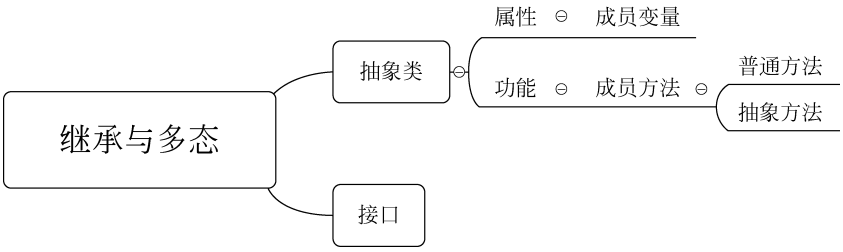
第5章 继承与多态

本章导图：



主要内容：

- 子类与父类。
- 子类的继承性。
- 子类对象的构造过程。
- 成员变量的隐藏和方法重写。
- super 关键字。
- final 关键字。
- 对象的上转型对象。
- 抽象类。
- 接口。

难点：

- 成员变量的隐藏和方法重写。
- 抽象类。

Java 程序设计的过程就是设计类的过程。Java 的类只能有一个直接父类，为了实现多重继承，Java 语言引入了接口的概念。在 Java 中定义好的类依据实现功能的不同，可以分为不同的集合，每个集合是一个包，所有的包合称为类库。

本章将进一步讨论 Java 类、包和接口等面向对象机制。首先给出基本概念，然后结合具体实例阐明 Java 的类、接口、包以及封装、继承、重载等有关内容。

5.1 继　　承

继承性是面向对象的重要特性。继承允许一个类成为另一个类的子类，子类继承了父类的所有特性，并且可以扩展出自己的特征。类的继承性提供了一种明确描述共性的

方法，减少了类似的重复说明。继承机制提高了软件的可用性、代码的复用性以及界面的一致性。

通过使用子类，可以实现继承。从最一般的类开始，逐步特殊化，可派生出一系列的子类。父类和子类之间的关系呈现出层次化。同时，继承实现的代码复用，使程序复杂度线性地增长，而不是呈几何级数增长。在 Java 中任何一个类都有父类（除了 object 类以外）。Java 只支持单重继承，大大降低了继承的复杂度。

5.1.1 子类与父类

由继承而得到的类称为子类，被继承的类称为父类（超类）。Java 不支持多重继承（子类只能有一个父类）。

在类的声明中，通过使用关键字 extends 来声明一个类的子类，格式如下。

```
class 子类名 extends 父类名{
    …
}
```

例如：

```
class Student extends People {
    …
}
```

把 Student 声明为 People 类的子类，People 是 Students 的父类。

如果一个类的声明中没有使用 extends 关键字，这个类被系统默认为是 Object 的子类。Object 是 java.lang 包中的类。

5.1.2 类的继承性

类可以有两种重要的成员：成员变量和方法。子类的成员中有一部分是子类自己声明定义的，另一部分是从它的父类继承的。那么，什么叫继承呢？所谓子类继承父类的成员变量作为自己的一个成员变量，就好像它们是在子类中直接声明一样，可以被子类中自己声明的任何实例方法操作，也就是说，一个子类继承的成员应当是这个类的完全意义的成员，如果子类中声明的实例方法不能操作父类的某个成员变量，该成员变量就没有被子类继承；同样，子类继承父类的方法作为子类中的一个方法，就像它们是在子类中直接声明一样，可以被子类中自己声明的任何实例方法调用。

例 5.1

Example5_1.java

```
public class Example5_1{
    public static void main (String args[ ]) {
        Student mike = new Student("Mike", 18,47,98);
        System.out.println(mike.getName());
        mike.doHomework();
    }
}
```

例 5.1
视频讲解

Person. java

```
public class Person {
protected String name;
protected int age;

public Person()
{
}

public Person(String name, int age)
{
    this.name = name;
    this.age = 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 void eat()
{
    System.out.println(this.name + " am eating.");
}

public void sleep()
{
    System.out.println(this.name + " am sleeping.");
}
```

Student. java

```
public class Student extends Person{
     int num;
     int javaGrade;
```

```
    public Student()
    {
    }

    public Student(String name, int age)
    {
        super(name,age);
    }

    public Student(String name, int age, int num, int classNum)
    {
        super(name,age);
        this.num = num;
        this.javaGrade = classNum;
    }

    public int getNum()
    {
        return num;
    }

    public void setNum(int num)
    {
        this.num = num;
    }

    public int getJavaGrade()
    {
        return javaGrade;
    }

    public void setJavaGrade(int javaGrade)
    {
        this.javaGrade = javaGrade;
    }

    public void doHomework()
    {
        System.out.println(this.name + " do homework.");
    }
}
```

例 5.1 的程序运行结果如图 5.1 所示。

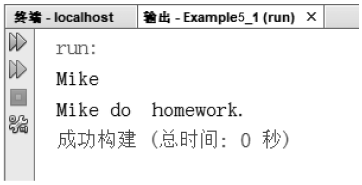


图 5.1 例 5.1 的程序运行结果

5.1.3 子类对象的构造过程

当用子类的构造方法创建一个子类的对象时，子类的构造方法总是先调用父类的某个构造方法，也就是说，如果子类的构造方法没有明显地指明使用父类的哪个构造方法，子类

就调用父类不带参数的构造方法。因此，当用子类创建对象时，不仅子类中声明的成员变量被分配了内存，而且父类的成员变量也都分配了内存空间，但只将其中一部分（子类继承的那部分）作为分配给子类对象的变量。也就是说，父类中的 private 成员变量尽管分配了内存空间，也不作为子类对象的变量，即子类不继承父类的私有成员变量。同样，如果子类和父类不在同一包中，尽管父类的友好成员变量分配了内存空间，但也不作为子类的成员变量，即如果子类和父类不在同一包中，子类不继承父类的友好成员变量。

通过上面的讨论，读者可能有这样的感觉：子类创建对象时似乎浪费了一些内存，因为当用子类创建对象时，父类的成员变量也都分配了内存空间，但只将其中一部分作为分配给子类对象的变量，例如，父类中的 private 成员变量尽管分配了内存空间，也不作为子类对象的变量，当然它们也不是父类某个对象的变量，因为根本就没有使用父类创建任何对象。这部分内存似乎成了垃圾，但实际情况并非如此，需注意到，子类中还有一部分方法是从父类继承的，这部分方法却可以操作这部分未继承的变量。

在例 5.2 中，子类对象调用继承的方法操作这些未被子类继承却分配了内存空间的变量。

例 5.2

Example5_2.java

```
public class Example5_2 {
    public static void main (String args[ ]) {
        ClassB b = new ClassB();
        b.setX(888);
        System.out.println("子类对象未继承的 x 的值是: " + b.getX());
        b.setY(12.678);
        System.out.println("子类对象的实例变量 y 的值是: " + b.getY());}
}
```

ClassA.java

```
public class ClassA {
    private int x;
    public void setX(int x) {
        this.x = x;
    }

    public int getX() {
        return x;
    }
}
```

ClassB.java

```
public class ClassB extends ClassA
{
   private double y = 12;
    public double getY() {
        return y;
    }
```

```
    public void setY(double y) {
        this.y = y;
        //this.y = y+x; 非法,子类没有继承 x
    }
}
```

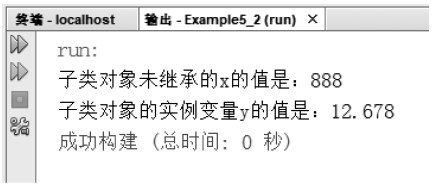


图 5.2　例 5.2 的程序运行结果

例 5.2 的程序运行结果如图 5.2 所示。

5.1.4　成员变量的隐藏和方法重写

子类也可以隐藏继承的成员变量，对于子类可以从父类继承成员变量，只要子类中定义的成员变量和父类中的成员变量同名时，子类就隐藏了继承的成员变量，即子类对象以及子类自己声明定义的方法操作与父类同名的成员变量是指子类重新声明定义的这个成员变量。

子类可以隐藏已继承的方法，子类通过方法重写来隐藏继承的方法。方法重写是指：子类中定义一个方法，并且这个方法的名字、返回类型、参数个数和类型与从父类继承的方法完全相同。子类通过方法的重写可以隐藏继承的方法，子类通过方法的重写可以把父类的状态和行为改变为自身的状态和行为。如果父类的方法 f()可以被子类继承，子类就有权利重写 f()，一旦子类重写了父类的方法 f()，就隐藏了继承的方法 f()，那么子类对象调用方法 f()一定是调用重写的方法 f()，重写的方法既可以操作继承的成员变量也可以操作子类声明定义的成员变量。如果子类想使用被隐藏的方法，必须使用关键字 super。

5.1.5　super 关键字

子类可以隐藏从父类继承的成员变量和方法，如果在子类中想使用被子类隐藏的成员变量或方法，就可以使用关键字 super。

1. 使用 super 调用父类的构造方法

子类不继承父类的构造方法，因此，子类如果想使用父类的构造方法，必须在子类的构造方法中使用，并且必须使用关键字 super 来表示，而且 super 必须是子类构造方法中的第一条语句。

需要注意的是，如果在子类的构造方法中，没有明显地写出 super 关键字来调用父类的某个构造方法，那么默认地有：

```
super( );
```

即调用父类的不带参数的构造方法。

如果类里定义了一个或多个构造方法，那么 Java 不提供默认的构造方法(不带参数的构造方法)，因此，当在父类中定义多个构造方法时，应当包括一个不带参数的构造方法，以防子类省略 super 时出现错误。

2. 使用 super 操作被隐藏的成员变量和方法

在子类中想使用被子类隐藏的成员变量或方法时就可以使用关键字 super。例如，super. x、super. play( )就是访问和调用被子类隐藏的成员变量 x 和方法 play( )。

需要注意的是，当子类创建一个对象时，除了子类声明的成员变量和继承的成员变量要分配内存外(这些内存单元是属于子类对象的)，被隐藏的成员变量也要分配内存，但该内存

单元不属于任何对象，这些内存单元必须用 super 调用。同样，当子类创建一个对象时，除了子类声明的方法和继承的方法要分配入口地址外，被隐藏的方法也要分配入口地址，但该入口地址只对 super 可见，所以必须由 super 来调用。当 super 调用隐藏的方法时，该方法中出现的成员变量是指被隐藏的成员变量。

在下面的例 5.3 中，子类 Average 使用 super 调用隐藏的方法。

例 5.3

Example5_3.java

```
public class Example5_3 {
    public static void main(String args[ ]) {
       ClassB classB = new ClassB();
        classB.n = 100;
        double result1 = classB.sum();
        double result2 = classB.sub();
        System.out.println("result1 = " + result1);
        System.out.println("result2 = " + result2);
}
```

ClassA.java

```
public class ClassA {
    int n;
    public double sum() {
        double sum = 0;
        for (int i = 1; i <= n; i++) {
            sum = sum + i;
        }
        return sum;
    }
}
```

ClassB.java

```
public class ClassB extends ClassA {
    double n;
    public double sum() {
        double c;
        super.n = (int) n;
        c = super.sum();
        return c + n;
    }
    public double sub() {
        double c;
        c = super.sum();
        return c - n;
    }
}
```

例 5.3 的程序运行结果如图 5.3 所示。

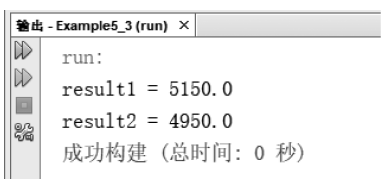


图 5.3　例 5.3 的程序运行结果

注意，如果将 Example5_3 类中的代码：

```
double result1 = classB.sum();
double result2 = classB.sub();
```

改写成(颠倒次序)：

```
double result2 = classB.sub();
double result1 = classB.sum();
```

那么运行结果如图 5.4 所示。

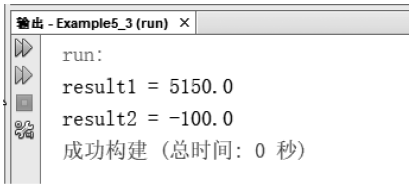


图 5.4　例 5.3 修改后的程序运行结果

这是因为执行 classB.sub()过程中需要执行 super.sum()，那么 super.sum()中出现的 n 是隐藏的 n，而 n 还没有赋值(默认值是 0)。

5.1.6　对象的上转型对象

我们经常说“老虎是哺乳动物”“狗是哺乳动物”等，若哺乳类是老虎类的父类，这样说当然正确，但是，当说老虎是哺乳动物时，老虎将失掉老虎独有的属性和功能。下面介绍对象的上转型对象。

假设 A 类是 B 类的父类，当用子类创建一个对象，并把这个对象的引用放到父类的对象中时，例如：

```
A a;
a = new B( );
```

或

```
A a;
B b = new B( );
a = b;
```

称对象 a 是对象 b 的上转型对象。

对象的上转型对象的实体是由子类负责创建的，但上转型对象会失去原对象的一些属性和功能(上转型对象相当于子类对象的一个“简化”对象)。上转型对象具有如下特点。

(1) 上转型对象不能操作子类新增的成员变量(失掉了这部分属性)；不能使用子类新增的方法(失掉了一些功能)。

(2) 上转型对象可以操作子类继承或隐藏成员变量，也可以使用子类继承的或重写的方法。上转型对象操作子类继承或重写的方法时，就是通知对应的子类对象去调用这些方法。因此，如果子类重写了父类的某个方法后，对象的上转型对象调用这个方法时，一定是调用了这个重写的方法。

(3) 可以将对象的上转型对象再强制转换到一个子类对象，这时，该子类对象又具备了子类所有属性和功能。

例 5.4 中，班长类 Monitor 是学生类 Student 的子类，而学生类 Student 是人类 Person 的子类，运行时 aStudent 对象是班长类 Monitor 创建的对象 joy 的上转型对象。注意：Person 类更通用一些，故放在 general 包中，Student、Monitor 以及 HaveLesson 类都与学校相关，故放在 school 包中，通过该例可进一步观察不同包下各类中成员的访问权限。运行

效果如图 5.4 所示。

例 5.4

Example5_4.java

例 5.4
视频讲解

```
public class Example5_4
{
    public static void main(String args[ ]) {
        Student mike = new Student("MIke", 12, 55);                    //Mike 是学生
        System.out.println(mike.getName());
        mike.doHomework();

        Monitor joy = new Monitor("Joy", 13, 5, "cleaning blackboard");  // Joy 是班长
        System.out.println(joy.getName());
        joy.doHomework();
        joy.onDuty();

        Student aStudent = joy;              //子类的对象能够赋值给父类的引用上转型对象
        aStudent.doHomework();               //上转型对象调用的仍然是子类覆盖后的方法
        //aStudent.onDuty();                 //上转型对象会丢失子类新增的方法
        if(aStudent instanceof Monitor) ((Monitor)aStudent).onDuty(); /* 上转型对象能够被
转换成子类的对象,进而恢复子类所丢失的方法 */

       //Monitor aMonitor = mike;            //父类的对象不能够赋值给子类的引用

        System.out.println("");

        HaveLesson.study(mike);
        HaveLesson.study(joy);}
}

public class Anthropoid {
    double m = 12.58;
    void crySpeak (String s) {
        System.out.println(s);
    }
}
```

Person.java

```
package example5_4.general;
public class Person
 {
    protected String name;
    protected int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = 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 void say(String something)
    {
        System.out.println(this.name + ":" + something);
    }

}
```

Student. java

```
package example5_4.school;
import example5_4.general.Person;
public class Student extends Person
{
    int num;
    public Student() {
    }
    public Student( String name, int age,int num) {
        super(name, age);
        this.num = num;
    }

    public int getNum() {
        return num;
    }

    public void setNum(int num) {
        this.num = num;
    }

    public void doHomework()
    {
        System.out.println(this.name + " is doing homework.");
    }
}
```

Monitor.java

```
package example5_4.school;
public class Monitor extends Student
{
    private String duty;
    public Monitor() {
    }

    public Monitor(String name, int age,int num, String duty ) {
        super( name, age,num);
        this.duty = duty;
    }

    public void onDuty()
    {
        System.out.println(this.name + " is " + duty);
    }

    @Override
    public void doHomework() {
        System.out.println(this.name + " do and collect homework.");
    }
}
```

HaveLesson.java

```
package example5_4.school;
public class HaveLesson
{
    public static void study(Student student)
    {
        student.doHomework();
    }
}
```

例 5.4 的程序运行结果如图 5.5 所示。

通过例 5.4 可观察出，上转型对象会丢失子类新增的属性和方法。如果子类重写了父类的某个方法后，上转型对象调用这个方法时，一定是调用了重写后的方法。故在 HaveLesson 类中的 study()方法的参数是 Student 类型，在通过 Student 类型调用 doHomework()方法时，由于其子类 Monitor 中进行了不同的重写，故 mike 和 joy 对象对于同样的 HaveLesson.study()方法的调用其运行结果是不同的，而这就是由继承性的特点所产生的多态！而在多态的应用中较多的是通过抽象类和接口进行实现。

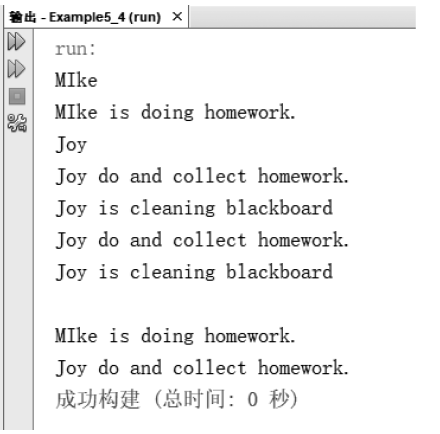


图 5.5　例 5.4 的程序运行结果

5.2 抽 象 类

5.2.1 抽象类的定义

用关键字 abstract 修饰的类称为 abstract 类(抽象类)。例如:

```
abstract class A {
    …
}
```

5.2.2 抽象类的实现

abstract 类中可以有 abstract 方法。和普通的类相比,abstract 类可以有 abstract 方法(抽象方法),也可以有非 abstract 方法。

下面 A 类中的 min( )方法是 abstract 方法,max( )方法是普通方法。

```
abstract class A {
    abstract int min(int x, int y);
    int max(int x, int y) {
        return x > y?x:y;
    }
}
```

abstract 类不能用 new 运算创建对象。对于 abstract 类,不能使用 new 运算符创建该类的对象。如果一个非抽象类是某个抽象类的子类,那么它必须重写父类的抽象方法,给出方法体,这就是为什么不允许使用 final 和 abstract 同时修饰一个方法的原因。

下面的例 5.5 使用了 abstract 类。

例 5.5

Example5_5.java

```
public class Example5_5 {
    public static void main(String args[ ]) {
         ClassB b = new ClassB();
        int sum = b.sum(30, 20);          //调用重写的方法
        int sub = b.sub(30, 20);          //调用继承的方法
        System.out.println("sum = " + sum);
        System.out.println("sub = " + sub);
    }
}
```

ClassA.java

```
public abstract class ClassA {
    abstract int sum(int x, int y);
    int sub(int x, int y) {
        return x - y;
    }
}
```

ClassB. java

```
public class ClassB extends ClassA
{
    int sum (int x, int y) {                    //子类必须重写父类的 sum()方法
        return x + y;
    }
}
```

例 5.5 的程序运行结果如图 5.6 所示。

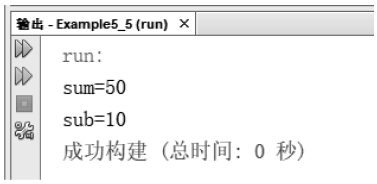


图 5.6　例 5.5 的程序运行结果

抽象类只关心操作，即只关心方法名字、类型以及参数，但不关心这些操作具体实现的细节，即不关心方法体。当在设计程序时，可以给出若干各抽象类表明程序的重要特征，也就是说，可以通过在一个抽象类中声明若干个抽象方法，表明这些方法的重要性。抽象类可以让程序设计者忽略具体的细节，以便更好地设计程序。例如，在设计地图时，首先考虑地图最重要的轮廓，不必去考虑诸如城市中的街道牌号等细节。细节应当由抽象类的非抽象子类去实现，这些子类可以给出具体的实例，来完成程序功能的具体实现。

5.2.3　抽象类与多态

在设计程序时，经常会使用 abstract 类，其原因是，abstract 类只关心操作，但不关心这些操作具体实现的细节，可以使程序的设计者把主要精力放在程序的设计上，而不必拘泥于细节的实现(将这些细节留给子类的设计者)，即避免设计者把大量的时间和精力花费于具体的算法上。在设计一个程序时，可以通过在 abstract 类中声明若干个 abstract 方法，表明这些方法在整个系统设计中的重要性，方法体的内容细节由它的 abstract 子类去完成。

使用多态进行程序设计的核心技术之一是使用上转型对象，即将 abstract 类声明对象作为其子类的上转型对象，那么这个上转型对象就可以调用子类重写的方法。

下面的例 5.6 中，准备设计一个动物饲养员，希望所设计的饲养员可以喂养各种不同的动物，而不同的动物吃的是不同的食物。

首先设计了一个抽象类 Animal，该抽象类有两个抽象方法 eat( )和 sleep( )，那么 Animal 的子类必须重写 eat( )和 sleep( )方法，即要求各种具体的动物给出自己吃的行为和睡的行为。

然后设计饲养员 Raiser 类，该类有一个 feed(Animal animal)方法，该方法的参数是 Animal 类型。显然，参数 animal 可以是抽象类 Animal 的任何一个子类对象的上转型对象，即参数 animal 可以调用 Animal 的子类重写的 eat( )方法执行具体动物吃的行为，如图 5.7 所示。

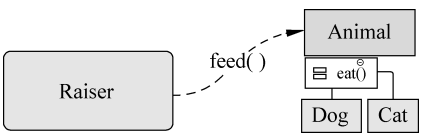


图 5.7　饲养员喂养动物的类关系图

该程序的代码如下。

例 5.6

Example5_6.java

例 5.6
视频讲解

```
public class Example5_6{
    public static void main(String args[ ]) {
         Cat cat = new Cat("Cat", "white");
        Dog dog = new Dog("Dog", "black");
        System.out.println(cat.getName());
        System.out.println(dog.getName());
        Raiser mike = new Raiser("Mike");
        mike.feed(cat);
        mike.feed(dog);}
}

public abstract class Animal
{
    String name;
    String color;
    public Animal(String name, String color) {
        this.name = name;
        this.color = color;
    }
    public Animal() {
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public abstract void eat();
    public abstract void sleep();
}

public class Cat extends Animal
{
    public Cat(String name, String color) {
        super(name, color);
    }
    public Cat() {
    }
    @Override
    public void eat() {
        System.out.println(this.name + " is eating fish.");
```

```
    }
    @Override
    public void sleep() {
        System.out.println(this.name + " is sleeping on the bed.");
    }
}

public class Dog extends Animal
{
    public Dog() {
    }
    public Dog(String name, String color) {
        super(name, color);
    }
    @Override
    public void eat() {
        System.out.println(this.name + " is gnawing bone.");
    }
    @Override
    public void sleep() {
        System.out.println(this.name + " is sleeping on the floor.");
    }
}

public class Raiser {
    private String name;
    public Raiser() {
    }
    public Raiser(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void feed(Animal animal)
    {
        animal.eat();
    }
}
```

例 5.6 的程序运行结果如图 5.8 所示。

从例 5.6 可以看出，饲养员 Mike 在调用方法 feed(Animal animal)方法时，对于不同的上转型对象 cat 和 dog 所运行的结果是不同的：猫吃鱼而狗啃骨头。利用多态可以通过通用的调用方式而产生不同的结果形态，使程序更加灵活。再比如我们要设计一款象棋游戏的程序。我们无法预知用户下一步要走哪一枚棋子，就只能在用户走棋时判断用户所使用棋子的类型(如车、马、炮)，再调用相应的走棋方法(如走直线、走日、翻山)，而且每次走棋都

要先判断棋子类型再调用走棋方法，非常烦琐，运行效率也很受影响。如果使用抽象类来实现多态的思想，可将棋子定义为抽象类，而走棋定义为抽象方法，由不同的棋子对走棋的抽象方法进行具体的实现，这样在下棋时，无论走哪一枚棋子只需调用走棋的方法就可以了，各种棋子的上转型对象自然会调用各自实现的具体方法，其类结构示意图为图 5.9。

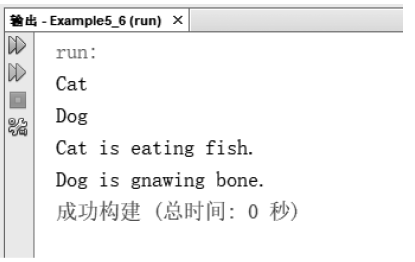


图 5.8　例 5.6 的程序运行结果

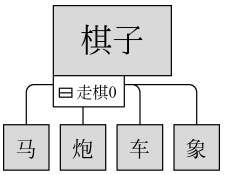


图 5.9　象棋棋子的类结构示意图

5.3　接　　口

5.3.1　接口的声明

1. 接口的声明

前面曾使用 class 关键字来声明类，接口通过使用关键字 interface 来声明。格式为：

```
interface 接口的名字
```

2. 接口体

接口体中包含常量定义和方法定义两部分。接口体中只进行方法的声明，不许提供方法的实现，所以，方法的定义没有方法体，且用分号“；”结尾。例如：

```
interface Printable {
    final int MAX = 100;
    void add( );
    float sum (float x, float y);
}
class A implements Printable,Addable
```

3. 接口的使用

一个类通过使用关键字 implements 声明自己实现一个或多个接口。如果实现多个接口，用逗号隔开接口名，如 A 类实现 Printable 和 Addable 接口：

```
class A implements Printable,Addable
```

如果一个类实现了某个接口，那么这个类必须重写该接口的所有方法。需要注意的是，重写接口的方法时，接口中的方法一定是 public abstract 方法，所以类在重写接口方法时不仅要去掉 abstract 修饰给出方法体，而且方法的访问权限一定要明显地用 public 来修饰。

实现接口的类一定要重写接口的方法，因此也称这个类实现了接口中的方法。

类重写的接口方法以及接口中的常量可以被类的对象调用，而且常量也可以用类名或接口名直接调用。

接口声明时，如果关键字 interface 前面加上 public 关键字，就称这样的接口是一个 public 接口。public 接口可以被任何一个类声明实现。如果一个接口不加 public 修饰，就称为友好接口类，友好接口可以被与该接口在同一包中的类声明实现。

如果父类实现了某个接口，那么子类也就自然实现了该接口，子类不必再显式地使用关键字 implements 声明实现这个接口。

接口也可以被继承，即可以通过关键字 extends 声明一个接口是另一个接口的子接口。由于接口中的方法和常量都是 public 的，子接口将继承父接口中的全部方法和常量。

5.3.2 理解接口

接口的语法规则很容易记住，但真正理解接口更重要。假如计算机有 USB 接口，实现了 USB 接口的设备都可以连接到计算机上进行工作，比如打印机、扫描仪、摄像头、键盘、鼠标、移动硬盘等，很大程度上提高了计算机的扩展性。接口的思想就在于它可以增加很多类都需要实现的功能，使用相同接口的类不一定有继承关系。同一个类也可以实现多个接口。接口只关心功能，并不关心功能的具体实现。

例 5.7
视频讲解

例 5.7

Example5_7.java

```
public class Example5_7 {
    public static void main(String[] args) {
        Printer hp = new Printer();
        USBStore seagate = new USBStore();
        Computer dell = new Computer();
        dell.workby(hp);
        dell.workby(seagate);

        Student mike = new Student();
        mike.turnOn(hp);
    }
}

public interface IUSB
{
    public void install();
    public void work();
}

public interface IPower
{
    public void start();
}

public class Printer implements IUSB,IPower
{
    @Override
```

```
    public void install() {
        System.out.println("安装打印机驱动程序");
    }
    @Override
    public void work() {
        System.out.println("打印机打印资料.");
    }
    @Override
    public void start() {
        System.out.println("打印机复印文件资料.");
    }
}

public class USBStore implements IUSB
{
    @Override
    public void install() {
        System.out.println("安装 U 盘驱动程序");
    }
    @Override
    public void work() {
        System.out.println("传输并保存文件");
    }
}

public class Computer
{
    public void workby(IUSB usb)
    {
        usb.install();
        usb.work();
    }
}

public class Student {
    public void turnOn(IPower power)
    {
        power.start();
    }
}
```

例 5.7 的程序运行结果如图 5.10 所示。

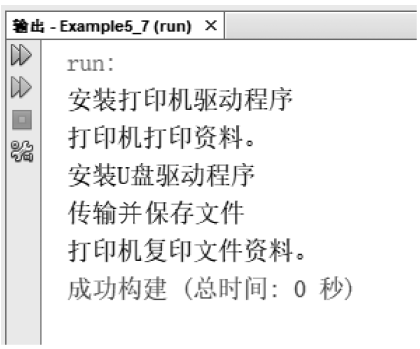


图 5.10　例 5.7 的程序运行结果

5.3.3　接口回调

接口回调是指：可以把实现某一接口的类创建的对象的引用赋给该接口声明的接口变量中。那么该接口变量就可以调用被类实现的接口中的方法。实际上，当接口变量调用被类实现的接口中的方法时，就是通知相应的对象调用接口的方法。

在下面的例 5.8 中，使用了接口的回调技术。

例 5.8

Example5_8.java

```
public class Example5_8 {

    public static void main(String[] args) {
        Desk desk = new Desk(10 , 20);
        Cake cake = new Cake(12 , 8);
        BasketBall basketBall = new BasketBall(10);

        Student mike = new Student("Mike");
        mike.calArea(desk);
        mike.calArea(cake);
        mike.calArea(basketBall);
    }
}

public interface IArea {
    public float PI = 3.1415926f;
    public String getType();
    public float getArea();
}

public class BasketBall implements IArea
{
    private float radius;
    private String type = "BasketBall";
    public BasketBall()
    {
    }
    public BasketBall(float radius)
    {
        this.radius = radius;
    }
    public float getRadius()
    {
        return radius;
    }
    public void setRadius(float radius)
    {
        this.radius = radius;
    }
    @Override
    public String getType() {
        return type;
    }
    @Override
    public float getArea()
    {
        return this.radius * this.radius * IArea.PI;
```

```
    }
}

public class Cake implements IArea
{
    private float width;
    private float height;
    private String type = "Cake";
    public Cake()
    {
    }
    public Cake(float width, float height)
    {
        this.width = width;
        this.height = height;
    }
    public float getWidth()
    {
        return width;
    }
    public void setWidth(float width)
    {
        this.width = width;
    }
    public float getHeight()
    {
        return height;
    }
    public void setHeight(float height)
    {
        this.height = height;
    }
    @Override
    public String getType() {
        return type;
    }
    @Override
    public float getArea()
    {
        return this.width * this.height/2;
    }
}

public class Desk implements IArea
{
    private float length;
    private float width;
    private String type = "Desk";
    public Desk()
    {
    }
```

```
    public Desk(float length, float width)
    {
        this.length = length;
        this.width = width;
    }
    public float getLength()
    {
        return length;
    }
    public void setLength(float length)
    {
        this.length = length;
    }
    public float getWidth()
    {
        return width;
    }
    public void setWidth(float width)
    {
        this.width = width;
    }
    @Override
    public String getType() {
        return type;
    }
    @Override
    public float getArea()
    {
        return this.length * this.width;
    }
}

public class Student {
    private String name;
    public Student() {
    }
    public Student(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void calArea(IArea area)
    {
        System.out.println("Area of " + area.getType() + " is " + area.getArea());
    }
}
```

例 5.8 的程序运行结果如图 5.11 所示。

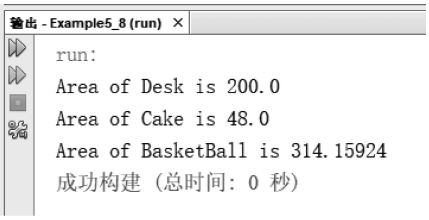


图 5.11　例 5.8 的程序运行结果

5.3.4　接口与多态

5.3.3 节学习了接口回调，即当把实现接口的类的实例的引用赋值给接口变量后，该接口变量就可以回调类重写的接口方法。由接口产生的多态就是指不同的类在实现同一个接口时可能具有不同的实现方式，那么接口变量在回调接口方法时就可能具有多种形态。

在设计程序时，经常会使用接口，其原因是接口只关心操作，但不关心这些操作的具体实现细节，可以使我们把主要精力放在程序的设计上，而不必拘泥于细节的实现。也就是说，可以通过在接口中声明若干个 abstract 方法，表明这些方法的重要性，方法体的内容细节由实现接口的类去完成。使用接口进行程序设计的核心思想是使用接口回调，即接口变量存放实现该接口的类的对象的引用，从而接口变量就可以回调类实现的接口方法。

下面的例 5.9 中，准备设计一个广告牌，希望所设计的广告牌可以展示许多公司的广告词。

首先设计了一个接口 Advertisement，该接口有两个方法：showAdvertisement()和 getCorpName()，那么实现 Advertisement 接口的类必须重写 showAdvertisement()和 getCorpName()方法，即要求各个公司给出具体的广告词和公司的名称。

然后设计 AdvertisementBoard 类(广告牌)，该类有一个 show(Advertisement adver)方法，该方法的参数 adver 是 Advertisement 接口类型(就像人们常说的，广告牌对外留有接口)。显然，该参数 adver 可以存放任何实现 Advertisement 接口的类的对象的引用，并回调类重写的接口方法：showAdvertisement()显示公司的广告词、回调类重写的接口方法；getCorpName()显示公司的名称。

详细代码如下。

例 5.9

Example5_9.java

```
public class Example5_9{
    public static void main(String agrs[ ]) {
        AdvertisementBoard board = new AdvertisementBoard( );
        board.show(new PhilipsCorp( ));
        board.show(new LenovoCorp( ));
    }
}

public interface Advertisement {              //接口
    public void showAdvertisement( );
```

```
    public String getCorpName( );
}

public class AdvertisementBoard {              //负责创建广告牌
    public void show(Advertisement adver) {
        System.out.println("广告牌显示" + adver.getCorpName( ) + "公司的广告词: ");
        adver.showAdvertisement( );
    }
}

public class PhilipsCorp implements Advertisement {    //PhilipsCorp 实现 Advertisement 接口
    public void showAdvertisement( ) {
        System.out.println("@@@@@@@@@@@@@@@@@@@@");
        System.out.println("没有最好,只有更好");
        System.out.println("@@@@@@@@@@@@@@@@@@@@");
    }
    public String getCorpName( ) {
        return "飞利浦";
    }
}

public class LenovoCorp implements Advertisement { //LenovoCorp 实现 Advertisement 接口
    public void showAdvertisement( ) {
        System.out.println("********************************");
        System.out.println("让世界变得很小");
        System.out.println("********************************");
    }
    public String getCorpName( ) {
        return "联想集团";
    }
}
```

例 5.9 的程序运行结果如图 5.12 所示。

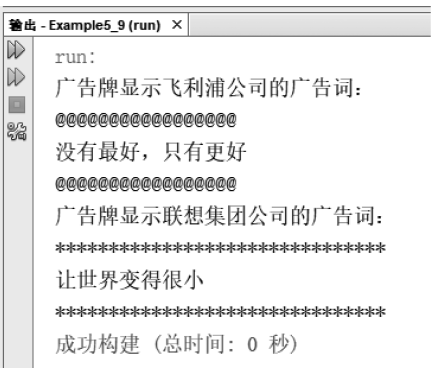


图 5.12　例 5.9 的程序运行结果

接口的用处具体体现在下面几方面。

(1) 通过接口实现不相关类的相同行为,而无须考虑这些类之间的关系。

(2) 通过接口指明多个类需要实现的方法。

(3) 通过接口了解对象的交互界面,而无须了解对象所对应的类。

5.3.5　抽象类与接口的比较

接口和抽象类的比较如下。

(1) 抽象类和接口都可以有 abstract 方法。

(2) 接口中只可以有常量,不能有变量；而抽象类中既可以有常量也可以有变量。

(3) 抽象类中也可以有非 abstract 方法,接口不可以。

在设计程序时应当根据具体的分析来确定是使用抽象类还是接口。抽象类除了提供重要的需要子类去实现的 abstract 方法外,也提供了子类可以继承的变量和非 abstract 方法。

如果某个问题需要使用继承才能更好地解决，比如，子类除了需要实现父类的 abstract 方法外，还需要从父类继承一些变量或继承一些重要的非 abstract 方法，就可以考虑用 abstract 类。如果某个问题不需要继承，只是需要若干个类给出某些重要的 abstract 方法的实现细节，就可以考虑使用接口。

5.4 应用实例：POS 刷卡机

应用实例
视频讲解

在日常生活中商家为了促销往往会为消费者办理各种充值卡，不同的充值卡其打折的力度也不尽相同。模拟充值卡消费，POS 刷卡机能够按照相应的充值卡进行消费扣款，并且商家可以自定义多种不同类型的充值卡。这样的问题使用面向对象的多态就能够非常灵活地加以解决。比如可以设定一个抽象类 Card，含有充值卡的一些基本属性，如卡类型和卡内金额，并且含有一个抽象方法来计算打折后的实际应付金额。对于不同的充值卡可继承该抽象类，对于不同的打折策略进行具体计算应付金额，当需要增加新的充值卡类型时只需在 Card 类下再派生出一个新类就可以了，类之间的关系如图 5.13 所示。

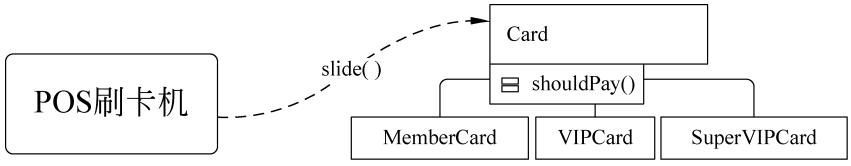


图 5.13 POS 刷卡机的类关系示意图

新建项目 **MemberCardDemoApp**，其类结构如图 5.14 所示。

在 cardpackage 包下是有关各种充值卡的类，其中，Card 是一个抽象类作为基类，下面分别派生出了会员卡（MemberCard 打 9 折）、VIP 卡（VIPCard 打 7 折）和超级 VIP 卡（SuperVIPCard 打 5 折）；pospackage 包下面是 POS 刷卡机类（POS）和 POS 刷卡机的界面类（POSView），其中，POS 刷卡机类中包含一个刷卡的方法 public static boolean slide (Card card，float totalPrice)，参数 card 为要刷的卡，totalPrice 为总的消费金额。在刷卡时会调用相应卡的消费方法，先计算应付金额，并与卡内金额相比较，卡内金额充足则扣款消费返回消费成功 true，否则返回消费失败 false。

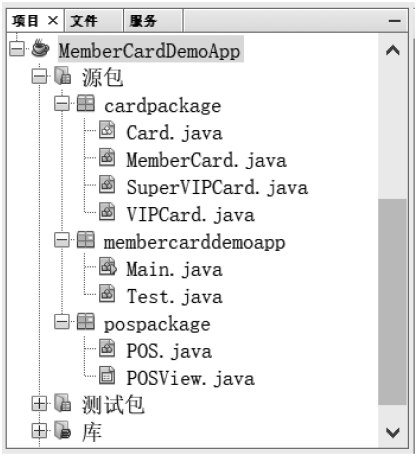


图 5.14 POS 刷卡机的类结构示意图

其主要类的实现代码如下。

```
package cardpackage;
public abstract class Card {
    String type;
    float amount;
    public Card() {
```

```
    }
    public Card(String type) {
        this.type = type;
    }
    public Card(String type, int amount) {
        this.type = type;
        this.amount = amount;
    }
    /**
     * @return the type
     */
    public String getType() {
        return type;
    }
    public boolean comsume(float totalPrice) {
        if (this.amount >= this.shouldPay(totalPrice)) {
            this.amount = this.amount - this.shouldPay(totalPrice);
            return true;
        } else {
            return false;
        }
    }
    public abstract float shouldPay(float totalPrice);
    /**
     * @return the amount
     */
    public float getAmount() {
        return amount;
    }
    /**
     * @param amount the amount to set
     */
    public void setAmount(float amount) {
        this.amount = amount;
    }
}
```

```
package cardpackage;
public class MemberCard extends Card
{
    public MemberCard()
    {
        this.type = "MemberCard";
    }
    public MemberCard(String type, int amount)
    {
        super(type, amount);
    }
    @Override
    public float shouldPay(float totalPrice)
    {
        return totalPrice * 0.9f;
    }
}
```

```
package cardpackage;
public class VIPCard extends Card
{
    public VIPCard()
    {
        this.type = "VIPCard";
    }
    public VIPCard(String type, int amount)
    {
        super(type, amount);
    }
    @Override
    public float shouldPay(float totalPrice)
    {
        return totalPrice * 0.7f;
    }
}
```

```
package cardpackage;
public class SuperVIPCard extends Card
{
    public SuperVIPCard()
    {
        this.type = "SuperVIPCard";
    }
    public SuperVIPCard(String type, int amount)
    {
        super(type, amount);
    }
    @Override
    public float shouldPay(float totalPrice)
    {
        return totalPrice * 0.5f;
    }
}
```

```
package pospackage;
import cardpackage.Card;
public class POS {
    public static boolean slide(Card card,float totalPrice)
    {
        return card.comsume(totalPrice);
    }
}
```

在 membercarddemoapp 包中创建了一个测试类，在类中分别创建了三种卡的对象，其中，会员卡（memberCard）充值 500 元，VIP 卡（vipCard）充值 1000 元，超级 VIP 卡（superVIPCard）充值 2000 元。POS 刷卡机的程序运行结果如图 5.15 所示。

如果选择会员卡消费 100 元，单击“刷卡消费”按钮后，结果是打 9 折，500 元的会员卡扣除 90 元，余额为 410 元，如图 5.16 所示。

图 5.15　POS 刷卡机的程序运行结果

图 5.16　使用会员卡消费 100 元

如果选择超级 VIP 卡消费 1000 元，单击“刷卡消费”按钮后，结果是打 5 折，2000 元的会员卡扣除 500 元，余额为 1500 元，如图 5.17 所示。

图 5.17　使用超级 VIP 卡消费 1000 元

小　　结

（1）继承是一种由已有的类创建新类的机制。利用继承可以先创建一个共有属性的一般类，根据该一般类再创建具有特殊属性的新类。

（2）所谓子类继承父类的成员变量作为自己的一个成员变量，就好像它们是在子类中直接声明一样，可以被子类中自己声明的任何实例方法操作。

（3）所谓子类继承父类的方法作为子类中的一个方法，就像它们是在子类中直接声明一样，可以被子类中自己声明的任何实例方法调用。

（4）多态是面向对象编程的又一重要特性。子类可以体现多态，即子类可以根据各自的需要重写父类的某个方法，子类通过方法的重写可以把父类的状态和行为改变为自身的状态和行为。接口也可以体现多态，即不同的类在实现同一接口时，可以给出不同的实现手段。

习　　题

1. 子类将继承父类的哪些成员变量和方法？子类在什么情况下隐藏父类的成员变量和方法？

2. 什么叫对象的上转型对象？

3. 什么叫接口的回调？

4. 请给出下面程序的输出结果。

```
class A {
    double f(double x, double y) {
        return x + y;
    }
}
class B extends A {
    double f(int x, int y) {
        return x * y;
    }
}
public class E {
    public static void main(String args[ ]) {
        B b = new B( );
        System.out.println(b.f(3,5));
        System.out.println(b.f(3.0,5.0));
    }
}
```

5. 请给出下面程序中 E 类运行的结果。

```
class A {
    double f(double x, double y) {
        return x + y;
```

```
    }
    static int g(int n) {
        return n * n;
    }
}
class B extends A {
    double f(double x, double y) {
        double m = super.f(x,y);
        return m + x * y;
    }
    static int g(int n) {
        int m = A.g(n);
        return m + n;
    }
}
public class E {
    public static void main(String args[ ]) {
        B b = new B( );
        System.out.println(b.f(10.0,8.0));
        System.out.println(b.g(3));
    }
}
```

6. 需要为一个景区实现计算景区门票的程序，已知成年人的门票价格是 100 元，儿童票打 3 折，老年票打 5 折。使用抽象类来为任意多张不同类型的票计算总价。其 UML 类图如图 5.18 所示。

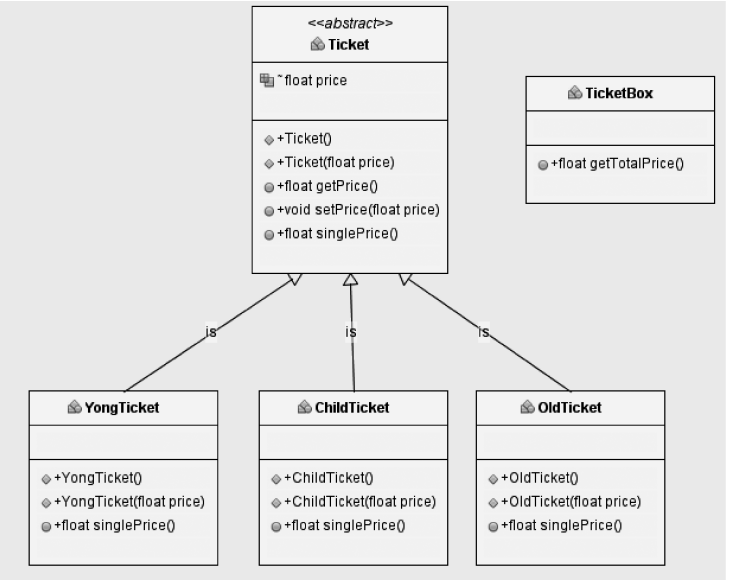


图 5.18　UML 类图