIn;
    }
    public double getMonthlyPay(){
        return monthlyPay;
    }
    public void setMonthlyPay(double monthlyPayIn){
        monthlyPay = monthlyPayIn;
    }
    public double calculateAnnualPay(){
        return monthlyPay * 12;
    }
    public String getStatus(){           //实现抽象方法
        return "Full-Time";
    }
}
```

【例7.8】 PartTimeEmployee类

// PartTimeEmployee.java

```
public class PartTimeEmployee extends Employee{
    private double hourlyPay;
    public PartTimeEmployee(String idIn, String nameIn,
double hourlyPayIn){
        super(idIn, nameIn);
        hourlyPay=hourlyPayIn;
    }
    public double getHourlyPay(){
        return hourlyPay;
    }
    public void setHourlyPay(double hourlyPayIn){
        hourlyPay=hourlyPayIn;
    }
    public double calculateWeeklyPay(int noOfHoursIn){
        return noOfHoursIn * hourlyPay;
    }
    public String getStatus(){           //实现抽象方法
        return "Part-Time";
    }
}
```

7.5 接口

接口的定义如下：

```
public interface <接口名>{  
    public static final type valueName;  
    public abstract returnTypemethod (value);  
}
```

接口中只有常量和抽象方法，例子：

```
public interface Runner{  
    int ID=1;  
    void run();  
}
```

7.5 接口

接口作用：

1、丰富Java面向对象的思想：在Java语言中，**abstract class** 和 **interface** 是支持抽象类定义的两机制。正是由于这两种机制的存在，才赋予了Java强大的面向对象能力。

2、提供简单、规范性

3、提高维护、拓展性。

4、增强安全、严密性。

接口实例

- ◆ 定义一个接口，用**extends**关键字去**继承**一个接口
- ◆ 定义一个**类**，用**implements**关键字去实现接口中的所有方法
- ◆ 定义一个**抽象类**，用**implements**关键字实现接口中定义的部分方法

【例7.11】 接口实例

```
public interface Runner{
    void run();
}
interface Animal extends Runner{
    void breathe();
}
class Fish implements Animal{
    public void run(){
        System.out.println("fish is swimming");
    }
    public void breathe(){
        System.out.println("fish is bubbling");
    }
}
abstract LandAnimal implements Animal{
    public void breathe(){
        System.out.println("LandAnimal is breathing");
    }
}
```

7.6 final修饰符

final修饰类，意味着这个类**不能被继承**，声明的格式为：

```
final class finalClassName{  
    .....  
}
```

以下程序是否有错误？

```
public final class FinalClass{  
    int member;  
    void memberMethod(){};  
}
```

```
class SubFinalClass extends FinalClass{  
    int submember;  
    void subMemberMethod(){};  
}
```

7.6 final修饰符

final修饰方法，意味着这个方法**不能被重写**。

```
class FinalMethodClass{  
    final void finalMethod(){  
        ...  
    }  
}
```

以下程序是否有错误？

```
class FinalMethodClass{
    final void finalMethod(){
        ...
    }
}

class OverloadClass extends Final methodClass{
    void finalMethod(){
        ...
    }
}
```

7.7 Object类

- ◆ 如果一个类没有使用**extends**关键字明确表示继承另一个类，这个类默认**继承Object类**。
- ◆ **Object**是Java类层次中最高层，是**所有类的父类**。
- ◆ **Object**类中的方法**适用于所有类**。

7.7.1 toString()方法

- ◆ 该方法返回一个字符串，字符串由类名等信息组成。
- ◆ 该方法用于在实际运行或调试代码时获取字符串表示的对象状态信息。

```
public String toString()
```

Java中的大多数类都重写了该方法，通常的方式是将类名以及成员变量的状态信息组合转换为一个字符串返回。

7.7.2 equals()方法

- ◆ 该方法用于比较两个对象是否相等。
- ◆ String类的equals()方法比较两个字符串对象是否相等。
- ◆ equals()方法来自Object类，String类对其进行了重写以满足比较字符串内容的要求。

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


- ◆ Object类的实现没有比较两个对象是否逻辑上相等，而只是对两个引用进行了“==”比较，相当于比较两个引用是否指向同一个对象。

7.7 Object类

【例7.12】 toString()应用

// Student.java

```
public class Student{  
    public String name;  
    public int age;  
    public Student(){  
    }  
    public Student(String nameIn, int ageIn){  
        name=nameIn;  
        age=ageIn;  
    }  
    public String toString(){  
        return “ student name is ” + name +” , age is” + age;  
    }  
}
```



```
public static void main(String ags[]){  
    Object st=new Student("Jenny", 20);  
    System.out.println(st.toString());  
    System.out.println(st);  
}  
}
```

运行结果：

Student name is Jenny, age is 20

Student name is Jenny, age is 20

7.7.2 equals()方法

- ◆ 用于比较两个对象是否相等。
- ◆ 该方法来自Object类，String类对其进行了重写以满足比较字符串内容的要求。
- ◆ Object类设计该方法就是为了让继承它的子类重写。

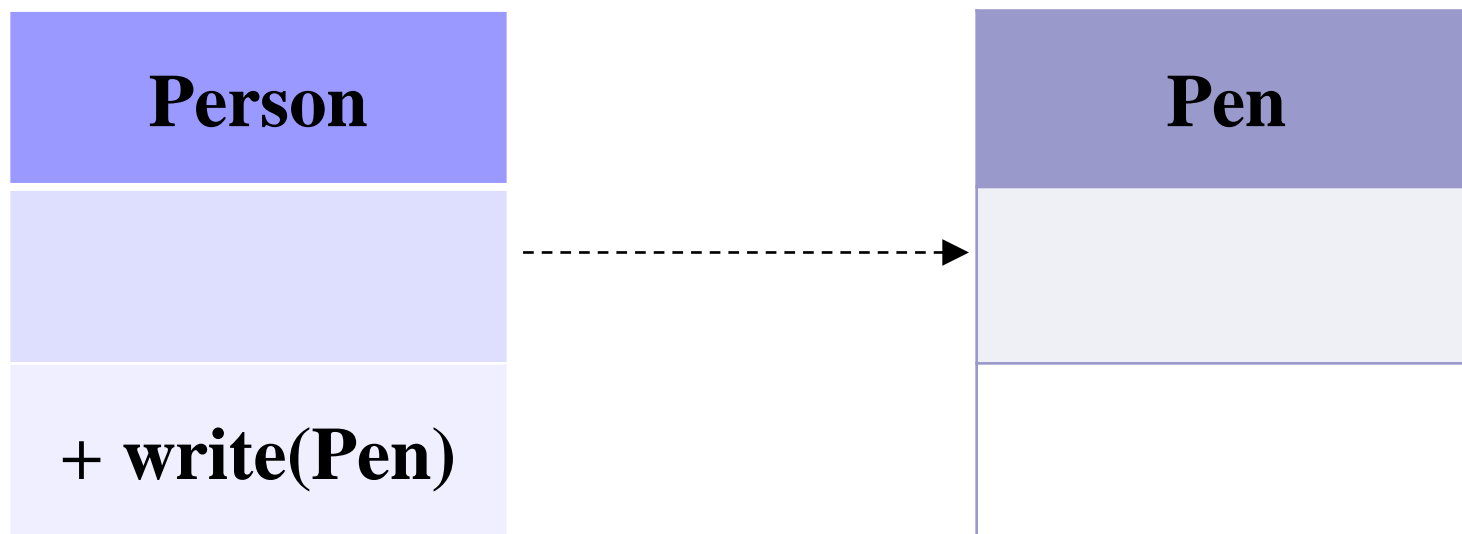
```
public Boolean equals(Object obj){  
    return (this == obj);  
}
```

7.8 类的关系

- ◆ 类别上可分为：纵向关系和横向关系。
- ◆ 纵向关系：继承关系
- ◆ 横向关系：
 - 依赖 (Dependency)
 - 关联 (Association)
 - 聚合 (Aggregation)
 - 组合 (Composition)

7.8.1 依赖

- ◆ 依赖关系是一种使用关系，特定事务的改变有可能会影响到使用它的事务，反之成立。



- ◆ 通常情况下，依赖关系体现在某个类的方法使用另一个类作为参数。

7.8.1 依赖

码示例：

```
public class Pen { //笔 可用于人类书写
```

```
.....
```

```
}
```

```
public class Person{
```

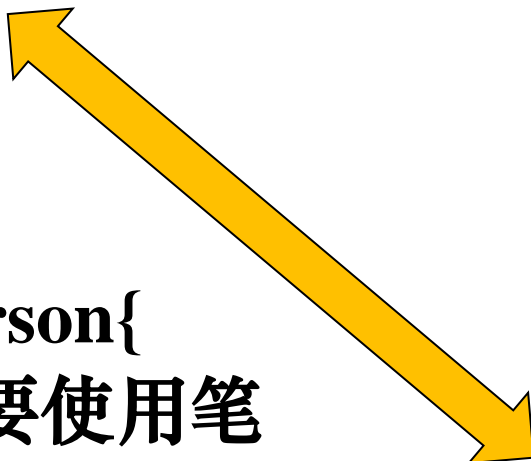
```
    //书写需要使用笔
```

```
    public void write(Pen mypen){
```

```
        .....
```

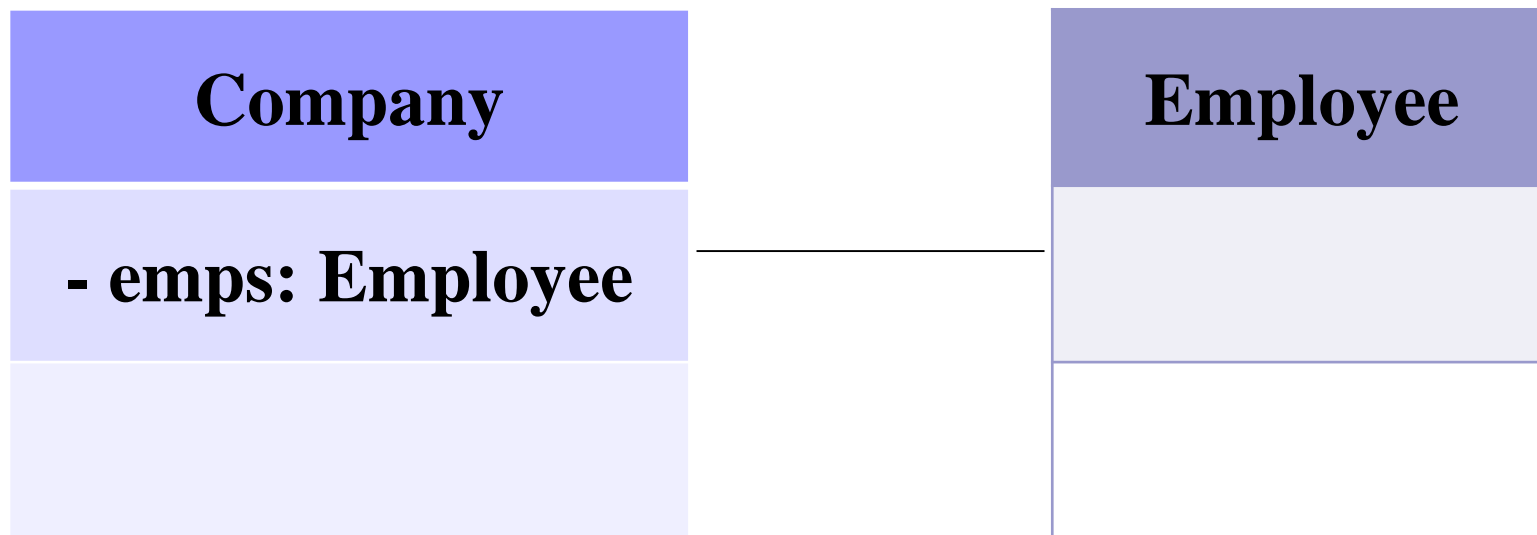
```
    }
```

```
}
```



7.8.2 关联

- ◆ 关联是一种结构关系，说明一个事物的对象与另一个事物的对象相联系。



- ◆ 通常情况下，关联关系是通过类的成员变量来实现。

7.8.2 关联

代码示例：

```
public class Company { //公司  
    private Employee emp;
```

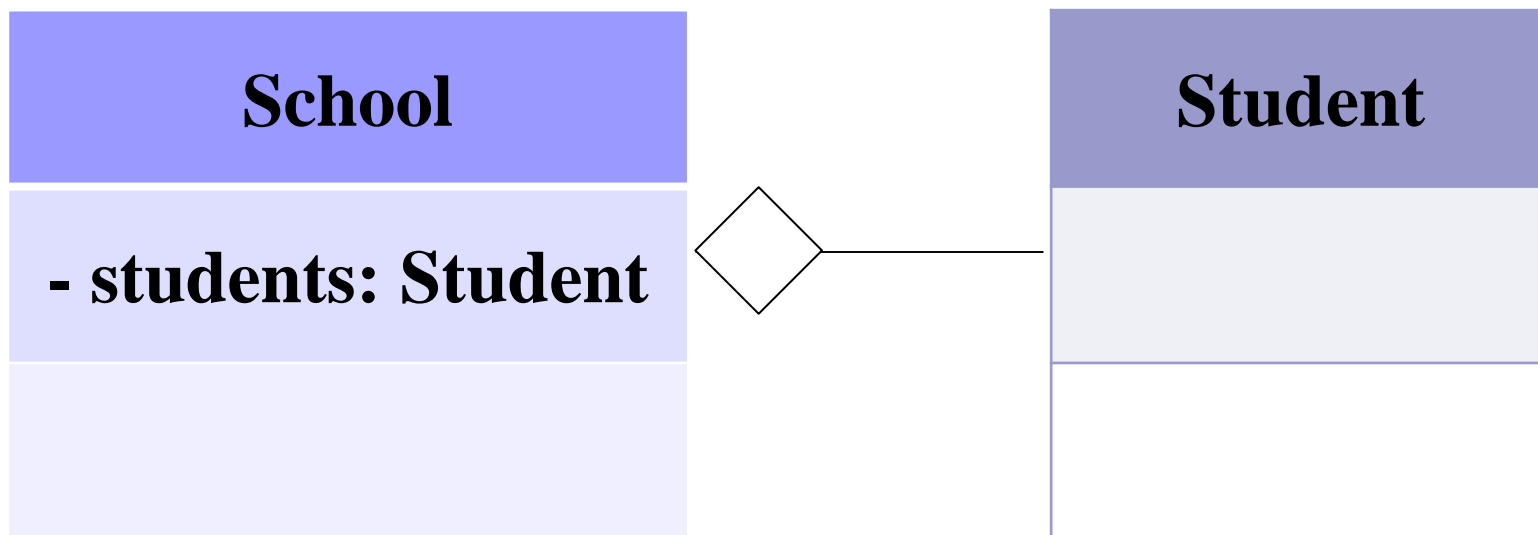
```
//一个公司雇员  
}
```

公司与雇员之间就是一种关联关系。

```
public class Employee{  
  
}
```

7.8.3 聚合

- ◆ 表示整体对象拥有部分对象。
- ◆ 关联关系和聚合关系在语法上无法区分，从语义上才能更好区分。



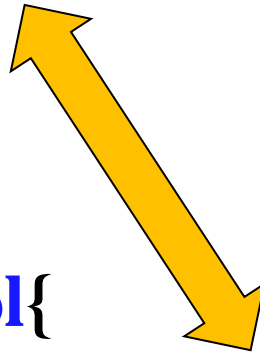
- ◆ 关联关系是通过类的成员变量来实现。

7.8.3 聚合

代码示例：

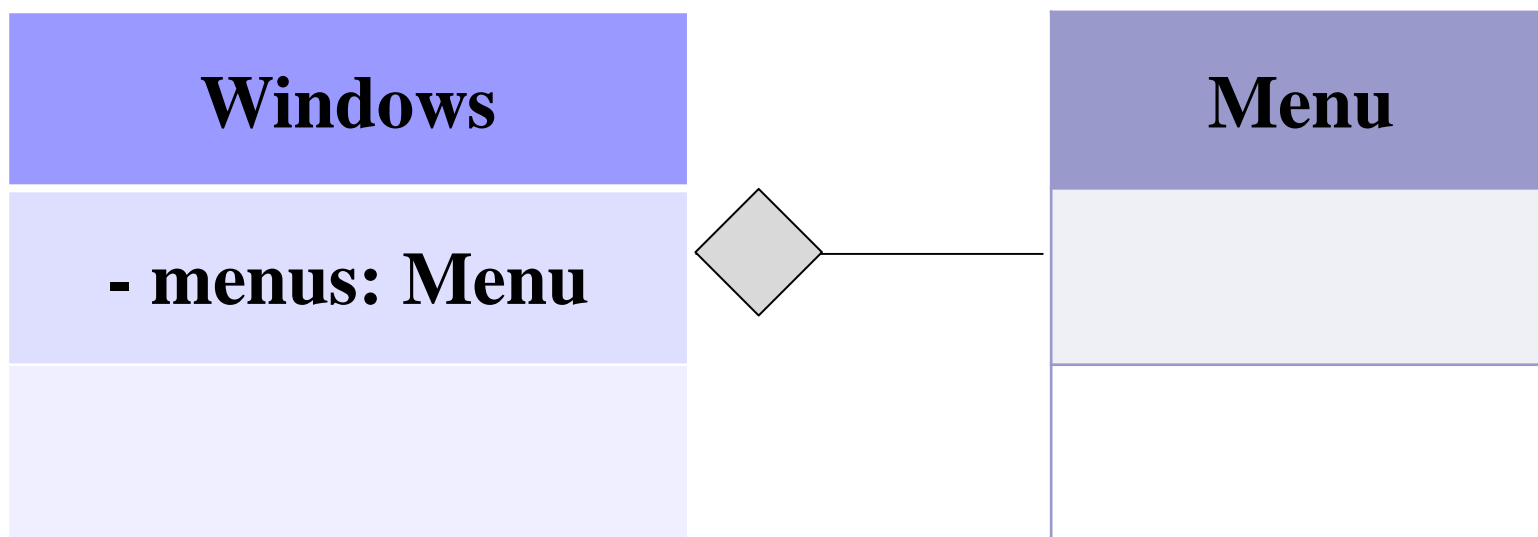
```
public class Student {  
}
```

```
public class School{  
    private Student[] students;  
}
```



7.8.4 组合

- ◆ 具有更强的拥有关系，强调整体与部分的生命周期是一致的，整体负责部分的生命周期的管理。



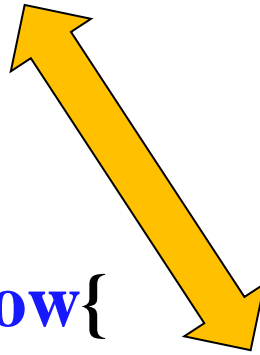
- ◆ 组合关系是通过类的成员变量来实现。

7.8.4 组合

代码示例：

```
public class Menu {  
}
```

```
public class Window{  
    private List <Menu> menus;  
}
```

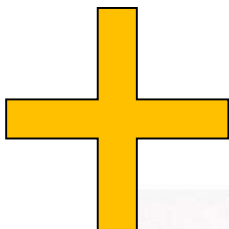


7.9 综合实例：组装计算机

说明：编写管理组装计算机的程序，计算机的主要部件有主板，主板中可以插入显卡、CPU等。允许用户设定显卡、CPU的型号。



主板



显卡



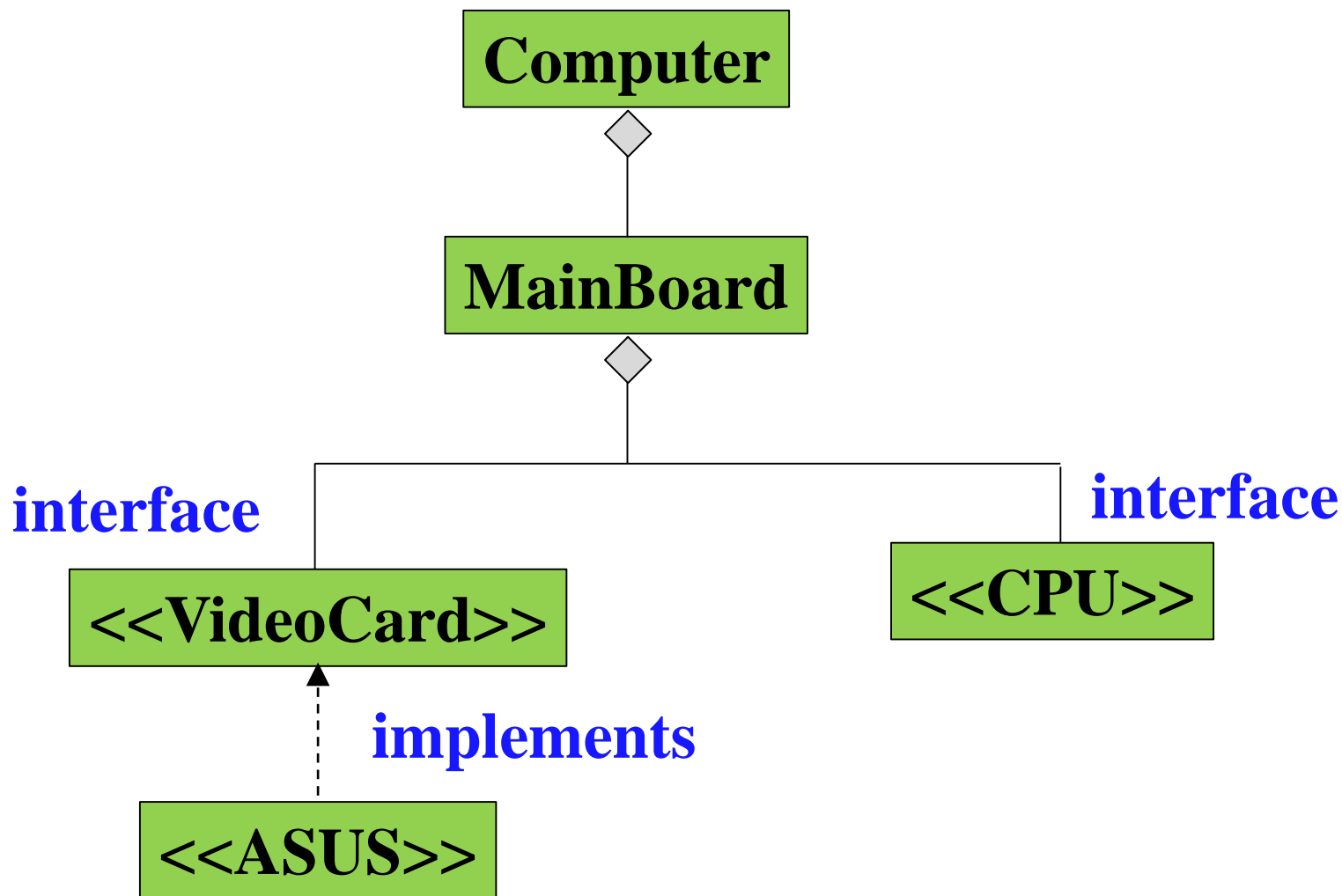
cpu

7.9 综合实例：组装计算机

分析：

- ◆ 主板是计算机的一部分，主板与计算机是组合关系；
- ◆ 主板由显卡、CPU等组成，因此也是组合关系；
- ◆ 显卡、CPU型号可以更改，考虑用接口定义显卡，组装时再确定显卡和CPU用何种型号。

7.9 综合实例：组装计算机



Computer:

- ◆ 创建显卡对象
- ◆ 创建主板对象
- ◆ 设定CPU型号
- ◆ 插入显卡
- ◆ 运行主板，开机



Mainboard:

- ◆ 设定CPU
- ◆ 插入显卡
- ◆ 显示CPU
- ◆ 显卡工作
- ◆ 主板正常工作



主板

VideoCard:

- ◆ 显卡是否工作
- ◆ 获取显卡名称



显卡

ASUS:

- ◆ 构造方法
- ◆ 给显卡设定名称
- ◆ 显卡是否工作
- ◆ 获取显卡名称



主板

```
package lesson8;
interface VideoCard{
    void Display();
    String getName();
}
class Mainboard{
    String strCPU; //CPU的名字
    VideoCard vc; //显卡
    void setCPU(String strCPU){           //设定CPU
        this.strCPU=strCPU;
    }
    //插入显卡
    void setVideoCard(VideoCard vc){
        this.vc=vc;
    }
}
```

//主板运行，模仿开机显示必要信息

void run(){

System.out.println(strCPU); //显示CPU

System.out.println(vc.getName()); //显示显卡的名称

vc.Display(); //显卡工作

System.out.println("Mainboard is working..."); //主板正常工作

}

}

class ASUS implements VideoCard{

String name; //显卡的名字

public ASUS(){ //构造方法

name="ASUS's video card";

}



//给OEM厂商等设定名称

```
public void setName(String name){  
    this.name=name;  
}  
public void Display(){  
    System.out.println("ASUS's video card is working..");  
}  
public String getName(){  
    return name;  
}  
}
```

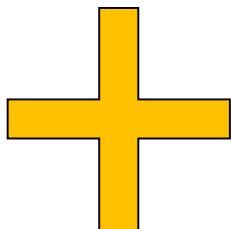
```
class Computer{
    public static void main(String[] args){
        ASUS d=new ASUS();           //购买显卡
        Mainboard m=new Mainboard(); //购买主板
        m.setCPU("Intel's CPU");     //设定CPU
        m.setVideoCard(d);           //插入显卡
        m.run();                     //运行主板，开机
    }
}
```

7.9 综合实例：组装计算机（作业）

说明：主板中插入内存、风扇等。允许用户设定内存、风扇的型号。



主板



cpu



内存



风扇

7.9 综合实例：组装计算机（作业）

分析：

◆ 内存、风扇型号可以更改，考虑用接口定义内存和风扇，**组装时**再确定内存和风扇用**何种型号**。

➤ 内存（RAM）

➤ 风扇（FAN）

(1) 重载方法和重写方法的区别

重载:

在同一个类中，声明多个同名方法，通过参数列表来区分不同的方法（参数列表的数量，类型、顺序）

(1) 重载方法和重写方法的区别

重写:

前提是发生在具有继承关系的**两个类之间**（子类可以继承父类非私有的方法），当父类方法不能满足子类需求时，子类可以对继承的方法进行重新编写。

重写规则:

- 参数列表必须保持一致；
- 返回值类型必须保持一致；
- 方法名必须保持一致；
- 重写的方法的访问权限范围必须大于等于父类方法；
- 重写方法抛出的异常范围不能大于父类方法；

(2) 抽象类和接口之间的区别

abstract修饰符：

- 1.abstract修饰的类为**抽象类**，此类**不能有对象**；
- 2.abstract修饰的方法为**抽象方法**，此方法**不能有方法体**（就是什么内容不能有）；

关于抽象类的使用特点：

- 1.抽象类**不能有对象**，（不能用new此关键字来创建抽象类的对象）；
- 2.**有抽象方法的类一定是抽象类**，但是抽象类中不一定有抽象方法；
- 3.抽象类中的抽象方法**必须在子类中被重写**。

(2) 抽象类和接口之间的区别

接口就是一个规范和抽象类比较相似。它只管做什么，不管怎么做。通俗的讲，借口就是某个事物对外提供的一些功能的声明，其定义和类比较相似，只不过是通过interface关键字来完成。

1.接口中的所有属性默认为：

`public static final ****;`

2.接口中的所有方法默认为：

`public abstract ****;`

3.接口不再像类一样用关键字 `extends` 去“继承”，而是用 `implements` 去“实现”，也就是说类和接口的关系叫做实现，

（例如：A类实现了B接口，那么成为A为B接口的实现类。而类与类之间的继承的话，叫做A类继承了B类，其中B类即为A类的父类）。实现接口与类的继承比较相似