第5章 继承与多态


视频讲解

本章讨论面向对象程序设计另外两个最重要的特点：继承与多态。

继承是面向对象程序设计方法中实现软件重用的一种重要手段，通过继承可以更有效地组织程序结构，明确类之间的关系，并充分利用已有的类来创建新类，以完成更复杂的设计和开发。多态则可以统一多个相关类的对外接口（方法调用），并在运行时根据不同的情况执行不同的操作，提高类的抽象度和灵活性。

5.1 子类、父类与继承机制

5.1.1 继承的概念

在面向对象技术中，继承是最为显著的一个特征。继承表示存在于面向对象程序中的两个类之间的一种关系。当一个类充当另一个类的子类，就自动拥有另一个类的所有非私有属性（域和方法）时，就称这两个类之间具有继承关系。被继承的类称为父类（或超类），继承了父类的类称为子类。

在类的定义过程中，继承是一种由已有的类创建新类的机制。子类不仅可从父类中继承域和方法，而且可重定义及扩充新的内容。一般方法是：创建一个共有属性的父类，再创建具有特殊属性的子类。例如，将平面上的坐标点构成一个点类，则平面上的圆类可继承点类的所有属性，并以继承的坐标点为圆心，再自定义一个域为半径，完成圆上的各种操作，如图 5.1 所示。

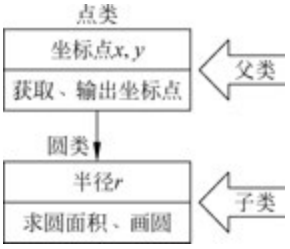


图 5.1 “圆”类继承“点”类图示

一个父类可以同时拥有多个子类，此时该父类实际上是所有子类的公共域和方法的集合，而每个子类则是对公共域和方法在功能、内涵方面的扩展和延伸。

父类、子类间的关系具有以下特性。

- 共享性，即子类可以共享父类的公共域和方法。
- 差异性，即子类和父类一定会存在某些差异，否则就应该是同一个类。
- 层次性，即由 Java 规定的单继承性，每个类都处于继承关系中的某个层面。

以电话为例，图 5.2 列举了各种电话类的层次结构以及这些类的域和方法。

从图 5.2 中可以看出，面向对象的程序设计中这种继承关系符合人们通常的思维方式。它描述了一种分类关系：电话分为固定电话和移动电话两大类，固定电话可分为 IP 电话和普通电话等。其中，抽象的电话类是所有电话类的父类，它包含所有电话的公共属性。这些公共属性包括话费余额等数据属性，以及计费方式、查询余额等行为属性。将电话分类并具体化，

就分别派生出两个子类：固定电话和移动电话。这两个子类一方面继承了父类电话的所有属性，即也拥有话费余额、计费方式、查询余额等属性；另一方面又专门定义了适合本身特殊需要的属性，如对于固定电话，应该有座机费属性，这些属性对移动电话不适合(这里设移动电话没有座机费)。从固定电话到 IP 电话和普通电话的继承遵循相同的原则。

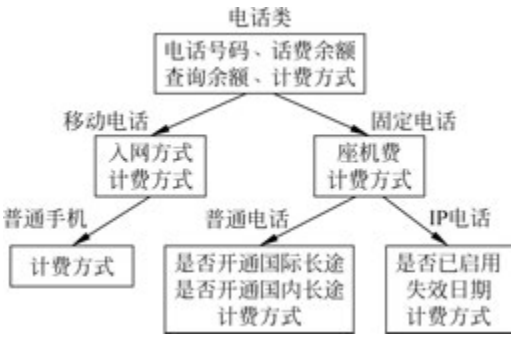


图 5.2　电话及其子类的继承关系

采用面向对象的设计方法，可以仅在抽象的父类——电话类——中定义电话号码、话费余额等公共属性，其多个子类则从父类那里继承这些属性。当公共属性发生修改时，只需要在父类中修改一次即可，不但维护工作量可以大大减少，而且可以避免修改遗漏引起的不一致性。

继承的主要优点如下。

- 程序结构清晰。
- 编程量减少。
- 易于修改和维护。

一般来说，面向对象程序设计语言中的继承分为单继承和多继承。单继承采用树状结构，设计、实现容易；而多继承采用网状结构，设计、实现较复杂。

Java 语言出于安全性和可靠性的考虑，仅提供了单继承机制，即 Java 程序中的每个类只允许有一个直接的父类，而 Java 多继承的功能则是通过接口机制来实现的。

5.1.2　类的层次

Java 语言中类的单继承机制使得 Java 的类具有严格的层次结构。除 Object 类之外，每个类都有唯一的父类。这种性质使得类的层次结构形成了如图 5.3 所示的一种树状结构。Object 类定义和实现了 Java 系统所需要的众多类的共同行为，它是所有类的父类，也即这个树状结构中的根类，所有的类都是由这个类继承、扩充而来的。这个 Object 类定义在 java.lang 包中。

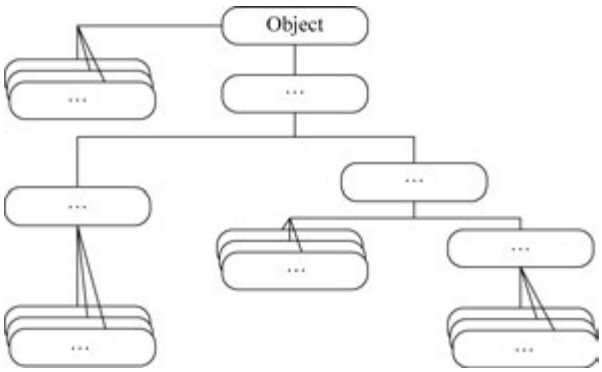


图 5.3　Java 语言中类的层次

5.2 继承的实现

本节讨论 Java 如何实现其继承性以及继承机制中的一些细节问题。

5.2.1 子类的定义

定义一个子类，即在定义一个类的时候加上 extends 关键字，并在之后带上其父类名，其一般格式为：

```
[类的修饰符] class  <子类名> extends <父类名>{
        <域定义>;
        <方法定义>;
}
```

这和前面定义类的格式并没有什么区别，只是“extends <父类名>”不再是可选项。

【例 5.1】 图 5.1 指出圆类继承点类，即先定义一个点类 Point，然后由点类派生一个圆类 Circle。其实现代码为：

```
class Point {
    int x, y;
    void setXY(int i, int j) {
        x = i; y = j;
    }
}
class Circle extends Point {
    double r;
    double area(){
        return 3.14 * r * r;
    }
}
```

在定义子类时用 extends 关键字指明新定义类的父类，就在两个类之间建立了继承关系。新定义的子类可以从父类那里继承所有非 private 的域和方法作为自己的属性。

【例 5.2】 下面的程序实现图 5.2 中电话类的继承结构。为了代码简洁，而把注意力集中在要讨论的语法机制上，这里假设移动电话仅一种，电话的计费方式简化为：国内长途话费是市话费的 3 倍，国际长途话费是市话费的 9 倍。

```
import java.util. * ;
abstract class Telephone{
    long phoneNumber;
    final static double local_Call = 1;
    final static double distance_Call = 3;                    //国内长途
    final static double international_Call = 9;               //国际长途
    double balance;
    abstract boolean charge_Mode(double call_Mode);           //抽象方法
    double getBalance() {
        return balance;
    }
}
class Mobile_Phone extends Telephone{
    String networkType;                                       //入网类型
    String getType() {
```

```
        return networkType;
    }
    boolean charge_Mode(double call_Mode) {
       if (balance > 0.6) {
          balance -= 0.6;                                    //移动电话为每分钟 0.6 元
          return true;                                       //可以打电话了
       } else return false;
    }

}

abstract class Fixed_Telephone extends Telephone{            //固定电话
       double monthFee;
}

class Ordinary_phone extends Fixed_Telephone {
    boolean longdistanceService;                             //国内长途服务是否开通
    boolean internationalService;                            //国际长途服务是否开通

    boolean charge_Mode(double call_Mode) {                  //call_Mode 为 1、3、9
       if (call_Mode == distance_Call
             &&!longdistanceService)return false;
                          //检查国内长途服务是否开通
       if (call_Mode == international_Call &&!internationalService)
           return false;
                          //检查国际长途服务是否开通
       if (balance > (0.2 * call_Mode)) {                    //0.2 是市话费
           balance -= (0.2 * call_Mode);                     //市话费乘以相应的倍数
           return true;
       } else  return  false;
    }
}

class IP_Phone extends Fixed_Telephone {                     //IP 电话
        boolean started;
        Calendar expireDate;
                          //Calendar 是系统类,其对象代表一个具体的日期
        boolean charge_Mode(double call_Mode) {

         if(!started) {
              started = true;
              expireDate = new GregorianCalendar();
                          //获得当前日期
              expireDate.set(Calendar.MONTH,expireDate.get(Calendar.MONTH) + 6);
                          //设置 6 个月后过期
          };
          if (balance > 0.3
              && expireDate.after(new GregorianCalendar())){
           //new GregorianCalendar()创建一个包含当前日期的 Calendar 类的对象
           //after()方法是 Calendar 类的方法,expireDate 在当前日期之后时
           //expireDate.after(new GregorianCalendar())返回 true;
                if (call_Mode > local_Call){
                    balance -= (0.1 * call_Mode + 0.2);
                       //用 IP 电话打国际长途和国内长途
                       //其中 0.2 是市话费
```

```
                return true;
            } else
                return false;                          //假定 IP 电话只打长途
        }else
            return false;
    }
}
```

例 5.2 定义了 5 个类，其中 Mobile_Phone 类和 Fixed_Phone 类是 Telephone 类派生的子类；IP_Phone 类和 Ordinary_phone 类是 Fixed_Phone 类派生的子类。

在这个程序中，只在 Telephone 类中定义了 phoneNumber、local_Call、distance_Call、international_Call、balance 这些域，但是在 Mobile_Phone、IP_Phone、Ordinary_phone 类中使用了 balance 域和其他域，它们使用的这些域都是从父类 Telephone 继承的。

另外，Telephone 类还定义了一个抽象方法 charge_Mode()，由于它的子类 Fixed_Telephone 也是抽象类，可以不实现这个抽象方法。而其他派生的 3 个电话类不是抽象类，因此，分别根据自己的具体情况来实现 charge_Mode()方法。例如，普通电话类 Ordinary_phone 的 charge_Mode()方法要检查国内长途和国际长途服务是否开通，然后按通话类别计费。IP_Phone 类的 charge_Mode()方法定义的话费最低，但是要加上市话费，并且必须在失效日期之前拨打电话。而且，这些非抽象类必须实现 charge_Mode()方法，否则编译时会出现错误。

现在检验一下这些类的定义是否可行，为此建立 IP 电话的测试类。在该类中建立一个 IP 电话的对象，再预置 50 元话费；然后分别将 Telephone.international_Call 和 Telephone.distance_Call 作为参数调用 charge_Mode()方法，表示分别用 IP 电话打 1 分钟国际长途和用 IP 电话打 1 分钟国内长途。IP 电话的测试类的代码如下。

```
class IP_PhoneTest
{
    public static void main(String[] args)
    {
       IP_Phone ipp = new IP_Phone();
        ipp.balance = 50;
        ipp.charge_Mode(Telephone.international_Call);
                        //用 IP 电话打 1 分钟国际长途
        System.out.println("当前电话费的余额为" + ipp.balance);
        ipp.charge_Mode(Telephone.distance_Call);
                        //用 IP 电话打 1 分钟国内长途
        System.out.println("当前电话费的余额为" + ipp.balance);
    }
}
```

该测试类的执行结果如图 5.4 所示。

图 5.4　IP 电话的测试类的执行结果

5.2.2 域的继承与隐藏

1. 域的继承

子类可以继承父类的所有非私有域。例如,各类电话类所包含的域分别如下所示。

1) Telephone 类

```
long phoneNumber;
final static double local_Call;
final static double distance_Call;
final static double international_Call
double balance;                          //5 个基本域
```

2) Mobile_Phone 类

```
long phoneNumber;
      …
double balance;                          //来自继承父类 Telephone 的 5 个基本域
String  networkType;
```

3) Fixed_Phone 类

```
long phoneNumber;
      …
double balance;                          //来自继承父类 Telephone 的 5 个基本域
double monthFee;
```

4) IP_Phone 类

```
long phoneNumber;
      …
double balance;                          //来自继承父类 Fixed_Telephone 的 5 个基本域
double monthFee;                         //来自继承父类 Fixed_Telephone
boolearn started;
Date expireDate;
```

5) Ordinary_phone 类

```
long phoneNumber;
      …
double balance;                          //来自继承父类 Fixed_Telephone 的 5 个基本域
double monthFee;                         //来自继承父类 Fixed_Telephone
boolean longdistanceService;
boolean internationalService;
```

可见父类所有的非私有域实际是各子类都拥有的域。子类从父类继承域而不需要重复定义父类的域,其好处是简化程序,降低维护的工作量。

2. 域的隐藏

上面提到子类可以从父类继承域而不需要重复定义父类的域,子类还可以重新定义一个与从父类继承来的域变量完全相同的变量,这种方式称为域的隐藏。即子类中定义了与父类同名的域变量,就是子类变量对同名父类变量的隐藏。这里的隐藏是指子类拥有了两个相同名字的变量,一个来自继承父类,另一个由自己定义。在这种情况下,当子类执行继承的父类方法时,处理的是父类的变量,而当子类执行它自己定义的方法时,操作的就是它自定义的变量,而把来自继承父类的变量“隐藏”起来了。

【例 5.3】 下面是一个测试域的隐藏的程序，其类层次结构如图 5.5 所示。

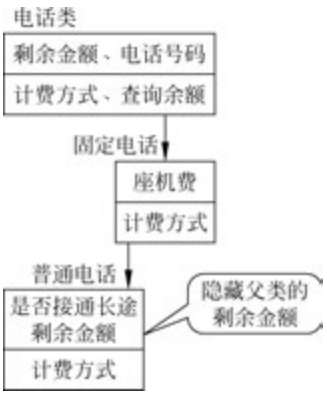


图 5.5 含有域的隐藏的类层次图

实现图 5.5 中含有域的隐藏的类层次图的代码如下。

```
//HiddenFieldExample.java
abstract class Telephone{
    long phoneNumber;
    final static double local_Call = 1;
    final static double distance_Call = 3;
    final static double international_Call = 9;
    double balance;
    abstract boolean charge_Mode (double call_Mode);
    double getBalance() {
        return balance;
    }
}
class Mobile_Phone extends Telephone{
    String networkType;
    String getType() {
        return networkType;
    }
        boolean charge_Mode(double call_Mode) {
            if (balance > 0.6) {
                balance -= 0.6;
                return true;
            } else
                return false;
        }
}

abstract class Fixed_Telephone extends Telephone{
    double monthFee;
}

class Ordinary_phone extends Fixed_Telephone{
    boolean longdistanceService;
    boolean internationalService;
    double  balance;                        //隐藏父类的 balance
    boolean charge_Mode(double call_Mode) {
        if(call_Mode == distance_Call &&! longdistanceService)
            return false;
          if(call_Mode == international_Call &&! internationalService)
              return false;
                                            //检查国内长途和国际长途服务是否开通
          if(balance > (0.2 * call_Mode)) {
              balance -= (0.2 * call_Mode);
              return true;
          } else
              return  false;
    }
}

public class HiddenFieldExample{            //主类的定义
    public static void main(String args[]){
        Ordinary_phone  myfamilyphone = new Ordinary_phone();
        myfamilyphone.internationalService = true;
               //对象 myfamilyphone 有两个 balance 变量,
```

```
            //一个来自继承父类,另一个是自定义的
        myfamilyphone.balance = 50.0;
            //为 myfamilyphone 自定义的 balance 变量赋值
        System.out.println("父类被隐藏的金额为:"
                           + myfamilyphone.getBalance());
            //getBalance()方法返回的是 myfamilyphone 对象,
            //来自继承父类的 balance 的数值(没被赋值),其默认值是 0.0
        if (myfamilyphone.charge_Mode(Telephone.international_Call))
           //调用 myfamilyphone 对象的 charge_Mode()方法,
           //修改 myfamilyphone 对象自身的 balance 变量
         System.out.println("子类的剩余金额为: " + myfamilyphone.balance);
           //输出修改后 myfamilyphone 对象自定义的 balance 变量值
    }
}
```

这个程序的执行结果如图 5.6 所示。

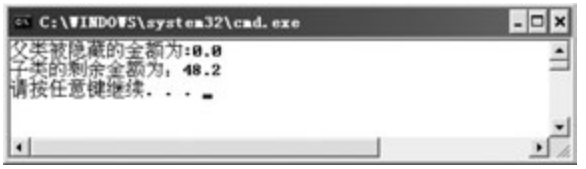


图 5.6　域的隐藏的测试结果

由于该程序在 Ordinary_phone 类中增加定义了一个与从父类那里继承来的变量完全相同的 balance 变量。这样修改后,Ordinary_phone 类中的域变为:

```
long phoneNumber;
    …
double balance;                          //来自继承父类 Fixed_Telephone 的 5 个基本域
double balance;                          //Ordinary_phone 类自定义的域
double monthFee;                         //来自继承父类 Fixed_Telephone
boolean longdistanceService;             //Ordinary_phone 类自定义的域
boolean internationalService;            //Ordinary_phone 类自定义的域
```

此时子类有两个同名变量 balance,一个来自继承父类,另一个由自己定义,当执行不同操作时,处理不同的变量。

5.2.3　方法的继承与覆盖

1. 方法的继承

父类的非私有方法作为类的非私有成员,可以被子类所继承。根据方法的继承关系,将例 5.3 中的电话类及其各子类所包含的方法列举如下。

1) Telephone 类

```
abstract boolean charge_Mode();
double getBalance();
```

2) Fixed_Telephone 类

```
abstract boolean charge_Mode();          //来自继承父类 Telephone
double getBalance();                     //来自继承父类 Telephone
```

3) Ordinary_phone 类

```
double   getBelance();                   //来自继承父类 Fixed_Telephone
boolean  charge_Mode();
```

各个类的对象可以自由使用从父类继承的方法。

2. 方法的覆盖

方法的覆盖(Override)是指子类重新定义从父类继承的同名方法,此时父类的这个方法在子类中将不复存在。这是子类通过重新定义与父类同名的方法,实现自身的行为。

【例 5.4】 方法的覆盖程序举例。

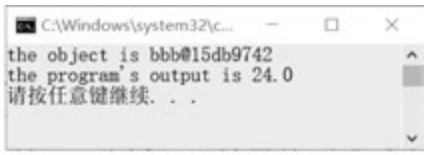
```
class aaa{
    double f(double x,double y)
    {
        return x * y;
    }
}

class bbb extends aaa{
//假如将这里的 f 方法注释掉
//就没有产生覆盖,程序执行的结果就不一样
    double f(double x, double y)
    {
        return x + y;
    }
}

public class OverrideExample{
    public static void main(String[ ] args)
    {
        bbb obj = new bbb();
        System.out.println("the object is " + obj.toString());
        System.out.println("the program's output is " + obj.f(4,6));
    }
}
```

在这个程序的 bbb 类中,由于覆盖了父类的方法,bbb 类的对象 obj 调用的是自定义的方法,其执行结果如图 5.7(a)所示。假如在这个程序的 bbb 类中将 f 方法注释掉,这个程序仍然可以执行,这时候的执行结果如图 5.7(b)所示。分析一下结果就不难看出它们各自执行的是哪个类的 f 方法。

(a) 覆盖了父类的方法



(b) 未覆盖了父类的方法

图 5.7 方法的覆盖

方法的覆盖中需要注意的问题是,子类在重新定义父类已有的方法时,应保持与父类完全相同的方法头部声明,即应保持与父类有完全相同的方法名、返回类型和参数列表,否则就不是方法的覆盖,而是子类定义了自己的、与父类无关的方法,父类的方法仍然存在于子类中。

方法的覆盖与域的隐藏的不同之处在于,子类隐藏父类的域只是使之不可见,父类的同名域在子类对象中仍然占有自己独立的内存空间;而子类方法对父类同名方法的覆盖将清除父类方法占用的内存空间,从而使父类方法在子类对象中不复存在。

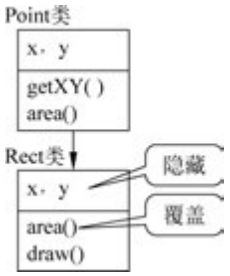


图 5.8　同时含有隐藏与覆盖的示意说明

域的隐藏和方法的覆盖的意义在于：通过隐藏域和覆盖方法可以把父类的状态和行为改为自身的状态和行为，对外统一名字与接口，又不失其继承性。

图 5.8 进一步给出了同时含有域的隐藏和方法的覆盖的示意说明。假设在主类中，建立 Rect 类的对象 c，若有调用：

(1) c.area()，这是在使用 Rect 类的 area()操作自己的 x、y，求矩形面积。

(2) c.getXY()，这是在使用继承父类 Point 的 getXY()处理所继承父类的 x、y。

5.3　多　态　性

多态性是面向对象程序设计的又一个重要的技术和手段。多态性是指同名的不同方法在程序中共存，即为同一个方法定义几个版本，运行时根据不同情况执行不同的版本。调用者只需使用同一个方法名，系统会根据不同情况，调用相应的不同方法，从而实现不同的功能。多态性又被称为“一个名字，多个方法”。

5.3.1　多态性的概念

在面向过程的程序设计中，各函数是不能重名的，否则在用名字调用时就会产生歧义和错误。而在面向对象的程序设计中，有时却需要利用这样的“重名”现象来提高程序的抽象度和简洁性。

考查图 5.2 中电话类的层次树，计费方式是所有电话都具有的操作，但不同的电话计费方式操作的具体实现是不同的。如果不允许这些目标和功能相同的操作使用同样的方法名字，就必须分别用多个不同名字的方法定义普通电话、IP 电话和移动电话类的计费方式。这样使用起来不方便，要记忆和区分多个方法的名字，继承的优势就不明显了。在面向对象的程序设计中，为了解决这个问题，引入了多态性的概念。

多态性的实现有以下两种。

1) 覆盖实现多态性

通过子类对继承父类方法的重定义来实现。使用时注意，在子类重定义父类方法时，要求与父类中方法的原型(参数个数、类型、顺序)完全相同。

2) 重载实现多态性

通过定义同一个类中的多个同名的不同方法来实现。编译时是根据参数(个数、类型、顺序)的不同来区分不同方法的。

5.3.2　覆盖实现多态性

第一种多态，即通过子类对父类方法的覆盖实现的多态。以图 5.2 中各类电话为例，Telephone 类有一个各子类共有的方法 charge_Mode()代表计费方式的功能，根据继承的特点，Telephone 的每个子类都将继承这个方法。但是，该方法的功能在不同种类的电话中其具体实现是不同的。因此，不同的子类可以重新定义、编写 charge_Mode()的内容，以实现特定计费方式的方法。在这些子类中，虽然实现的方式不同，但却共享同一个方法名——charge_

Mode()。

既然在覆盖实现多态性的方式中，子类重定义父类方法时，方法的名字、参数个数、类型、顺序完全相同，那么如何区别这些同名的不同方法呢？由于这些方法存在于一个类层次结构的不同类中，在调用方法时只需要指明调用哪个类（或对象）的方法，就很容易选择正确的版本，其调用形式为：

```
对象名.方法名(参数表)
类名.方法名(参数表)
```

例如，IP 电话的计费，若建立 IP_Phone 类的对象 my，其调用为：

```
my.charge_Mode();
```

假如 charge_Mode()是一个类方法，则可以使用类名调用：

```
IP_Phone.charge_Mode();
```

运行时，系统会根据不同的调用者分辨所使用的是哪种类型的电话，并调用相应的具体计费方式的方法，从而实现计费方式功能的多态性。

【例 5.5】 这个例子展示了使用父类对象和子类对象调用同名方法时，不同的调用结果。

```
class Area{
    double f(double r)
    {
        return  3.14 * r * r;
    }
    double g(double x,double y)
    {
        return  0.5 * x * y;
    }
}
class Circle extends Area{
    double f(double r)
    {
        return  3.14 * 2.0 * r;
    }
}
public class PolyMorphism_Example1 {
    public static void main(String[] args){
        Area obj = new Area();
            //obj 为父类对象
        Circle cir = new Circle();
                //cir 为子类对象
        System.out.println("圆面积:" + obj.f(5.0));
                //调用父类的方法 f 求的是圆面积
        System.out.println("圆周长:" + cir.f(5.0));
                //调用子类的方法 f 求的是圆周长
        System.out.println("三角形面积:" + cir.g(2.0,8.0));
                //调用所继承父类的方法 g 求三角形面积
    }
}
```

PolyMorphism_Example1 类的执行结果如图 5.9 所示。

图 5.9　覆盖实现多态性的实例

5.3.3 重载实现多态性

第二种多态性通过重载来实现,它是在同一个类中定义多个同名方法,这些方法同名的原因是具有相同的功能且目的相同,但在实现该功能的具体方式和细节方面则有所不同,因此需要定义多种不同的实现方法。例如,加法属于一类操作,目的是把两个数通过“相加”变成一个数。但不同数加法的具体实现是不一样的,其中差别较大的有整数的加法、复数的加法和分数的加法。利用重载,可以给这些加法都取名为 add(x, y),然后再分别定义整数、复数和分数加法的具体实现。

由于重载发生在同一个类中,不能再用类名或对象名来区分不同的方法了,所以在重载中采用的区分方法是使用不同的形式参数表,包括形式参数的个数不同、类型不同或顺序的不同。例如,在加法中,整数加法的形参类型是整型,复数加法的形参类型是复数型。在具体实施加法时,尽管调用的是同一个方法名,但根据填入的参数的类型或者参量等的不同,系统可以确定调用哪个加法函数来完成加法计算。这样,使用时只需用不同的参数调用 add(x,y),即可实现不同类型的加法运算。

如何实现重载在 4.4.4 节中已有详细的叙述,下面补充一个实例。

【例 5.6】 普通电话的计费方式通常包括在节假日和晚上某个时间段打折,也就是说,在普通电话类中就至少包含两种计费方式。因此,可定义两个同名的不同 charge_Mode()方法,不同情况下调用不同的方法。为了简单起见,设置一个布尔变量 discount_time,需要计费打折时 discount_time 为真。重载一个 charge_Mode()方法的程序如下。

```
abstract class Telephone{
    long phoneNumber;
    final static double local_Call = 1;
    final static double distance_Call = 3;
    final static double international_Call = 9;
    double balance;
    abstract boolean charge_Mode(double call_Mode);
    double getBalance() {
         return balance;
    }
}
class Mobile_Phone extends Telephone{
    String networkType;
    String getType() {
        return networkType;
    }
    boolean charge_Mode(double call_Mode) {
        if(balance > 0.6) {
            balance -= 0.6;
            return true;
        } else
            return false;
    }
}

abstract class Fixed_Telephone extends Telephone {
    double monthFee;
}

class Ordinary_phone extends Fixed_Telephone{
```

```
    boolean longdistanceService;
    boolean internationalService;
    boolean charge_Mode(double call_Mode) {
        if(call_Mode == distance_Call
            &&! longdistanceService) return false;
                //检查国内长途服务是否开通
        if(call_Mode == international_Call
            &&! internationalService)return false;
                //检查国际长途服务是否开通
        if(balance > (0.2 * call_Mode)) {
            balance -= (0.2 * call_Mode);
            return true;
        } else
            return  false;
    }
    boolean charge_Mode(double call_Mode, boolean discount_time){
        if (discount_time)                              //如果是在打折的时间
            return  charge_Mode(call_Mode/2);           //收费减半
        else
            return  charge_Mode(call_Mode);             //正常时间的计费方式
        }
}

public  class  ReloadExample{                           //主类定义
    public static void main(String args[]){
        Ordinary_phone myOfficePhone = new Ordinary_phone();
        myOfficePhone.internationalService = true;
        myOfficePhone.balance = 500.0;
        if (myOfficePhone.charge_Mode(Telephone.international_Call,true))
                                            //调用第二个 charge_Mode 方法
            System.out.println("半价计费方式后剩余金额为: "
                + myOfficePhone.getBalance());
        if (myOfficePhone.charge_Mode(9))
                                            //调用第一个 charge_Mode 方法
            System.out.println("一般计费方式后剩余金额为: "
                + myOfficePhone.getBalance());
        }
}
```

程序运行结果如图 5.10 所示。

图 5.10　两种计费方式计费后的余额

图 5.10 中表明,用半价方式通话 1 分钟消耗话费 0.9 元,再用普通方式通话 1 分钟,消耗话费 1.8 元。仔细分析这个程序,会发现一个有趣的现象:在 Ordinary_phone 类中定义的 charge_Mode()方法实现了父类的同名抽象方法,接着又定义一个同名方法重载这个方法。

5.4　构造方法的继承与重载

因为构造方法的特殊性,所以本节单独讨论它的继承与重载问题。

5.4.1 构造方法的重载

构造方法的重载是指同一个类中定义不同参数的多个构造方法,以完成不同情况下对象的初始化。例如,对于表示平面坐标中的点的 Point 类,可定义不同的构造方法创建不同的点对象。

```
Point();                                    //未初始化坐标
Point(x);                                   //初始化一个坐标
Point(x,y);                                 //初始化两个坐标
```

一个类的若干构造方法之间可以相互调用。当类中一个构造方法需要调用另一个构造方法时,可以使用关键字 this,括号中可带也可不带参数,并且这个调用语句应该是该构造方法的第一个可执行语句。

【例 5.7】 前面没有定义 Ordinary_phone 类的构造方法,下面可以根据需要定义几个构造方法。

```
Ordinary_phone()                            //无参数的构造方法,对象的各域均置为默认初始值
       {  }
Ordinary_phone(boolean disService) {        //一个参数的构造方法
        this();                             //调自身的无参数的构造方法
        longdistanceService = disService;   //确定是否开通国内长途电话
}
Ordinary_phone(boolean disService, boolean intService) {
                                            //两个参数的构造方法
        this(disService);                   //调自身的带一个参数的构造方法
        internationalService = intService; //确定是否开通国际长途电话
}
Ordinary_phone(boolean disService, boolean intService, double b) {
                                            //3 个参数的构造方法
        this(disService, intService);       //调自身的带两个参数的构造方法
        balance = b;                        //设置话费金额
}
```

使用 this 关键字调用同类的其他构造方法,其优点是可以最大限度地提高对已有代码的利用程度,提高程序的抽象、封装程度,以及减少程序维护的工作量。

5.4.2 构造方法的继承

子类可以继承父类的构造方法,构造方法的继承遵循以下原则。

(1) 子类无条件地继承父类的无参数构造方法。

(2) 如果子类没有定义构造方法,则它将继承父类的无参数构造方法作为自己的构造方法;如果子类定义了构造方法,则在创建新对象时,将先执行来自继承父类的无参数构造方法,然后再执行自己的构造方法。

(3) 对于父类的带参数构造方法,子类可以通过在自己的构造方法中使用 super 关键字来调用它,但这个调用语句必须是子类构造方法的第一个可执行语句。

下面将分别对这 3 个原则举例说明。

1. 若有子类,父类一定要有无参构造方法给子类继承

前面谈到,子类无条件地继承父类的无参数的构造方法。这时一个类若有子类,这个类一定要有无参构造方法给子类继承。

【例 5.8】 父类没有定义无参构造方法给子类继承。

下面这个类没有定义无参构造方法，因此在有子类继承时会出现问题。父类和子类的代码如下。

```
class A{
   public A(int a){
      System.out.println(" String a" +a);
   }
}
class B extends A{
   public B(){
    System.out.println(" String b");
   }
}
public class ConstructorInheritance{
   public static void main(String args[]){
     B b = new B();
   }
}
```

图 5.11 父类没有定义无参构造方法给子类继承时出现的编译错误

在编译这个程序时显示如图 5.11 所示的信息。

要消除这个错误，类 A 中必须增加一个无参构造方法。例如，增加如下构造方法就不再出现编译错误：

```
public A(){
    System.out.println(" String a");
}
```

此时，子类就可以无条件地继承父类的无参数的构造方法。

当然如果子类不使用无参构造方法，父类也可以没有无参构造方法。例如，下面的程序就是一个合法的程序。

```
class A{
   public A(int a){
      System.out.println(" String a" +a);
   }
}
class B extends A{
   public B(int a){
    super(a);
    System.out.println(" String b");
   }
}
```

这个程序主要是 B 类的构造方法中有 super(a)语句，该语句的作用是调用父类的构造方法。假如以上程序没有这条语句，该程序也难以通过编译。下面还会进一步给出该语句的实例。

2. 父类与子类的构造方法的执行顺序

对于 5.4.2 节中的第(2)点，下面给出一个简单示意性的例子。

【例 5.9】 试分析下面程序的继承关系以及构造方法的调用顺序。我们是否可以先不看执行结果，分析构造方法的执行顺序是 A()、B()、C()还是 C()、B()、A()?

```
class A{
    public A(){
        System.out.println(" String a");
    }
}
class B extends A{
    public B(){
        System.out.println(" String b");
    }
}
class C extends B{
    public C(){
        System.out.println(" String c");
    }
}
public class ConstructorTest{
    public static void main(String args[]){
        C c = new C();
    }
}
```

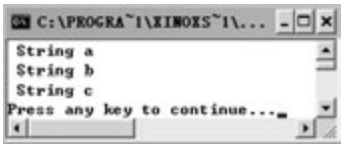


图 5.12 有继承关系的类构造方法的执行顺序

ConstructorTest 类的执行顺序如图 5.12 所示。

3. 在构造方法中 super 关键字的使用

super 是表示父类对象的关键字,super 表示当前对象的直接父类,代表当前父类对象的一个引用,其作用是利用 super 可使用父类对象的方法或域。

请看下面的例子,假设父类 Fixed_Telephone 有 4 个构造方法。

```
Fixed_Telephone()                              //无参数的构造方法,对象的各域均置为默认初始值
    { }
Fixed_Telephone(long pn) {
    phoneNumber = pn;                          //初始化电话号码
}
Fixed_ Telephone(long pn, double mf) {
    phoneNumber = pn;
    monthFee = mf;                             //初始化座机费
}
Fixed_Telephone(long pn, double mf, double b) {
    phoneNumber = pn;
    monthFee = mf;
    balance = b;                               //初始化电话费的话费余额
}
```

设计子类的构造方法可选择如下方式。

(1) 仅调用父类的无参数构造方法。

(2) 定义自己的一个(或多个)构造方法并使用 super()调用父类的带参数的构造方法。

根据上述方法,子类 Ordinary_phone 的构造方法设计如下。

```
Ordinary_phone(long pn, double mf, boolean ds) {
    super(pn, mf);                             //调用父类两个参数的构造方法为继承的域赋初值
    longdistanceService = ds;                  //用参数初始化自定义域
}
Ordinary_phone(long pn, double mf, double b, boolean ds){
    super(pn,mf, b);                           //调用父类 3 个参数的构造方法为继承的域赋初值
    longdistanceService = ds;                  //用参数初始化自定义域
}
```

按照这种方式,使用 super()可以很方便地定义各种构造方法。

5.4.3 重载和覆盖的综合举例

【例 5.10】 构造方法的继承与重载以及方法的覆盖的综合举例。

```
abstract class Telephone{
    long phoneNumber;
    final static double local_Call = 1;
    final static double distance_Call = 3;
    final static double international_Call = 9;
    double balance;
    abstract boolean charge_Mode(double call_Mode);
    double getBalance(){
        return balance;
        }
}
abstract class Fixed_Telephone extends Telephone{
    double monthFee;
    Fixed_Telephone()                                    //构造方法的重载
    {  }
    Fixed_Telephone(long pn) {
        this();
        phoneNumber = pn;
    }
    Fixed_Telephone(long pn, double mf) {
        this(pn);
        monthFee = mf;
    }
    Fixed_Telephone(long pn,double mf,double b) {
        this(pn,mf);
        balance = b;
    }
}
class Ordinary_phone extends Fixed_Telephone{
    boolean longdistanceService;
    boolean internationalService;
    Ordinary_phone(long pn,double mf,boolean a){
        super(pn, mf);
        longdistanceService = a;
    }
    Ordinary_phone(long pn,double mf,double b,boolean a){
        super(pn,mf, b);
        longdistanceService = a;
    }
    boolean charge_Mode(double call_Mode) {   //覆盖
        if (call_Mode == distance_Call
                          &&! longdistanceService) return false;
        if (call_Mode == international_Call
                        &&! internationalService) return false;
                                                 //检查国内长途和国际长途服务是否开通
        if (balance > (0.2 * call_Mode)){
            balance -= (0.2 * call_Mode);
            return true;
        } else
            return  false;
```

```
        }
        boolean charge_Mode(double call_Mode, boolean discount_time){
            if (discount_time){                          //如果是在打折的时间
                return  charge_Mode(call_Mode/2);
            } else
                return  charge_Mode(call_Mode);          //正常时间的计费方式
        }

        double getBalanee(){                             //覆盖 Telephone 的对应方法
            if (balance > monthFee)
                return balance - monthFee;
            else
                return -1;
        }
        public String toString() {                       //覆盖 Object 的对应方法
            String yn = longdistanceService ? "是":"否";
            return ( "电话号码为:" + phoneNumber
                   + "\n 每月座机费:" + monthFee
                   + "\n 剩余金额:" + balance
                   + "\n 可通国内长途:" + yn);
        }
    }
public class SumaryExample{                              //定义主类
    public static void main(String args[]){
        Ordinary_phone myHomePhone =
            new Ordinary_phone(87688888,25.0,100.0, true);
        System.out.println(myHomePhone.toString());
    }
}
```

本例中使用重载技术定义了 Fixed_TelephoneCsrd 类的 4 个构造方法；使用继承和重载技术定义了 Ordinary_phone 类的两个构造方法；使用覆盖技术在 Ordinary_phone 类中覆盖了父类的 getBalance()方法、charge_Mode()方法以及 Object 类的 toString()方法。主类中创建类的对象 myHomePhone 时使用了第二个构造方法，并对大部分的域都进行了初始化，图 5.13 输出了初始化后的结果。

图 5.13　Ordinary_phone 类的一个对象的初始化后的结果

5.5　接　　口

接口(interface)也有人翻译为界面，是用来实现类间达成共性和多重继承功能的一种结构，是相对独立的特定功能的属性集合。凡是需要实现这种特定功能的类，都可以继承并使用它。一个类只能直接继承(extends)一个父类，但可以同时实现(implements)若干接口。利用接口实际上就获得了多个特殊父类的属性，即实现了多重继承。

Java 的接口是用来组织应用中的各种类和调节它们之间的相互关系的一种结构，在语法上与类有些相似。它定义了若干抽象方法和常量，形成一个属性集合，该属性集合通常对应了某一组功能。接口定义的仅是实现某特定功能的一组对外接口和规范，而这个功能的真正实现是在实现这个接口的各类中完成的。

5.5.1 接口与多继承

Java 只支持单重继承，即一个类至多只能有一个直接父类。接口的主要作用就是可以帮助实现类似于类的多重继承的功能。所谓多重继承，是指一个子类可以有一个以上的直接父类，该子类可以继承它所有的直接父类的属性。某些面向对象的语言（如 C++）提供多重继承的语法级支持。而在 Java 中，出于简化程序结构的考虑，不直接支持类间的多重继承。但在解决实际问题的过程中，往往又需要这种机制。

由于 Java 只直接支持单重继承，所以 Java 程序中的类层次结构是树状结构，这种树状结构在随着类结构树的生长，越是处在下层的子类，它的间接父类就越多，所以继承的域及方法也会越来越多，有可能造成子类成员的膨胀、庞杂。

为了使 Java 程序的类层次结构更加合理，更符合实际问题的本质，可以把用于完成特定功能的若干属性（抽象方法和常量）组织成相对独立的属性集合，凡是需要实现这种特定功能的类，都可以继承这个属性集合并在类中使用它，这种属性集合就是接口。

需要特别说明的是，接口也可以认为是这些类之间行为的协议。但它只是定义了行为的协议，没有定义履行接口协议的具体方法。如果 Java 中的一个类要获取某一接口定义的功能，并不是通过直接继承这个接口中的属性和方法来实现的，因为接口中的属性都是没有方法体的抽象方法，接口定义的仅仅是实现某一特定功能的一组对外的协议和规范，并没有真正地实现这个功能。这些功能的真正实现是在实现这个接口的各个类中完成的，由这些类来具体定义接口中各抽象方法的方法体，以适合某些特定的行为。因而在 Java 中，通常把对接口功能的继承下来后具体实施描述的过程称为“实现”（implement）。

接口包含的是未实现的一些抽象的方法，它与抽象类有些相似。研究一下接口与抽象类到底有什么区别是很有意义的。它们之间存在以下的区别。

- 接口一般不含有任何实现了的方法（新版的 JDK 允许默认方法），而抽象类可以。
- 接口不能有任何变量，而抽象类可以。
- 接口可以继承（实现）多个接口，但抽象类只能继承一个父类。
- 类有严格的层次结构，而接口不是类层次结构的一部分，没有联系的类可以实现相同的接口。

5.5.2 接口的定义

接口是由常量和抽象方法组成的特殊类。定义一个接口与定义一个类是类似的。接口的定义包括两部分：接口声明和接口体。定义接口的一般简洁格式如下。

```
[public] interface 接口名 [extends 父接口名表] {
    域类型   域名 = 常量值;                          //常量域声明
    返回值   方法名(参数表);                         //抽象方法声明
}
```

接口声明中有两部分是必需的：interface 关键字和接口的名字。用 public 修饰的接口是公共接口，可以被所有的类和接口使用；没有 public 修饰符的接口则只能被同一个包中的其他类和接口使用。

像类之间可以继承一样，接口也具有继承性，子接口可继承父接口的所有属性和方法。但是，类只能继承一个父类，而接口可以继承（extends）多个父接口，即在 extends 后的“父接口名

表"中以逗号分隔所有的父接口名,这些接口就可以被新定义的接口所继承。图 5.14 给出接口的一个实例,并给出了这个接口声明的各部分的含义。

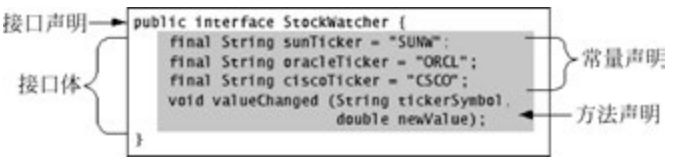


图 5.14 StockWatcher 接口的各部分及其含义

该接口定义了 3 个常量,它们是所监视的股票代码。这个接口还定义了 valueChanged()方法。实现这个接口的类将为这个方法提供具体的实现。

因为所有定义在接口中的常量都默认为 public、static 和 final,所有定义在接口中的方法默认为 public 和 abstract,所以不需要用修饰符限定它们。

假如已经编写了一个 StockMonitor 类,这个类的功能是监督股票的价格。它可以执行一个方法让其他的对象注册,以便得到通知。该类允许其他的类调用它的 watchStock()方法,从而知道什么时候特定的股票的价格发生改变。

```
public class StockMonitor{
    public void watchStock(StockWatcher watcher,
                           String tickerSymbol, double delta)
    {…}
}
```

这个方法的第一个参数 watcher 为 StockWatcher 对象。watcher 对象所属的类必须实现 StockWatcher 接口。其他两个参数提供了股票的代码和观察改变的数目。当 StockMonitor 类检测到一个感兴趣的变化时,它就会调用 watcher 对象的 valueChanged()方法。watchStock()方法通过第一个参数的数据类型确保所有替代 watcher 参数的对象实现 valueChanged()方法。

通过使用接口类型作为参数,替代 watcher 参数的对象类可以是 Applet 或者 Thread 等各种类型的类,但这些类必须实现 StockWatcher 接口。

5.5.3 接口的实现

为了使用接口,要编写实现接口的类。如果一个类实现一个接口,那么这个类就应提供在接口中定义的所有抽象方法的具体实现。

一个类可以根据定义在接口中的协议来实现接口。为了声明一个类来实现某个接口,在类的声明中要包括一条 implements 语句。因为 Java 支持接口的多继承,一个类可以实现多个接口,因此可以在 implements 后面列出要实现的多个接口,这些接口以逗号分隔。

以下是一个 Applet 类,它实现 StockWatcher 接口。

```
public class StockApplet extends Applet implements StockWatcher{
    …
    public void valueChanged(String tickerSymbol, double newValue){
        if (tickerSymbol.equals(sunTicker))
            { … }
        else if (tickerSymbol.equals(oracleTicker))
            { … }
```

```
        else if (tickerSymbol.equals(ciscoTicker))
            { … }
    }
}
```

这个类引用了定义在 StockWatcher 接口中的常量，如 oracleTicker、sunTicker 等。因为实现接口的类继承了接口中定义的常量，所以可以使用一般的变量名字来引用常量，也可以用下面的语句的方式，在其他任何类中使用接口常量：

```
StockWatcher.sunTicker
```

StockApplet 实现 StockWatcher 接口，因此它应提供 valueChanged 方法的实现。当一个类实现一个接口中的抽象方法时，这个方法的名字和参数类型及数目必须与接口中的方法匹配。

下面归纳一下实现接口时应注意的问题。

(1) 在类的声明部分，用 implements 关键字声明该类将要实现哪些接口。

(2) 类在实现抽象方法时，必须用 public 修饰符。

(3) 除抽象类以外，在类的定义部分必须为接口中所有的抽象方法定义方法体，且方法首部应该与接口中的定义完全一致。

(4) 若实现某接口的类是 abstract 的抽象类，则它可以不实现该接口所有的方法。但是对于这个抽象类的任何一个非抽象子类，不允许存在未被实现的接口方法，即非抽象类中不能有抽象方法存在。

5.5.4 接口的使用

定义一个新的接口时，实际上是定义了一个新的引用数据类型。在可以使用其他类型的名字(如变量声明、方法参数等)的地方，都可以使用这个接口名。例如，前面在 StockMonitor 类中的 watchStock()方法中的第一个参数的数据类型为 StockWatcher 接口。只有实现 StockWatcher 接口的类的对象可以替代 watcher 形参。

此外还应该注意，接口不能被覆盖，即不能有多个版本。假如想在 StockWatcher 中增加一个收集当前股票价格的方法，于是，试图定义一个新的版本：

```
public interface StockWatcher{
    final String sunTicker = "SUNW";
    final String oracleTicker = "ORCL";
    final String ciscoTicker = "CSCO";
    void valueChanged(String tickerSymbol,double newValue);
    void currentValue(String tickerSymbol,double newValue);
}
```

如果做了这个改变，实现老版本的 StockWatcher 接口的所有类的继承和实现关系都将中断，因为它们没有实现这个接口的所有方法。

为了达到以上增加一个方法的目的，可以创建新的接口来继承老接口。例如，可以创建一个 StockWatcher 的子接口 StockTracker：

```
public interface StockTracker extends StockWatcher{
    void currentValue(String tickerSymbol,double newValue);
}
```

这就可以避免上面提及的问题。

5.5.5 接口的完整实例

上面给出的例子属于示意性的例子,对于说明关于接口的某部分的语法机制很有作用。下面给出使用接口的一个完整实例。程序的讲解穿插在程序中的注解行中。

【例 5.11】 接口应用的完整实例。

```
//接口的声明
interface Speaker{
    public void speak();
    public void announce(String str);
}
//接口的实现
class Philosopher implements Speaker{
    private String philosophy;
    //初始化哲学家的哲理
    public Philosopher(String philosophy) {
        this.philosophy = philosophy;
    }
    //"唠叨"哲学家的哲理
    public void speak(){
         System.out.println(philosophy);
    }
    //发表一个宣言
    public void announce(String announcement){
         System.out.println(announcement);
    }
    //反复"唠叨"哲学家的哲理
    public void pontificate(){
        for (int count = 1; count <= 5; count++)
        System.out.println(philosophy);
    }
}
//接口的实现
class Dog implements Speaker{
    //发表狗的哲理
    public void speak(){
        System.out.println("woof");
    }
    //发表狗的哲理和宣言
    public void announce(String announcement) {
        System.out.println("woof: " + announcement);
    }
}
//演示使用一个接口多态性
class Talking{
      //初始化 Speaker 接口的一个引用
      //先后指向两个不同类的对象,调用它们的公共方法
    public static void main(String[] args) {
        Speaker current;
        current = new Dog();
        current.speak();

        current = new Philosopher("I think, therefore I am.");
        current.speak();
```

```
            ((Philosopher) current).pontificate();
        }
    }
```

程序中声明了接口 Speaker 的一个引用 current，用它先后指向两个不同的类对象，分别调用它们的公共方法 speak()，这两次调用分别使用的是不同的方法体，因为这是由两个类提供了 speak()方法的不同实现。在调用时，系统可以根据不同的对象找到正确的调用，这就是由接口实现的继承的多态性。前面我们在电话类及其子类看到抽象类的抽象方法 charge_Mode()具有多态性，这个例子展示了接口也有类似的功能。

另外还要指出，在这个程序最后一行，不能直接用 current.pontificate()进行调用，因为 current 属于 Speaker 类型，而通过 Speaker 接口只能请求到该接口所包含的方法的调用。因此，在上面的一条语句中可以通过 current 对象调用 speak()方法，而在这里必须将 current 进行类型转换，变成 Philosopher 的对象，再调用 pontificate()方法(pontificate 一词的意思是：装作教皇说话的样子)。上述程序的执行结果如图 5.15 所示。

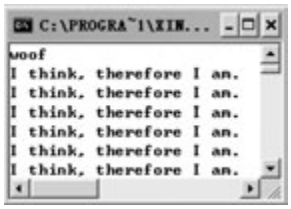


图 5.15　完整地实现一个接口后的执行结果

5.6　利用抽象类、接口和 Object 类实现多态性下的计算

现在提出一个比较特别的应用，要计算大量的(如 1000 个或更多)多种形状的面积之和，这些形状是随机产生的，即不知道什么时候产生什么样的形状。设定这样的条件是为了打消在已知图形的形状的情况下，使用一个一个地、按图形类别来计算面积的方式将计算出的面积进行累加的念头，也就是说必须用循环语句使用统一的方式进行面积的计算。

下面讨论用抽象类、接口和 Object 类三种方式实现多种形状图形的面积累加。

5.6.1　用抽象类实现多种形状面积的累加

定义 Shape、Rectangle、Circle 和 Square 类要利用多态性，确保每种形状分别用不同的方法来计算它们的面积。因此，定义 Shape 类作为超类，该类包含抽象方法 computeArea，然后，在其子类中实现和覆盖这个方法，下面是这些类的实现。

```
public abstract class Shape
{   protected double area;
    public abstract void computeArea();
}

public class Rectangle extends Shape
{   protected double width, height;
    public Rectangle(double _width, double _height)
    {   width = _width;
        height = _height;
    }
    public void computeArea()
    {   area = width * height; }
}
public class Circle extends Shape
{   public Circle(double _radius)
    {   type = "Circle";
```

```
        radius = _radius;
    }
    public void computeArea()
    {  area = Math.PI * radius * radius; }
}
public class Square extends Rectangle
{  public Square(double _side)
    {  super(_side, _side);
    }
}
```

有了上面这些类,就很容易创建它们的实例,并利用多态性选择适当的方法来计算。下面再定义一个类 ShapeTestWithAbstract,该类的 main()方法中声明了 1000 个形状对象的 Shape 数组,然后循环 1000 次随机产生 1000 个平面图形对象,形状为圆、矩形或正方形。该类的代码如下。

```
public class ShapeTestWithAbstract
{  public static void main(String args[])
    {  Shape shapes[] = new Shape[1000];
       double total_area = 0.0;
       int k = 0;
       for (int i = 0; i < shapes.length; i = i + 1){
             k = (int)Math.ceil(Math.random() * 3);
             System.out.println("k is :" + k);
             switch(k) {
           case 1: shapes[i] = new Circle(Math.random() * 10);
                     break;
           case 2: shapes[i] = new Rectangle(Math.random() * 10, Math.random() * 10);
                     break;
           case 3: shapes[i] = new Square(Math.random() * 10);
                     break;
             }
       }
       for (int i = 0; i < shapes.length; i = i + 1) {
            shapes[i].computeArea();
              total_area += shapes[i].area;
       }
       System.out.println("Total area is :" + total_area);
    }
}
```

图 5.16 是将程序中的 1000 改成 10,即随机产生的 10 个形状对象,然后对面积累加的结果。

图 5.16　10 个随机产生的平面图形对象面积之和

ShapeTestWithAbstract 类中有两轮循环,第一轮循环随机产生的 k 取值 1、2 或 3,分别代表圆、矩形或正方形,然后在 case 语句中创建相应的 3 种对象。第二轮循环,对每个对象计

算面积,然后累加面积。

5.6.2 用接口实现多种形状面积的累加

用接口的方式实现多种形状面积的累加,需要将用抽象类表示的 Shape 类改成接口。由于接口的语法定义要求,要把原来抽象类中的成员变量去掉,成员方法改成抽象方法。去掉成员变量后,没有存储单元保存面积,因此要将求图形面积的无返回类型 computeArea()方法改成返回一个 double 类型的抽象方法。所以,这个接口定义为:

```
public interface Shape2
{   public abstract double computeArea();
}
```

定义好了接口后,再要让原来的图形类由继承抽象类改为实现 Shape2 接口,因此这些图形类分别定义为:

```
public class Circle2 implements Shape2
{   protected double radius;
    public Circle2(double _radius) {
      radius = _radius;
    }
    public double computeArea()
    { return Math.PI * radius * radius; }
}

public class Rectangle2 implements Shape2
{   protected double width, height;
    public Rectangle2(double _width, double _height)
    {   width = _width;
        height = _height;
    }
    public double computeArea()
    {   return width * height; }
}

public class Square2 extends Rectangle2
{   public Square2(double _side)
    {   super(_side, _side);
    }
}
```

定义了图形类之后,再对原来的主类 ShapeTestWithAbstract 稍做修改,把抽象类数组类型改成接口数组类型,把对各对象的保存面积的成员变量的值的累加改成各对象的求面积的成员方法的返回值的累加。新的主类的代码如下。

```
public class ShapeTestWithInterface
{   public static void main(String args[])
    {   Shape2 shapes[] = new Shape2[10];              //接口类型的数组
        double total_area = 0.0;
        int k = 0;
        for (int i = 0; i < shapes.length; i = i + 1){
                k = (int)Math.ceil(Math.random() * 3);
                System.out.println("k is :" + k);
                switch(k) {
               case 1: shapes[i] = new Circle2(Math.random() * 10);
                       break;
```

```
                case 2: shapes[i] = new Rectangle2(Math.random() * 10, Math.random() * 10);
                        break;
                case 3: shapes[i] = new Square2(Math.random() * 10);
                        break;
                  }
            }
            for (int i = 0; i < shapes.length; i = i + 1) {
                    total_area += shapes[i].computeArea();
                                                        //求面积的方法的返回值的累加
            }
            System.out.println("Compute Aera With Interface,Total area is :" + total_area);
        }
}
```

该程序的执行结果如图 5.17 所示。

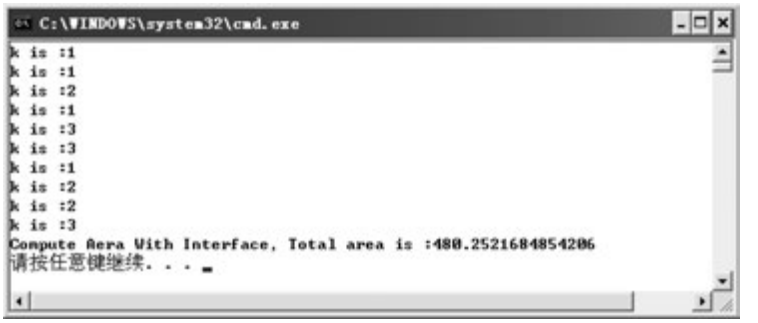


图 5.17　用接口实现多种形状面积的累加的结果

5.6.3　用一个 Object 数组实现多种形状面积的累加

定义一个数组,它可以同时存储矩形、圆和正方形,这个想法看起来似乎不可行。一个数组必须包含同种类型的元素,但矩形、圆和正方形是不同类型的对象。

如前所述,每个 Java 类都是由 Object 扩展而来的。因此,所有的类都属于 Object 类型,我们可以创建一个 Object 类型的数组来存储任何类型的对象,也就可以存储矩形、圆和正方形对象。反过来,对每个数组元素,在 instanceof 的帮助下能确定其原来的类型。下面的 ShapeTestWithObject 类中使用的还是 5.6.1 节中定义的 Circle、Rectangle 和 Square 类。

```
public class ShapeTestWithObject
{   public static void main(String args[])
    {   Object shapes[] = new Object[1000];
        double total_area = 0.0;
        int k = 0;
        for (int i = 0; i < shapes.length; i = i + 1){
                k = (int)Math.ceil(Math.random() * 3);
            System.out.println("k is :" + k);
            switch (k) {
                case 1: shapes[i] = new Circle(Math.random() * 10);
                        break;
                case 2: shapes[i] = new Rectangle(Math.random() * 10, Math.random() * 10);
                        break;
                case 3: shapes[i] = new Square(Math.random() * 10);
                        break;
                  }
        }
```

```
        for (int i = 0; i < shapes.length; i = i + 1) {
            if ((shapes[i]) instanceof Circle) {
                ((Circle)shapes[i]).computeArea(); //恢复原来的对象类型并求面积
                total_area += ((Circle)shapes[i]).area;
              }else if(shapes[i] instanceof Rectangle) {
             ((Rectangle)shapes[i]).computeArea();  //恢复原来的对象类型并求面积
                total_area += ((Rectangle)shapes[i]).area;
              }else if(shapes[i] instanceof Square) {
                ((Square)shapes[i]).computeArea();
                                                    //恢复原来的对象类型并求面积
                total_area += ((Shape)shapes[i]).area;
              }
        }
        System.out.println("Total area is :" + total_area);
    }
}
```

该程序的执行结果如图 5.18 所示。

图 5.18　用一个 Object 数组实现多种形状面积累加的结果

仔细分析一下上述程序，在使用 Object 数组实现多种形状面积累加时，类的设计可以更加简化，现在可以不需要抽象类 Shape 了。5.6.1 节中 Circle、Rectangle 和 Square 类的定义可以按如下方式进行。

```
public class Circle{ }                              //不需要 extends Shape
public class Rectangle{ }                           //不需要 extends Shape
public class Square extends Rectangle{ }
```

即 Circle 类和 Rectangle 类不再继承 Shape 类，ShapeTestWithObject 需要稍做修改，修改后的完整代码如下。

```
public class Rectangle
{   protected double width, height;
    public Rectangle(double _width, double _height)
    {   width = _width;
        height = _height;
    }
    public double computeArea()
    {   return width * height; }
}
public class Circle
{   protected double radius;
    public Circle(double _radius)
    {   radius = _radius;
```

```
    }
    public double computeArea()
    {  return Math.PI * radius * radius; }
}

public class Square extends Rectangle
{  public Square(double _side)
   {  super(_side, _side);
   }
}

public class ShapeTestWithObjectArray
{  public static void main(String args[])
   {  Object shapes[] = new Object[10];
      double total_area = 0.0;
      int k = 0;
      for (int i = 0; i < shapes.length; i = i + 1){
            k = (int)Math.ceil(Math.random() * 3);
       System.out.println("k is :" + k);
       switch (k) {
          case 1: shapes[i] = new Circle(Math.random() * 10);
                  break;
          case 2: shapes[i] = new Rectangle(Math.random() * 10, Math.random() * 10);
                  break;
          case 3: shapes[i] = new Square(Math.random() * 10);
                  break;
        }
      }
      for (int i = 0; i < shapes.length; i = i + 1) {
          if ((shapes[i]) instanceof Circle) {
                      //恢复 Circle 对象类型并求面积
              total_area += ((Circle)shapes[i]).computeArea();
        }else if(shapes[i] instanceof Rectangle) {
                      //恢复 Rectangle 对象类型并求面积
              total_area += ((Rectangle)shapes[i]).computeArea();
        }else if(shapes[i] instanceof Square) {
                      //恢复 Square 对象类型并求面积
              total_area += ((Square)shapes[i]).computeArea();
          }
      }
      System.out.println("Total area is :" + total_area);
   }
}
```

该类的执行结果除了与图 5.18 中的数值不一样之外,其他方面是完全一样的。

5.7 泛型与容器类

Java 中“泛型”与“容器类”是紧密关联、相互配合的核心特性,它们的关系主要体现在:泛型为容器类提供了类型安全保障,而容器类是泛型最典型的应用场景。

通过在容器类定义时指定类型参数(如 List < String >),编译器会在编译阶段检查元素类型,确保存入容器的元素类型与指定类型一致,避免了运行时错误。

5.7.1 泛型

泛型(Generics)是 Java 5 引入的特性,是 Java 中一种参数化类型的机制,允许在定义类、

接口或方法时指定类型参数，在使用时再确定具体类型。它解决了集合中元素类型不明确导致的类型转换问题，提供了编译时类型安全检查，实现类型安全的代码复用。

其核心作用如下。

- 编译时检查类型，避免运行时类型转换错误。
- 消除代码中的强制类型转换，使代码更简洁。

泛型的基本语法是使用<类型参数>，例如 List < String >表示只能存储 String 类型的列表 List。

1. 泛型的基本概念

泛型，即参数化类型，将类型由原来的具体类型予以参数化，然后在调用时传入具体的类型。泛型的好处是在编译时检查类型安全，并且所有的强制转换都是自动和隐式的，提高了代码的重用率。

定义泛型类，可以在类名后面添加尖括号，里面包含一个或多个类型参数。在定义泛型时，有以下几个基本概念。

- 类型参数。用< T >表示，T 可以是任意标识符（通常用 E 表示元素、K 表示键、V 表示值）。
- 参数化类型。如 List < String >，指定了具体类型 String。
- 原始类型。未指定类型参数的泛型类型（如 List），不推荐使用。
- 泛型擦除。编译后泛型信息会被擦除，实际运行时不保留类型参数信息。

2. 泛型的使用形式

泛型的使用有以下 3 种形式。

- 泛型类。在类定义时声明类型参数。
- 泛型接口。在接口定义时声明类型参数。
- 泛型方法。在方法声明时声明类型参数，可独立于类的泛型。

3. 泛型通配符

类型通配符用问号（?）表示，表示未知类型。通配符可以用于参数、字段、局部变量和返回类型。通配符的上限和下限可以限制类型的范围。

- <?>。无界通配符，表示任意类型。
- <?extends T >。通配符的上界，即类型必须是 T 或 T 的子类。
- <?super T >。通配符的下界，即类型必须是 T 或 T 的父类。

5.7.2 容器类

Java 集合框架提供了一系列的接口和类，即容器类（Container Classes），用于存储和操作一组对象。主要的接口有如下几种。

- List。有序可重复的集合（如 ArrayList、LinkedList）。
- Set。无序不可重复的集合（如 HashSet、TreeSet）。
- Map。键值对集合（如 HashMap、TreeMap）。

下面逐一讨论这 3 种类。

1. List

List 是有序的集合，可以包含重复元素。常见的实现类有 ArrayList、LinkedList。下面给出使用 List 的一个实例。

```
import java.util.ArrayList;
import java.util.List;

public class ListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Orange");

        //使用 for-each 循环遍历
        for (String fruit : list) {
            System.out.println(fruit);
        }
    }
}
```

2. Set

Set 不允许重复元素。常见的实现类有 HashSet、TreeSet。这里也给出 HashSet 的一个应用实例。

```
import java.util.HashSet;
import java.util.Set;

public class SetExample {
    public static void main(String[] args) {
        Set<Integer> set = new HashSet<>();
        set.add(1);
        set.add(2);
        set.add(3);
        set.add(1);                                    //重复元素,不会被添加

        System.out.println(set);                       //输出:[1, 2, 3]
    }
}
```

3. Map

Map 用于存储键值对,键不能重复。常见的实现类有 HashMap、TreeMap。下面是使用 HashMap 的程序实例。

```
import java.util.HashMap;
import java.util.Map;

public class MapExample {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 28);

        //通过键获取值
        System.out.println(map.get("Bob"));            //输出:30

        //遍历 Map
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
```

```
            System.out.println(entry.getKey() + " : " + entry.getValue());
        }
    }
}
```

5.7.3 容器类与泛型结合使用的实例

下面给出一个容器类通常与泛型结合使用的示例程序。

```
//GenericsContainerExample.java
import java.util.*;

//自定义泛型类
class Box<T> {
    private T content;

    public void setContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}

public class GenericsContainerExample {
    public static void main(String[] args) {
        //1. 泛型类的使用
        Box<String> stringBox = new Box<>();
        stringBox.setContent("Hello Generics");
        System.out.println("Box 中的内容: " + stringBox.getContent());

        Box<Integer> intBox = new Box<>();
        intBox.setContent(100);
        System.out.println("Box 中的数字: " + intBox.getContent());

        //2. List 容器(有序可重复)
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana"); fruits.add("Apple"); //允许重复元素
        System.out.println("\n 水果列表: " + fruits);

        //3. Set 容器(无序不可重复)
        Set<String> uniqueFruits = new HashSet<>();
        uniqueFruits.add("Apple"); uniqueFruits.add("Banana");
        uniqueFruits.add("Apple");                    //重复元素会被忽略
        System.out.println("不重复的水果: " + uniqueFruits);

        //4. Map 容器(键值对)
        Map<String, Integer> fruitCounts = new HashMap<>();
        fruitCounts.put("Apple", 3);
        fruitCounts.put("Banana", 2);
        fruitCounts.put("Orange", 5); System.out.println("\n 水果数量:");
        for (Map.Entry<String, Integer> entry : fruitCounts.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
```

```
        }

        //5. 泛型方法
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        System.out.println("\n 数字列表: " + numbers);
        System.out.println("列表中的最大值: " + findMax(numbers));
    }

    //泛型方法:查找列表中的最大值
    public static <T extends Comparable<T>> T findMax(List<T> list) {
        if (list.isEmpty()) {
            return null;
        }
        T max = list.get(0);
        for (T item : list) {
            if (item.compareTo(max) > 0) {
                max = item;
            }
        }
        return max;
    }
}
```

该程序执行后的结果如图 5.19 所示。

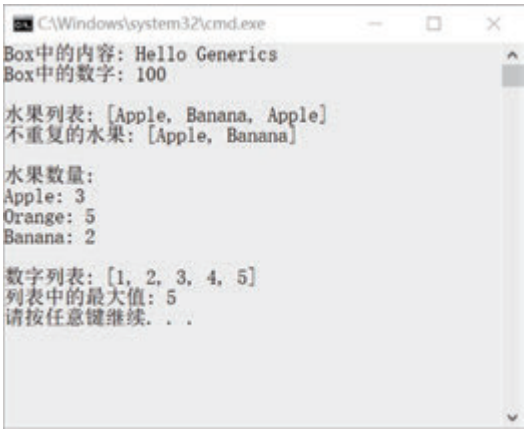


图 5.19　容器类与泛型结合使用的实例程序的执行结果

对于这个比较抽象的程序给出以下必要的解释。

1. 自定义泛型类 Box

使用<T>定义类型参数,表示 Box 可以存储任意类型的数据; setContent()和 getContent()方法使用 T 作为参数和返回值类型;实例化时指定具体类型(如 Box<String>、Box<Integer>)。

2. List 容器

使用 ArrayList 实现,存储 String 类型;允许添加重复元素,保持插入顺序。

3. Set 容器

使用 HashSet 实现,存储 String 类型;自动去掉重复元素,元素顺序不固定。

4. Map 容器

使用 HashMap 实现,键为 String 类型,值为 Integer 类型;通过 put()添加键值对,

entrySet()遍历所有键值对。

5. 泛型方法 findMax

使用< T extends Comparable < T >>限制 T 必须是可比较的类型；可以对任意实现了 Comparable 接口的列表(如 Integer、String)查找最大值。从上面的例子可以看出，泛型与容器类常常结合使用，泛型让容器类在保持灵活性和复用性的同时，实现了编译时类型检查，避免了类型转换错误；而容器类则通过泛型的特性，成为 Java 中处理集合数据的核心工具。两者结合，构成了 Java 中类型安全、灵活高效的集合处理体系。

5.8 Java 的函数式接口与 Lambda 表达式

内置函数式接口配合 Lambda 表达式，可以大幅简化集合处理、数据筛选、类型转换等操作，是 Java 函数式编程的基础工具。

5.8.1 函数式接口

函数式接口(Functional Interface)是 Java 8 引入的一种特殊接口，只包含一个抽象方法(可以有多个默认方法或静态方法)。它的作用是为 Lambda 表达式提供类型支持，是 Lambda 表达式的“载体”。

Java 中用@FunctionalInterface 注解标识函数式接口，该注解并非必需，但推荐使用，编译器会检查是否符合规范。

常见的内置函数式接口如下。

- Runnable。无参数无返回值(void run())。
- Consumer < T >。接收一个参数，无返回值(void accept(T t))。
- Supplier < T >。无参数，返回一个值(T get())。
- Function < T, R >。接收 T 类型参数，返回 R 类型结果(R apply(T t))。
- Predicate < T >。接收 T 类型参数，返回 boolean(boolean test(T t))。

5.8.2 Lambda 表达式

Lambda 表达式是函数式接口的实例化方式，用于简化匿名内部类的写法，让代码更简洁。它的本质是一段可以传递的代码(函数作为参数)。

Lambda 表达式基本语法是：

```
(参数列表) ->{代码体}
```

若参数列表只有一个参数，可省略括号；若代码体只有一行，可省略花括号{}和 return(如果有返回值)。

5.8.3 函数式接口与 Lambda 表达式举例

【例 5.12】 函数式接口与 Lambda 表达式示例程序。

```
//LambdaExample.java
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
```

```
import java.util.function.Function;
import java.util.function.Predicate;

//自定义函数式接口
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}

public class LambdaExample {
    public static void main(String[] args) {
        //1. 使用自定义函数式接口
        Calculator add = (a, b) -> a + b;          //简化写法
        Calculator multiply = (int a, int b) -> {  //完整写法
            return a * b;
        };
        System.out.println("3 + 5 = " + add.calculate(3, 5));
        System.out.println("3 * 5 = " + multiply.calculate(3, 5));

        //2. 使用内置函数式接口:Consumer(消费数据)
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        System.out.println("\n 打印所有名字:");
        names.forEach(name -> System.out.println(name));
                            //简化为 names.forEach(System.out::println)

        //3. 使用内置函数式接口:Predicate(判断条件)
        System.out.println("\n 长度大于 3 的名字:");
        filterNames(names, name -> name.length() > 3);

        //4. 使用内置函数式接口:Function(数据转换)
        System.out.println("\n 名字转换为长度:");
        convertNames(names, name -> name.length());
    }

    //利用 Predicate 过滤名字
    private static void filterNames(List<String> names, Predicate<String> predicate) {
        for (String name : names) {
            if (predicate.test(name)) {
                System.out.println(name);
            }
        }
    }

    //利用 Function 转换名字
    private static void convertNames(List<String> names, Function<String, Integer> function) {
        for (String name : names) {
            System.out.println(name + " 的长度是:" + function.apply(name));
        }
    }
}
```

运行该程序,其结果如图 5.20 所示。

对照程序的执行结果,不难看出其主要部分如下。

图 5.20　一个函数式接口与 Lambda 表达式示例程序的执行结果

(1) 自定义函数式接口。Calculator 只有一个 calculate 方法,用于演示 Lambda 如何实现自定义接口。

(2) Lambda 表达式的两种写法。①简化写法。省略参数类型、花括号和 return。②完整写法。指定参数类型并使用花括号包裹代码体。

程序中共有 5 处使用了 Lambda 表达式,它们采用了不同的简化写法,具体如下。

① 加法计算器(简化写法)。

```
Calculator add = (a, b) -> a + b;
```

该写法省略了参数类型(int)、花括号({})和 return 关键字。因为代码体只有一行且有返回值,符合 Lambda 的简化规则。

② 乘法计算器(完整写法)。

```
Calculator multiply = (int a, int b) -> {
    return a * b;
};
```

该写法保留了参数类型(int)、花括号({})和 return 关键字,是 Lambda 的完整形式。适合代码体有多行的场景。

③ 遍历打印名字(参数简化)。

```
names.forEach(name -> System.out.println(name));
```

该写法省略了参数圆括号,因为只有一个参数 name;同时省略了花括号({}),因为代码体只有一行。

④ 过滤名字的条件(单行判断)。

```
filterNames(names, name -> name.length() > 3);
```

该写法省略了参数圆括号(因为是单个参数 name)、花括号({})和 return(因为是单行返回条件判断)。它用于 Predicate 接口的 test 方法,直接返回判断结果。

⑤ 转换名字为长度(单行转换)。

```
convertNames(names, name -> name.length());
```

该写法同样省略了参数圆括号、花括号({})和 return(单行返回转换结果)。它用于 Function 接口的 apply 方法,实现从 String 到 Integer 的转换。

(3) 内置函数式接口的应用。

① Consumer。通过 forEach 消费集合中的元素(打印名字)。

② Predicate。定义过滤条件(名字长度大于 3)。

③ Function。实现数据转换(将名字转换为其长度)。

这里的 Consumer、Predicate 和 Function 是最常用的内置函数式接口,分别用于处理不同场景的数据操作,以下是具体解释。

• Consumer<T>。消费数据(只输入不输出)。

核心方法:void accept(T t)。该方法接收一个 T 类型的参数,执行某种操作,但没有返回值(仅"消费"数据,不产生新结果)。相当于"消费者",负责对输入的数据进行处理(如打印、保存、修改等),但不返回任何结果。

例如,消费 String 类型的数据(打印名字)如下。

```
names.forEach(name -> System.out.println(name));
```

这里 forEach 方法接收 Consumer<String>作为参数,Lambda 表达式 name-> System.out.println(name)实现了"消费"逻辑:接收名字并打印。

• Predicate<T>。判断条件(输入并返回布尔值)。

核心方法:boolean test(T t)。该方法接收一个 T 类型的参数,执行判断逻辑,返回 boolean 结果(true 或 false)。相当于"过滤器",负责对输入的数据进行条件判断,返回是否符合条件。

例如,判断 String 类型的数据是否符合条件(长度大于 3):

```
filterNames(names, name -> name.length() > 3);
```

这里 filterNames 方法接收 Predicate<String>作为参数,Lambda 表达式 name-> name.length()>3 实现了判断逻辑:接收名字并返回其长度是否大于 3。

• Function<T,R>。数据转换(输入并返回新数据)。

核心方法:R apply(T t),该方法接收一个 T 类型的参数,执行转换逻辑,返回一个 R 类型的新结果(将一种类型数据转换为另一种类型)。例如,将 String 类型转换为 Integer 类型(获取名字长度):

```
convertNames(names, name -> name.length());
```

这里 convertNames 方法接收 Function<String,Integer>作为参数,Lambda 表达式 name-> name.length()实现了转换逻辑:接收名字(String)并返回其长度(Integer)。

程序中同时展示了 Lambda 表达式的写法可以根据场景灵活简化。完整写法和不同程度的简化写法的核心规则如下。

• 单个参数可省略括号()。

• 单行代码可省略花括号{}和 return。

• 参数类型可省略(编译器会自动推断其类型)。

5.9 小　　结

继承是面向对象程序设计方法中的一种重要手段,通过继承一个子类可以拥有父类的非私有域和方法。继承不仅可以实现软件的重用,而且体现面向对象程序设计的另一个主要特

征——多态。多态可以用两种方法实现：覆盖和重载。

实现覆盖时，对于重定义继承的方法，系统会根据调用该方法对象所属的类来决定选择调用哪个方法。对于子类创建的对象，如果子类重定义了父类的方法，则运行时系统调用子类的方法；如果子类未重定义而是继承了父类的方法，则运行时调用父类的方法。

重载是通过定义同一个类中多个同名的不同方法实现的。编译时可根据参数（个数、类型、顺序）的不同来区分不同的方法，以便调用时选择不同的操作。

接口也是实现多态的语法机制之一。接口的声明仅仅给出了抽象方法，而具体实现接口所规定的功能则需要某个类为接口中的抽象方法定义实在的方法体，即实现这个接口。同定义的类一样，凡是可以使用其他类型名的地方都可以使用接口。由于一个类可以实现多个接口，所以它也是 Java 实现多继承的机制。

总之，继承机制带来了 Java 类的可重用性，而多态性给我们带来的好处是，可以为多个相关类提供对外统一的接口，提高类的抽象性和灵活性。

习　题　5

1. 什么是继承？继承的意义是什么？如何定义继承关系？

2. “子类的域和方法的数目一定大于或等于父类的域和方法的数目”，这种说法是否正确？为什么？

3. 设有小学生、中学生、大学生类，为他们设计状态与行为。并利用第 4 章的学生类设计类的继承程序，分别创建这些学生对象并输出其信息。

4. 什么是域的隐藏？什么是方法的覆盖？方法的覆盖与域的隐藏有什么不同？方法的覆盖与方法的重载有什么不同？

5. 如何实现方法的覆盖？实现方法的覆盖应注意什么？

6. 什么是多态？面向对象程序设计为什么要引入多态的特性？使用多态有什么优点？

7. Java 程序如何实现多态？有哪些实现方式？

8. 解释 this 和 super 的意义和作用。

9. 什么是构造方法的重载？如何实现重载的构造方法之间的调用？

10. 构造方法的继承原则有哪些？继承方式下子类的构造方法如何设计？

11. 以第 4 章第 15 题中定义的 Address 类作为超类，定义一个子类，使它拥有另外一个成员变量来存储电话号码。

12. 下面的程序创建一个名为 SuperClass 的类，它的构造方法有一个 String 类型的参数。另一个名为 SubClass 的子类，它从 SuperClass 类扩展而来，也有一个参数为 String 类型的构造方法。如下的 Test 类在实例化 SubClass 的一个对象时出现错误，试改正。

```
public class SuperClass
{   public SuperClass(String msg)
    {   System.out.println("SuperClass constructor: " + msg); }
}
public class SubClass extends SuperClass
{   public SubClass()
    {   System.out.println("SubClass constructor"); }
}
public class Test
```

```
{  public static void main(String args[])
   {  SubClass descendent = new SubClass(); }
}
```


13. 用最终成员变量和抽象方法,为一个“Java 州立大学”的学生建立账单系统,州内外的学生收费不同,州内每学分收费 $750,州外为 $2000,每个学生的账单上有学校名称、学生姓名、信用卡使用的时间以及账单的总数。

14. 定义一个 Object 数组,它可以存储一个矩形、一个圆、一个双精度数或一个整数。

15. 创建一个名为 List 的类,它可存储任何类型的对象,并可以在任何时候增加或删除对象。

16. 定义一个接口 OneToN,在接口体中包含一个抽象方法 disp()。定义 Sum 和 Pro 类,并分别用不同代码实现 OneToN 中的 disp()方法,在 Sum 的方法中计算 1～n 的和,在 Pro 的方法中计算 1～n 的乘积。

17. 编写一个类使用第 16 题中的接口类型衍生的类。

18. 给定下面类定义:

```
class Test1 {
    float aMethod(float a, float b) { return a;}
}
class Test2 extends Test1 {
    //xxxx
}
```

请问下面方法中,可以加入类 Test2 中的注释行处的是(　　)。(多选题)

A. float aMethod(float b,float a){ return a;}

B. public int amethod(int a,int b) { return a;}

C. protected float aMethod(float p,float q){ return a;}

D. private float aMethod(float p,float q) { return a;}

19. 给定以下代码,下列选项中,能替换其中的注释行的是(　　)。

```
class A {
      A(int i) { }
}
public class B extends A {
      B() {
          //xxxxx
      }
      public static void main(String args[]) {
          B b = new B();
      }
}
```

A. super(100);　　B. this(100);　　C. super();　　D. this();

20. 编译、运行下面代码将发生(　　)。

```
class Base {}
class Sub extends Base {}
public class BaseToSub {
    public static void main(String argv[]) {
        Base b = new Base();
        Sub s = (Sub) b;
    }
}
```

A. 编译和运行都不会出错

B. 编译出错

C. 运行时抛出例外

21. 考虑类定义：

```
class BaseWidget extends Object{
    String name = "BaseWidget";
    void speak() { System.out.println("I am a " + name); }
}
class TypeAWidget extends BaseWidget{
    TypeAWidget() { name = "TypeA"; }
}
```

基于以上类定义，下列代码片段中正确的是(　　)。

A. Object A = new BaseWidget(); A.speak();

B. BaseWidget B = new TypeAWidget(); B.speak();

C. TypeAWidget C = new BaseWidget(); C.speak();

22. 编写一个泛型类 Pair，该类有两个类型参数 K 和 V，分别表示键和值的类型。要求包含获取和设置键值对的方法，并编写测试程序。

23. 编写一个泛型方法，该方法接收一个任意类型的数组和一个元素，返回该元素在数组中出现的次数。如果数组为空或元素不在数组中，则返回 0。

24. 编写一个方法，该方法接收一个 List <? extends Number >，返回列表中所有元素的平方和(注意：使用 Number 的 doubleValue 方法)。

25. 创建一个 ArrayList，添加一些字符串，然后编写代码按字母顺序对列表进行排序(不区分大小写)。

26. 使用 HashSet 存储一组整数，然后编写代码检查该集合中是否包含某个特定的整数，并计算集合的大小。

27. 使用 HashMap 存储一个简单的电话簿(姓名和电话号码)，然后实现根据姓名查找电话号码的功能，并能够遍历打印所有联系人的信息。

28. 使用 Runnable 函数式接口和 Lambda 表达式创建两个线程，分别打印 1～10 和 11～20 的数字，要求使用 Lambda 简化线程创建。

29. 创建一个 List < Integer >包含 1～10 的数字，使用 Predicate 过滤出偶数，再使用 Function 将这些偶数乘以 2，最后使用 Consumer 打印结果。

30. 自定义一个函数式接口 StringProcessor，包含方法 String process(String str)。使用 Lambda 表达式实现该接口的 3 个实例。

- 将字符串转换为大写。
- 截取字符串的前 3 个字符。
- 反转字符串(如 "abc" 变为 "cba")。

编写测试代码验证这 3 个实例的功能。

习题 5