第1章 对象的基本结构

1.1 对象的引用和对象的变量

类封装了一类事物共有的属性和行为(相比 C 语言的结构体前进了一大步),并用一定的语法格式描述所封装的属性和行为。封装的关键是抓住事物的两个方面:属性和行为,即数据以及在数据上所进行的操作。

类是用于创建对象(对象也称类的实例)的一种数据类型(高级语言总是先有类型,再定义数据),也是 Java 语言中最重要的数据类型。类声明的变量称作对象变量,简称对象。

下面用简单的 CarSUV 类强调一下对象的基本结构。

```
public class CarSUV {
    int speed;
    int weight = 200;
    int upSpeed(int n){
        if(n<=0||n>=260)
           return speed;
        speed += n;
        return speed>=260?260:speed;
    }
}
```

上述 CarSUV 类是用户定义的一种数据类型,下列代码用这个类型声明了两个对象,分别是 redCar 和 blueCar:

```
CarSUV redCar = null;
CarSUV blueCar =null;
```

内存示意图如图 1.1 所示。

图 1.1 类声明的 redCar 对象和 blueCar 对象

此时,系统认为 redCar 对象和 blueCar 对象中的数据是 null,称这样的对象是空对象。程序要避免使用空对象,即在使用对象之前,要确保为对象分配了变量(也称为对象分配实体)。下列代码为 redCar 对象和 blueCar 对象分配变量:

```
redCar = new CarSUV();
blueCar = new CarSUV();
```

new 是 Java 中的一个关键字,也是一个运算符,new 只能与构造方法进行运算,其作用非常类似 C 语言中的分配内存的函数 *calloc()(*calloc()分配内存,并初始化所分配的内存;*malloc()分配内存,但不初始化所分配的内存),即为 CarSUV 类中的

成员变量 speed、weight 分配内存,为方法 speedUp()分配入口地址。new 运算符为成员变量 speed、weight 分配内存后,进行初始化,然后执行构造方法,最后计算出一个称作“引用”的 int 型的值。程序需要将这个引用赋值给某个对象,以确保所分配的成员变量 speed、weight 是属于这个对象的,即确保 speed 和 weight 是分配给该对象的变量。CarSUV 类可以声明多个不同的对象,并使用 new 运算符为这些对象分配各自的变量,分配给不同对象的变量各自占有不同的内存空间(给对象分配变量也称创建对象)。此时的内存示意图如图 1.2 所示(图中画的小汽车是为了形象而画)。

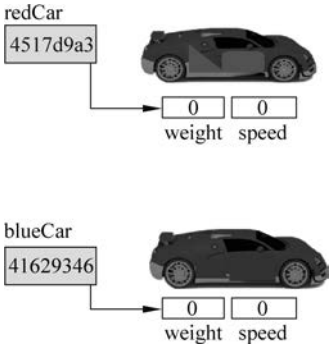

图 1.2 为 redCar 对象和 blueCar 对象分配变量

对象(变量)负责存放引用,以确保对象可以操作分配给该对象的变量(也称分配给对象的变量是对象的实体)以及调用类中的方法。图 1.2 中的箭头示意对象可以使用访问符“.”访问分配给自己的变量。执行下列代码:

```
redCar.weight = 100;
blueCar.weight = 120;
```

那么 redCar 对象的 weight 的值是 100,blueCar 对象的 weight 的值是 120。此时的内存示意图如图 1.3 所示。

当对象调用方法时,方法中出现的成员变量就是指分配给该对象的变量(体现封装性)。执行下列代码:

```
redCar. upSpeed(60);
blueCar. upSpeed(80);
```

那么 redCar 对象的 speed 的值由 0 变成 60,blueCar 对象的 speed 的值从 0 变成 80。此时的内存示意图如图 1.4 所示。

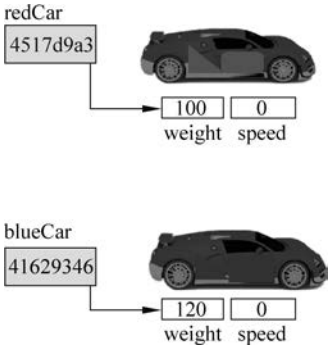

图 1.3 redCar 对象和 blueCar 对象各自访问自己的 weight 变量

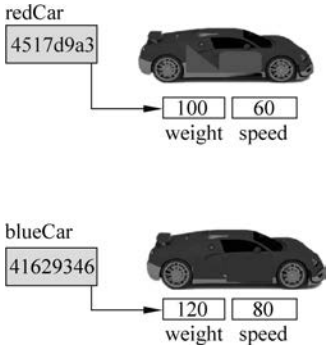

图 1.4 redCar 对象和 blueCar 对象调用 upSpeed()方法

需要注意的是,当用下列代码输出对象中存放的引用时:

```
System.out.println(redCar);
```

得到的结果不完全是引用值，而是一个字符序列 CarSUV@4517d9a3，该字符序列给引用值（十六进制）添加了前缀信息“类名@”。其原因是：

```
System.out.println(redCar);
```

等价于：

```
System.out.println(redCar.toString());
```

public String toString()方法是 Object 类提供的一个方法，CarSUV 类可以重写该方法，使得 toString()方法只返回引用值的字符序列表示。例如在 CarSUV 类中，进行如下的重写：

```
public String toString() {                          //重写 Object 类的方法
    String str = super.toString();
    String backStr = str.substring(str.indexOf("@") + 1);
    return backStr;
}
```

如果需要按整数输出对象的引用（一个整型值，有特殊意义的整数），可以使用 System 类提供的静态方法 identityHashCode()。例如：

```
int addr = System.identityHashCode(redCar);
```

返回的 addr 是 int 型值。

要很好地理解对象的基本结构，即对象（变量）负责存放引用，以确保对象可以操作分配给该对象的变量以及调用类中的方法。不要把对象和分配给对象的变量混淆（分配给对象的变量仅仅是对象的一部分）。避免使用匿名对象，例如，代码

```
new Car().weight = 100;
new Car().weight = 120;
```

的效果是，两个不同的对象（匿名对象）分别设置自己的 weight，而代码

```
redCar.weight = 100;
redCar.weight = 120;
```

的效果是，redCar 对象首先将自己的 weight 设置为 100，然后又重新设置为 120。

注意：在学习面向对象语言的过程中，一个简单的理念是需要完成某种任务时，首先要想到谁去完成任务，即哪个对象去完成任务；提到数据，首先要想到这个数据是哪个对象的。

例 1-1 演示了对象的基本结构。程序运行效果如图 1.5 所示。

```
null
null
4517d9a3
4517d9a3
41629346
41629346
60
60
80
80
```

图 1.5　程序运行效果

例 1-1

CarSUV.java

```
public class CarSUV {
    int speed;
    int weight = 200;
    int upSpeed(int n){
        if(n<=0||n>=260) {
           return speed;
        }
        speed += n;
        return speed>=260?260:speed;
    }
    public String toString() {       //重写 Object 类的方法
        String str = super.toString();
        String backStr = str.substring(str.indexOf("@")+1);
        return backStr;
    }
}
```

Example1_1.java

```
public class Example1_1 {
    public static void main(String args[]) {
         CarSUV redCar = null;
         CarSUV blueCar = null;
         System.out.println(redCar);
         System.out.println(blueCar);
         redCar = new CarSUV();
         blueCar = new CarSUV();
         System.out.println(redCar);
         int addr = System.identityHashCode(redCar);
         System.out.printf("%x\n",addr);
         System.out.println(blueCar);
         addr = System.identityHashCode(blueCar);
         System.out.printf("%x\n",addr);
         redCar.weight = 100;
         blueCar.weight = 120;
         int m = redCar.upSpeed(60);
         int n = blueCar.upSpeed(80);
         System.out.println(m);
         System.out.println(redCar.speed);
         System.out.println(n);
         System.out.println(blueCar.speed);
     }
}
```

1.2 具有相同引用的对象

没有实体(没有被分配变量)的对象称作空对象。程序要避免让一个空对象去调用方法产生行为,否则在运行时会出现 NullPointerException 异常。由于对象可以被动态地分配变量

(实体),所以 Java 编译器对空对象不做检查(不会出现编译错误)。因此,在编写程序时要避免使用空对象。

一个类声明的两个对象如果具有相同的引用,二者就具有完全相同的变量(实体)。当程序用一个类为两个对象 object1 和 object2 分配变量后,二者的引用是不同的,如图 1.6 所示。

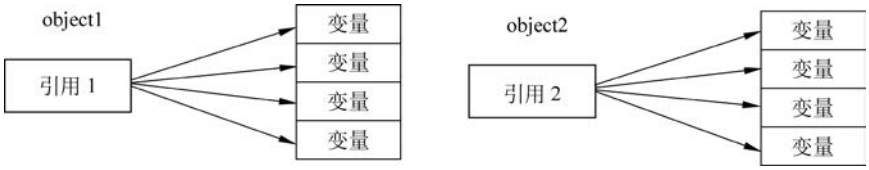

图 1.6　具有不同引用的对象

在 Java 中,对于同一个类的两个对象 object1 和 object2,允许进行如下的赋值操作:

```
object2 = object1;
```

这样,object2 中存放的是 object1 的值,即 object1 对象的引用,因此,object2 所拥有的变量(实体)就和 object1 完全一样了,如图 1.7 所示。

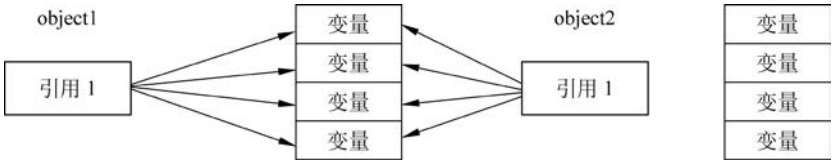

图 1.7　具有相同引用的对象

Java 有所谓的“垃圾收集”机制,这种机制周期地检测某个实体是否已不再被任何对象所拥有(引用),如果发现这样的实体,就释放实体占有的内存。

例 1-2 演示了如果类声明的两个对象具有相同的引用,二者就具有完全相同的变量。程序运行效果如图 1.8 所示。

```
p1的引用:Point@3a71f4dd
p2的引用:Point@7adf9f5f
p1的x,y坐标:1111,2222
p2的x,y坐标:-100,-200
将p2的引用赋给p1后:
p1的引用:7adf9f5f
p2的引用:7adf9f5f
p1的x,y坐标:-100,-200
p2的x,y坐标:-100,-200
```

图 1.8　程序运行效果

例 1-2

Example1_2.java

```
class Point {
    int x,y;
    void setXY(int m,int n){
        x = m;
        y = n;
    }
}
public class Example1_2 {
   public static void main(String args[]) {
      Point p1 = null,p2 = null;
      p1 = new Point();
      p2 = new Point();
      System.out.println("p1 的引用:" + p1);
```

```
        System.out.println("p2 的引用:" + p2);
        p1.setXY(1111,2222);
        p2.setXY( - 100, - 200);
        System.out.println("p1 的 x,y 坐标:" + p1.x + "," + p1.y);
        System.out.println("p2 的 x,y 坐标:" + p2.x + "," + p2.y);
        p1 = p2;                    //使得对象 p1 和 p2 的引用相同
        System.out.println("将 p2 的引用赋给 p1 后: ");
        int address = System.identityHashCode(p1);
        System.out.printf("p1 的引用: % x\n",address);
        address = System.identityHashCode(p2);
        System.out.printf("p2 的引用: % x\n",address);
        System.out.println("p1 的 x,y 坐标:" + p1.x + "," + p1.y);
        System.out.println("p2 的 x,y 坐标:" + p2.x + "," + p2.y);
    }
}
```

1.3 上转型对象

我们经常说“老虎是动物”“狗是动物”等。若动物类是老虎类的父类,这样说当然正确,因为人们习惯地称子类与父类的关系是“is-a”关系。但需要注意的是,当说老虎是动物时,老虎将失掉老虎独有的属性和功能。从人的思维方式来看,说“老虎是动物”属于上溯思维方式,这种思维方式类似 Java 语言中的上转型对象。

假设 Animal 类是 Tiger 类的父类(或间接父类),当用子类创建一个对象,并把这个对象的引用赋值到父类声明的对象中时,例如:

```
Animal animal;
animal = new Tiger();
```

或

```
Animal animal;
Tiger tiger = new Tiger();
animal = tiger;
```

称 animal 是对象 tiger 的上转型对象(好比说“老虎是动物”)。

对象的上转型对象的实体是子类负责创建的,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)。上转型对象操作子类继承的方法或子类重写的实例方法,其作用等价于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实例方法,当对象的上转型对象调用这个实例方法时,一定是调用了子类重写的实例方法。

上转型对象的实体本质上是子类分配的实体,只是少了一部分实体而已,因此,对于

```
Animal animal;
animal = new Tiger();
```

下列表达式的值是 true:

```
animal instanceof Tiger
```

注意：如果子类重写了父类的静态方法，那么子类对象的上转型对象不能调用子类重写的静态方法，只能调用父类的静态方法。

例 1-3 中使用了上转型对象。程序运行效果如图 1.9 所示。

```
这个Dog@31befd9f是狗吗?true
wang wang.....
这个Dog@31befd9f是猫吗?false
这个Cat@1fb3ebeb是猫吗?true
miao miao.....
```

图 1.9　程序运行效果

例 1-3

Example1_3.java

```
class Animal {
    void cry() {
        System.out.println("动物的叫声是怎样的?");
    }
}
class Dog extends Animal {
    void cry() {
        System.out.println("wang wang.....");
    }
}
class Cat extends Animal {
    void cry() {
        System.out.println("miao miao.....");
    }
}
public class Example1_3 {
    public static void main(String args[]) {
        Animal animal;
        animal = new Dog();
        boolean isOk = animal instanceof Dog;
        System.out.println("这个" + animal + "是狗吗?" + isOk);
        animal.cry();
        isOk = animal instanceof Cat;
        System.out.println("这个" + animal + "是猫吗?" + isOk);
        animal = new Cat();
        isOk = animal instanceof Cat;
        System.out.println("这个" + animal + "是猫吗?" + isOk);
        animal.cry();
    }
}
```

第 2 章 抽象类与接口

2.1 抽象类

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

```
abstract class A {
 …
}
```

用关键字 abstract 修饰的方法称为抽象方法(abstract 方法),例如:

```
abstract int min(int x,int y);
```

对于抽象方法,只允许声明,不允许实现(没有方法体),而且不允许使用 final 和 abstract 同时修饰一个方法或类,也不允许使用 static 和 private 修饰 abstract 方法,即抽象方法必须是非 private 的实例方法(访问权限必须高于 private)。

1. 抽象类中的抽象方法

和普通类(非抽象类)相比,抽象类中可以有抽象方法(非抽象类中不可以有抽象方法),也可以有非抽象方法。

下面的 A 类中的 min()方法是抽象方法,max()方法是普通方法(非抽象方法)。

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

注意:抽象类中也可以没有抽象方法。

2. 抽象类不能用 new 运算符创建对象

对于抽象类,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 同时修饰一个方法或类的原因。

3. 抽象类的子类

如果一个非抽象类是抽象类的子类,它必须重写父类的抽象方法,即去掉抽象方法的 abstract 修饰,并给出方法体。如果一个抽象类是抽象类的子类,它可以重写父类的抽象方法,也可以继承父类的抽象方法。

4. 抽象类与上转型对象

尽管抽象类不能使用 new 运算符创建对象,但抽象类声明的对象可以作为子类对象的上

转型对象，那么该对象就可以调用子类重写的方法。

5. 理解抽象类

理解抽象类的意义非常重要。理解抽象类的关键点是：

(1) 抽象类可以抽象出重要的行为标准，该行为标准用抽象方法来表示。即抽象类封装了子类必须有的行为标准。

(2) 抽象类声明的对象可以成为其子类的对象的上转型对象，调用子类重写的方法，即体现子类根据抽象类中的行为标准给出的具体行为。

人们已经习惯给别人介绍数量标准，例如，在介绍人的时候，可以说，人的身高是多少，体重是多少，但是学习了类以后，也要习惯介绍行为标准。所谓行为标准，仅仅是方法的名字、方法的类型而已。例如，人具有 run()行为，或 speak()行为，但仅仅说出行为标准，不要说出 speak()行为的具体体现，即不要说 speak()行为是用英语说话还是用中文说话，这样的行为标准就是抽象方法(没有方法体的方法)。这样一来，开发者可以把主要精力放在一个应用中需要哪些行为标准(不用关心行为的细节)，不仅节省时间，而且非常有利于设计出易维护、易扩展的程序(见后面的设计模式相关的章节，例如第 7 章，策略模式)。抽象类中的抽象方法可以由子类实现，即行为标准的实现由子类完成。

一个男孩要找女朋友，他可以提出一些行为标准，例如，女朋友具有 speak()和 cooking()行为，但可以不给出 speak()和 cooking()行为的细节。例 2-1 使用抽象类封装了男孩对女朋友的行为要求，即封装了他要找的任何具体女朋友都应该具有的行为。程序运行效果如图 2.1 所示。

```
你好
水煮鱼
hello
roast beef
```

图 2.1 程序运行效果

例 2-1

Example2_1.java

```
abstract class GirlFriend {                          //抽象类,封装了两个行为标准
    abstract void speak();
    abstract void cooking();
}
class ChinaGirlFriend extends GirlFriend {
    void speak(){
        System.out.println("你好");
    }
    void cooking(){
        System.out.println("水煮鱼");
    }
}
class AmericanGirlFriend extends GirlFriend {
    void speak(){
        System.out.println("hello");
    }
    void cooking(){
        System.out.println("roast beef");
    }
}
class Boy {
    GirlFriend friend;
```

```
    void setGirlfriend(GirlFriend f){
        friend = f;
    }
    void showGirlFriend() {
       friend.speak();
       friend.cooking();
    }
}
public class Example2_1 {
    public static void main(String args[]) {
        GirlFriend girl = new ChinaGirlFriend();        //girl 是上转型对象
        Boy boy = new Boy();
        boy.setGirlfriend(girl);
        boy.showGirlFriend();
        girl = new AmericanGirlFriend();                //girl 是上转型对象
        boy.setGirlfriend(girl);
        boy.showGirlFriend();
    }
}
```

2.2 接口

使用关键字 interface 定义一个接口。接口的定义和类的定义很相似,分为接口声明和接口体,例如:

```
interface Com {
   ...
}
```

1. 接口声明

定义接口包含接口声明和接口体,和类不同的是,定义接口时使用关键字 interface 来声明自己是一个接口,格式如下:

```
interface 接口的名字
```

2. 接口体

1) 接口体中的抽象方法和常量

接口中可以有抽象方法(在 JDK 8 版本之前,接口体中只可以有抽象方法),接口中所有的抽象方法的访问权限一定都是 public,而且允许省略抽象方法的 public 和 abstract 修饰符。接口体中所有的 static 常量的访问权限一定都是 public,而且允许省略 public、final 和 static 修饰符,因此,接口中不会有变量。例如:

```
interface Com {
    public static final int MAX = 100;                  //等价写法: int MAX = 100;
    public abstract void add();                         //等价写法: void add();
    public abstract float sum(float x ,float y);
     //等价写法: float sum(float x ,float y);
}
```

2）接口体中的 default 实例方法

从 JDK 8 版本开始，允许使用 default 关键字、在接口体中定义称作 default 的实例方法（不可以定义 default 的 static 方法）。和普通的实例方法相比，default 的实例方法是用关键字 default 修饰的带方法体的实例方法。default 实例方法的访问权限必须是 public（允许省略 public 修饰符）。例如，下列接口中的 max 方法就是 default 实例方法：

```
interface Com {
    public final int MAX = 100;
    public abstract void add();
    public abstract float sum(float x ,float y);
    public default int max(int a,int b) {          //default 方法
        return a > b?a:b;
    }
}
```

注意：不可以省略 default 关键字，因为接口里不允许定义通常的带方法体的 public 实例方法。

3）接口体中的静态（static）方法

从 JDK 8 版本开始，允许在接口体中定义静态方法。例如，下列接口中的 f()方法就是静态方法：

```
public interface Com {
    public static final int MAX = 100;
    public abstract void on();
    public abstract float sum(float x ,float y);
    public default int max(int a,int b) {
        return a > b?a:b;
    }
    public static void f() {          //static 方法
        System.out.println("注意是从 JDK 8 开始的");
    }
}
```

4）接口体中的私有（private）方法

从 JDK 9 版本开始，允许在接口体中定义私有的方法，其目的是配合接口中的 default 的实例方法，即接口可以将某些算法封装在私有的方法中，供接口中的 default 的实例方法调用，实现算法的复用。

3. 实现接口

在 Java 语言中，接口由类实现以便使用接口中的方法。一个类需要在类声明中使用关键字 implements 声明该类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。例如，A 类实现 Com 和 Addable 两个接口：

```
class A implements Com,Addable
```

再如，Animal 的 Dog 子类实现 Eatable 和 Sleepable 接口：

```
class Dog extends Animal implements Eatable,Sleepable
```

如果一个类实现了某个接口,那么这个类就自然拥有了接口中的常量和 default 关键字修饰的方法(但去掉了 default 关键字),该类也可以重写接口中 default 修饰的方法(注意,重写时需要去掉 default 关键字)。如果一个非抽象类实现了某个接口,那么这个类必须重写该接口的所有抽象方法,即去掉修饰方法的 abstract 关键字,给出方法体。如果一个抽象类实现了某个接口,该类可以选择重写接口的抽象方法或直接用接口的抽象方法。

需要特别注意的是,类实现接口,但类并不拥有接口的静态方法和私有方法。接口中除了私有方法,其他方法的访问权限默认都是 public 的,重写时不可省略 public(否则就降低了访问权限,这是不允许的)。

实现接口的非抽象类一定要重写接口的抽象方法,因此也称这个类实现了接口。

4. 接口回调

和类一样,接口也是 Java 中一种重要的数据类型,用接口声明的变量称作接口变量。那么,接口变量中可以存放怎样的数据呢?

接口属于引用型变量,接口变量中可以存放实现该接口的类的实例的引用,即存放对象的引用。例如,假设 Com 是一个接口,那么就可以用 Com 声明一个变量:

```
Com com;
```

其内存模型如图 2.2 所示,称此时的 com 是一个空接口,因为 com 变量中还没有存放实现该接口的类的实例(对象)的引用。

假设 ImpleCom 类是实现 Com 接口的类,用 ImpleCom 创建名字为 object 的对象:

```
ImpleCom object = new ImpleCom();
```

那么 object 对象不仅可以调用 ImpleCom 类中原有的方法,而且可以调用 ImpleCom 类实现的接口方法,如图 2.3 所示。

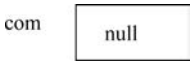

图 2.2　空接口

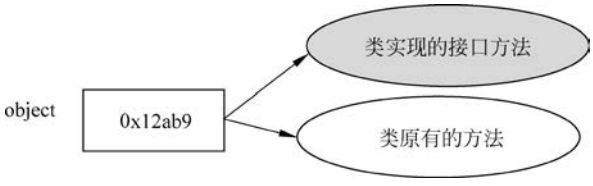

图 2.3　对象调用方法的内存模型

“接口回调”一词是借用了 C 语言中指针回调的术语,表示一个变量的地址在某一个时刻存放在一个指针变量中,那么指针变量就可以间接操作该变量中存放的数据。

在 Java 语言中,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值给该接口声明的接口变量,那么该接口变量就可以调用被类实现的接口方法以及接口提供的 default 方法或类重写的 default 方法。实际上,当接口变量调用被类实现的接口方法时,就是通知相应的对象调用这个方法。

例如,将上述 object 对象的引用赋值给 com 接口:

```
com = object;
```

那么内存模型如图 2.4 所示,箭头示意接口 com 变量可以调用类实现的接口方法(这一过程

称为接口回调)。但是,接口无法调用类中的其他的非接口方法。

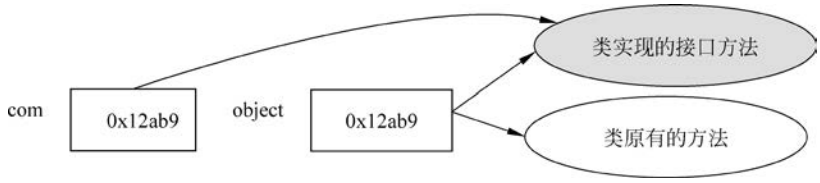

图 2.4 接口回调的内存模型

接口回调非常类似于上转型对象调用子类重写的方法(见 1.3 节)。

注意:如果接口 com 中存放了实现该接口的 ImpleCom 类的对象的引用,那么表达式 com instanceof ImpleCom 的值是 true。

例 2-2 使用了接口的回调技术,程序运行效果如图 2.5 所示。

```
true
11.23和22.78的算术平均值:17.01
false
true
11.23和22.78的几何平均值:15.99
```

图 2.5 程序运行效果

例 2-2

Example2_2.java

```
interface ComputerAverage {
    public double average(double a,double b);
}
class A implements ComputerAverage {
    public double average(double a,double b) {
        double aver = 0;
        aver = (a+b)/2;
        return aver;
    }
}
class B implements ComputerAverage {
    public double average(double a,double b) {
        double aver = 0;
        aver = Math.sqrt(a*b);
        return aver;
    }
}
public class Example2_2 {
    public static void main(String args[]) {
        ComputerAverage computer;
        double a = 11.23,b = 22.78;
        computer = new A();
        System.out.println(computer instanceof A);
        double result = computer.average(a,b);                    //接口回调 average()方法
        System.out.printf("%5.2f 和 %5.2f 的算术平均值:%5.2f\n",a,b,result);
```

```
            computer = new B();
            System.out.println(computer instanceof A);
            System.out.println(computer instanceof B);
            result = computer.average(a,b);            //接口回调 average()方法
            System.out.printf("%5.2f 和 %5.2f 的几何平均值:%5.2f",a,b,result);
        }
    }
```

2.3 抽象类与接口的比较

抽象类和接口都可以有抽象方法。接口中只可以有常量,不可以有变量;而抽象类中既可以有常量,也可以有变量。抽象类中可以有非抽象的,且不能用 default 关键字修饰的实例方法;而接口中不可以有非抽象的,不用 default 关键字修饰的 public 实例方法。

在设计程序时应当根据具体的分析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重写的抽象方法外,还提供子类可以继承的变量和非抽象方法。如果某个问题需要使用继承才能更好地解决,例如,子类除了需要重写父类的抽象方法,还需要从父类继承一些变量或一些重要的非抽象方法,就可以考虑用抽象类;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抽象方法的实现细节,就可以考虑使用接口。

第3章 组合

在实际生活中经常能见到对象组合的例子，例如一个公司有若干职员，一个房屋有若干家具等。组合是面向对象中的一个重要手段，通过组合，可以让对象之间进行必要的交互。

3.1 对象的组合

类的成员变量可以是 Java 允许的任何数据类型，因此，一个类可以把对象作为自己的成员变量。如果用这样的类创建对象，那么该对象中就会有其他对象，也就是说，该对象将其他对象作为自己的组成部分（这就是人们常说的 Has-A）。一个对象 a 通过组合对象 b 来复用对象 b 的方法，即对象 a 委托对象 b 调用其方法。当前对象随时可以更换所组合的对象，使得当前对象与所组合的对象是弱耦合关系。

现在，对圆锥体做一个抽象：

- 属性：底圆，高
- 操作：计算体积

那么，圆锥体的底圆应当是一个对象，例如 Circle 类声明的对象；圆锥体的高可以是 double 型的变量，即圆锥体将 Circle 类的对象作为自己的成员。

例 3-1 中，Circular 类负责创建“圆锥体”对象；Application.java 是主类。在主类的 main 方法中使用 Circle 类创建一个“圆”对象 circle，使用 Circular 类创建一个“圆锥”对象 circular，然后 circular 调用 setBottom(Circle c)方法将 circle 的引用传递给 circular 的成员变量 bottom，即让 circular 组合 circle。程序运行效果如图 3.1 所示。

```
圆锥的体积:69708.000
```

图 3.1 程序运行效果

例 3-1

Circle.java

```
public class Circle {
    double radius;
    double getArea() {
        double area = 3.14 * radius * radius;
        return area;
    }
}
```

Circular.java

```
public class Circular {                    //圆锥类
    Circle bottom;
    double height;
```

```
    void setBottom(Circle c) {
        bottom = c;
    }
    void setHeight(double h) {
        height = h;
    }
    double getVolume() {
        return bottom.getArea() * height/3.0;
    }
}
```

Application.java

```
public class Application {                          //主类
   public static void main(String args[]) {
       Circle circle = new Circle();
       circle.radius = 100;
       Circular circular = new Circular();
       circular.setBottom(circle);                  //将 circle 的引用传递给 circular 的 bottom
       circular.setHeight(6.66);
       System.out.printf("圆锥的体积:%5.3f\n",circular.getVolume());
    }
}
```

在上述 Application 类中,当执行代码

```
Circle circle = new Circle(100);
```

后,内存中诞生了一个 circle(圆)对象,circle 的 radius(半径)是 100。内存中对象的模型如图 3.2 所示。

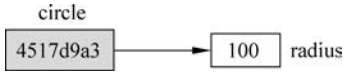

图 3.2 circle(圆)对象

当执行代码

```
Circular circular = new Circular(circle,6.66);
```

后,内存中又诞生了一个 circular 对象(圆锥),然后执行代码

```
circular.setBottom(circle);
```

将 circle 对象的引用传递给 circular 对象的 bottom(底),因此,bottom 对象和 circle 对象就有同样的实体(radius)。内存中对象的模型如图 3.3 所示。

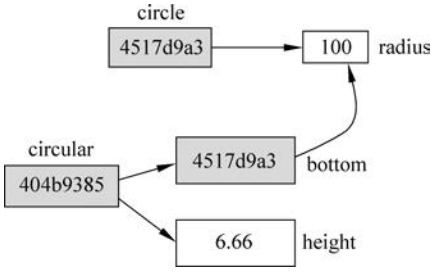

图 3.3 circular(圆锥)对象

3.2 组合关系是弱耦合关系

子类与父类的关系是强耦合关系，父子关系体现在类层次上，而不是对象层次上。组合关系最终体现在对象层次上，即一个对象将另一个对象作为自己的一部分。面向对象将对象与对象之间的组合关系归为弱耦合关系的主要原因有以下两点：

- 如果修改当前对象所组合的对象的类的代码，不必修改当前对象的类的代码。
- 当前对象可以在运行时指定所包含的对象。

公司和职员的关系是组合关系，一个公司可以根据公司的需求聘用或解聘某个职员。例如，Employee 是描述职员的类，Corp 是描述公司的类，让 Corp 和 Employee 类形成组合关系。那么，Corp 类的对象可以在运行时指定所包含的 Employee 子类的对象，即 Corp 类的实例可以调用 setEmployee(Employee em)方法在它的 employee 变量中存放任何 Employee 子类的对象的引用。具体代码见例 3-2，运行效果如图 3.4 所示。

```
本公司当前聘用的职员信息:
男性职员:周.
男性职员:吴.
女性职员:杨.
本公司当前聘用的职员信息:
男性职员:赵.
女性职员:孙.
```

图 3.4 程序运行效果

例 3-2

Employee.java

```
public abstract class Employee{
    String name;
    public abstract void showMess();
}
```

MaleEmployee.java

```
public class MaleEmployee extends Employee{
    public MaleEmployee(String name) {
        this.name = name;
    }
    public void showMess() {
        System.out.println("男性职员:" + name + ".");
    }
}
```

FemaleEmployee.java

```
public class FemaleEmployee extends Employee {
    public FemaleEmployee(String name) {
        this.name = name;
    }
    public void showMess() {
        System.out.println("女性职员:" + name + ".");
    }
}
```

Corp.java

```
public class Corp {
    Employee employee[];
```

```
    public void setEmployee(Employee ...em) {
           employee = new Employee[em.length];
           for(int i = 0;i < em.length;i++)
               employee[i] = em[i];
    }
    public void outEmployeeMess() {
        System.out.println("本公司当前聘用的职员信息:");
        for(int i = 0;i < employee.length;i++)
            employee[i].showMess();
    }
}
```

Application.java

```
public class Application {
  public static void main(String args[]){
      Corp corpSun = new Corp();
      corpSun.setEmployee
             (new MaleEmployee("周"),new MaleEmployee("吴"),new FemaleEmployee("杨"));
      corpSun.outEmployeeMess();
      corpSun.setEmployee
             (new MaleEmployee("赵"),new FemaleEmployee("孙"));
      corpSun.outEmployeeMess();
  }
}
```

3.3 基于组合的击鼓传花

击鼓传花是很多人玩过的游戏。一些人(不少于2人)围成圆圈,其中一人拿花,圈外一人背对着大家敲鼓。鼓响时众人开始依次传花,至鼓声停止为止。此时花在谁手中,谁就从圈中离开(实际娱乐中,常常要求出圈人上台表演节目);然后鼓声再次响起,圈中剩余的人继续依次传花。击鼓传花游戏如图3.5所示。

参与击鼓传花的人必须组合他的下一位,以便把花传给下一位;同时也必须组合上一位,以便自己退出时通知上一位,他的下一位发生了变化,见例3-3中的Person类。Flower类的对象负责在0,1,2,3,4,5,6,7中产生一个随机数,当花在某人手中,而此时Flower对象产生的随机数刚好是6(模拟鼓声停止),此人从圈中退出。程序运行效果如图3.6所示。

图3.5 击鼓传花游戏

```
代号7的人退出击鼓传花
代号8的人退出击鼓传花
代号2的人退出击鼓传花
代号5的人退出击鼓传花
代号6的人退出击鼓传花
代号9的人退出击鼓传花
代号10的人退出击鼓传花
代号3的人退出击鼓传花
代号12的人退出击鼓传花
代号4的人退出击鼓传花
代号1的人退出击鼓传花
代号11是最后一个人
```

图3.6 程序运行效果

例 3-3

Person. java

```
public class Person {
    int personNumber;                                   //代号
    Flower flowerInhand;                                //手中拿的花
    Person previousPerson;                              //组合传我花的人
    Person nextPerson;                                  //组合接我花的人
    public void setFlower(Flower flower){
        flowerInhand = flower;
        int n = flowerInhand.getRamdomNumber();
        if(n == 6) {
            previousPerson.setNextPerson(nextPerson);
            nextPerson.setPreviousPerson(previousPerson);
           try{ Thread.sleep(1000);
           }
           catch(Exception exp){}
           System.out.println("代号" + personNumber + "的人退出击鼓传花");

        }
        if(this == nextPerson){
           System.out.println("代号" + personNumber + "是最后一个人");
           return;
        }
        nextPerson.setFlower(flowerInhand );        //传花给下一位
    }
    public void setPreviousPerson(Person person){
        previousPerson = person;
    }
    public void setNextPerson(Person person){
        nextPerson = person;
    }
}
```

Flower. java

```
import java.util.Random;                            //负责产生随机数的 Random 类
public class Flower {
     Random random;                                 //负责产生随机数
     Flower(){
         random = new Random();
     }
     public int getRamdomNumber(){
        int number = random.nextInt(8);             //返回 0 到 7 中的某个数
        return number + 1;
     }
}
```

Application. java

```
public class Application {
    public static void main(String args[]) {
```

```
        Flower roseFlower = new Flower();
        int length = 12;
        if(length < 2)
           return;
        Person [ ] personInCircle = new Person[length];
        for(int i = 0;i < personInCircle.length;i++){
            personInCircle[i] = new Person();
            personInCircle[i].personNumber = i + 1;
        }
        for(int i = 0;i < personInCircle.length;i++){
           if(i == 0){
                personInCircle[i].setNextPerson(personInCircle[i + 1]);
                personInCircle[i].setPreviousPerson(personInCircle[length - 1]);
           }
           else if(i == length - 1){
                personInCircle[i].setNextPerson(personInCircle[0]);
                personInCircle[i].setPreviousPerson(personInCircle[i - 1]);
           }
           else {
                personInCircle[i].setNextPerson(personInCircle[i + 1]);
                personInCircle[i].setPreviousPerson(personInCircle[i - 1]);
           }
        }
        personInCircle[0].setFlower(roseFlower);
    }
}
```