第 5 章 类与对象


视频讲解

主要内容

- 面向对象的特性；
- 类；
- 构造方法与对象的创建；
- 参数传值；
- 对象的组合；
- 实例成员与类成员；
- 方法重载与多态；
- this 关键字；
- 包；
- import 语句；
- 访问权限。

面向对象语言有三个重要特性：封装、继承和多态。学习面向对象编程要掌握怎样通过抽象得到类，继而学习怎样编写类的子类来体现继承和多态。本章主要讲述类和对象，即学习面向对象的第一个特性——封装，第 6 章学习面向对象的另外两个特性——继承和多态。


视频讲解

5.1 面向对象的特性

随着计算机硬件设备功能的进一步提高，面向对象的编程成为可能。面向对象的编程更加符合人的思维模式，编写的程序更加健壮和强大，更重要的是，面向对象编程鼓励创造性的程序设计。面向对象编程是一种先进的编程思想，更加容易解决复杂的问题。面向对象编程主要具有下面三个特性。

1. 封装性

面向对象编程的核心思想之一是将数据和对数据的操作封装在一起。通过抽象，即从具体的实例中抽取共同的性质，形成一般的概念，例如类的概念。

在实际生活中，我们每时每刻都在与“对象”打交道，例如我们用的钢笔，骑的自行车，乘的公共汽车等。而我们经常见到的卡车、公共汽车、轿车等都会涉及以下几个重要的物理量：可乘载的人数、运行速度、发动机的功率、耗油量、自重、轮子数目等；另外，还有几个重要的功能：加速功能、减速功能、刹车、转弯功能等。可以把这些功能称作它们具有的方法，而物理量是它们的状态描述，仅仅用物理量或功能不能很好地描述它们。在现实生活中，对这些共有的属性和功能给出一个概念——机动车类。也就是说，人们经常谈到的机动车类就是从具体的实例中抽取共同的属性和功能形成的一个概念，那么一个具体的轿车就是机动车类的一个实例，即对象。一个对象将自己的数据和对这些数据的操作合理有效地封装在一起，例如，每辆

轿车调用“加大油门”改变的都是自己的运行速度。

2. 继承性

继承体现了一种先进的编程模式。子类可以继承父类的属性和功能，即继承了父类具有的数据和数据上的操作，同时又可以增添子类独有的数据和数据上的操作。例如，“人类”自然继承了“哺乳类”的属性和功能，同时又增添了人类独有的属性和功能。

3. 多态性

多态是面向对象编程的又一重要特征。有两种意义的多态。一种多态是操作名称的多态，即有多个操作具有相同的名字，但这些操作所接收的消息类型必须不同。例如，让一个人执行“求面积”操作时，他可能会问你求什么面积。所谓操作名称的多态，是指可以向操作传递不同消息，以便让对象根据相应的消息来产生一定的行为。另一种多态是和继承有关的多态，是指同一个操作被不同类型对象调用时可能产生不同的行为。例如，狗和猫都具有哺乳类的功能——“喊叫”，当狗操作“喊叫”时产生的声音是“汪汪…… ”，而当猫操作“喊叫”时产生的声音是“喵喵……”。

视频讲解

5.2 类

类是组成 Java 程序的基本要素，一个 Java 应用程序就是由若干个类构成的(见第 2 章)。类是 Java 语言中最重要的“数据类型”，类声明的变量被称作对象，即类是用来创建对象的“模板”。

类的定义包括两部分：类声明和类体。基本格式为：

```
class 类名{
   类体的内容
}
```

class 是关键字，用来定义类。“class 类名”是类的声明部分，类名必须是合法的 Java 标识符。两个大括号以及之间的内容是类体。

▶ 5.2.1 类声明

以下是两个类声明的例子。

```
class People {
   …
 }
class 植物{
   …
 }
```

“class People”和“class 植物”叫作类声明；“People”和“植物”分别是类名。类的名字要符合标识符规定，即名字可以由字母、下画线、数字或美元符号组成，并且第一个字符不能是数字(这是语法要求的)。给类命名时，遵守下列编程风格(这不是语法要求的，但应当遵守)。

(1) 如果类名使用拉丁字母，那么名字的首字母使用大写字母，如 Hello、Time 等。

(2) 类名最好容易识别、见名知义。当类名由几个单词复合而成时，每个单词的首字母大写，如 ChinaMade、AmericanVehicle、HelloChina 等。

▶ 5.2.2 类体

定义类的目的是根据抽象描述一类事物共有的属性和功能，即给出用于创建具体实例(对象)的一种数据类型，描述过程由类体实现。类声明之后的一对大括号“{”、“}”以及它们之间的内容称作类体，大括号之间的内容称作类体的内容。

抽象的关键是抓住事物的两个方面：属性和功能，即数据以及在数据上进行的操作。因此，类体的内容由两部分构成。

- 变量的声明：用来描述数据(体现对象的属性)。
- 方法：方法可以对类中声明的变量进行操作，即给出算法(体现对象具有的功能)。

下面是一个类名为 Ladder 的类(用来描述梯形)，类体内容的变量定义部分定义了 4 个 float 类型变量：above、bottom、height 和 area，方法定义部分定义了两个方法：computerArea 和 setHeight。

```
class Ladder {
    float above;                          //梯形的上底(变量声明)
    float bottom;                         //梯形的下底(变量声明)
    float height;                         //梯形的高(变量声明)
    float area;                           //梯形的面积(变量声明)
    float computerArea() {                //计算面积(方法)
       area = (above + bottom) * height/2.0f;
       return area;
    }
    void setHeight(float h) {             //修改高(方法)
       height = h;
    }
}
```

▶ 5.2.3 成员变量

类体分为两部分：一部分是变量的声明；另一部分是方法的定义。变量声明部分声明的变量被称作域变量或成员变量。

1. 成员变量的类型

成员变量的类型可以是 Java 中的任何一种数据类型，包括基本类型：整型、浮点型、字符型；引用类型：数组、对象和接口(对象和接口见后续内容)。例如：

```
class Factory {
   float a[];
   Workman zhang;
}
class Workman {
  double x;
}
```

Factory 类的成员变量 a 是类型为 float 的数组，zhang 是 Workman 类声明的变量，即对象。

2. 成员变量的有效范围

成员变量在整个类的所有方法中都有效，其有效性与它在类体中书写的先后位置无关。

例如，前述的 Ladder 类也可以等价地写成：

```
class Ladder {
    float above;                          //梯形的上底(变量声明)
    float area ;                          //梯形的面积(变量声明)
    float computerArea() {                //计算面积(方法)
       area = (above + bottom) * height/2.0f;
       return area;
    }
    float bottom ;                        //梯形的下底(变量声明)
    void setHeight(float h) {             //修改高(定义)
       height = h;
    }
    float height;                         //梯形的高(变量声明)
}
```

不提倡把成员变量的定义分散地写在方法之间或类体的最后，人们习惯先介绍属性再介绍功能。

3. 编程风格

(1) 一行只声明一个变量。我们已经知道，尽管可以使用一种数据的类型、用逗号分隔来声明若干个变量，例如：

```
float above,bottom;
```

但是在编码时却不提倡这样做(本书中某些代码可能没有严格遵守这个风格，是为了减少代码行数，降低书的成本)，其原因是不利于给代码增添注释内容。提倡的风格是：

```
float above;                              //梯形上底
float bottom;                             //梯形下底
```

(2) 变量的名字除了符合标识符规定外，名字的首单词的首字母使用小写；如果变量的名字由多个单词组成，从第 2 个单词开始的其他单词的首字母使用大写。

(3) 变量的名字应见名知义，避免使用诸如 m1、n1 等作为变量的名字，尤其是名字中不要将小写的英文字母 l 和数字 1 相邻接，人们很难区分“l1”和“ll”。

▶ 5.2.4 方法

我们已经知道一个类的类体由两部分组成：变量的声明和方法的定义。方法的定义包括两部分：方法声明和方法体。一般格式为：

```
方法声明部分{
    方法体的内容
}
```

1. 方法声明

最基本的方法声明包括方法名和方法的返回类型，例如：

```
double getSpeed() {
    return speed;
}
```

根据程序的需要，方法返回的数据的类型可以是 Java 中的任何一种数据类型；当一个方

法不需要返回数据时，返回类型必须是 void。很多的方法声明中都给出方法的参数，参数是用逗号隔开的一些变量声明。方法的参数可以是任意的 Java 数据类型。

方法的名字必须符合标识符规定，给方法起名字的习惯和给变量起名字的习惯类似。例如，如果名字使用拉丁字母，首字母小写；如果名字由多个单词组成，从第 2 个单词开始的其他单词的首字母大写。

2. 方法体

方法声明之后的一对大括号“{”、“}”以及之间的内容称作方法的方法体。方法体的内容包括局部变量的声明和 Java 语句，即在方法体内可以对成员变量和该方法体中声明的局部变量进行操作。在方法体中声明的变量和方法的参数被称作局部变量，例如：

```
int getSum(int n) {                          // 参数变量 n 是局部变量
    int sum = 0;                             // 声明局部变量 sum
    for(int i = 1;i <= n;i++) {              // for 循环语句
        sum = sum + i;
    }
    return sum;                              // return 语句
}
```

和类的成员变量不同的是，局部变量只在声明它的方法内有效，而且与其声明的位置有关。方法的参数在整个方法内有效，方法内的局部变量从声明它的位置之后开始有效。如果局部变量的声明是在一条复合语句中，那么该局部变量的有效范围是该复合语句，即仅在该复合语句中有效；如果局部变量的声明是在一个循环语句中，那么该局部变量的有效范围是该循环语句，即仅在该循环语句中有效。例如：

```
public class A {
    int m = 10,sum = 0;                      //成员变量，在整个类中有效
    void f() {
       if(m > 9) {
          int z = 10;                        //z 仅仅在该复合语句中有效
          z = 2 * m + z;
       }
       for(int i = 0;i < m;i++) {
           sum = sum + i;                    // i 仅仅在该循环语句中有效
       }
       m = sum;                              //合法，因为 m 和 sum 有效
       z = i + sum;                          //非法，因为 i 和 z 已无效
    }
}
```

写一个方法和在 C 语言中写一个函数完全类似，只不过在面向对象语言中称作方法，因此如果有比较好的 C 语言基础，编写方法的方法体就不再是难点。

3. 区分成员变量和局部变量

如果局部变量的名字与成员变量的名字相同，那么成员变量被隐藏，即这个成员变量在这个方法内暂时失效。例如：

```
class Tom {
    int x = 10,y;
    void f() {
       int x = 5;
```

```
        y = x + x;      //y 得到的值是 10,不是 20.如果方法 f 中没有"int x = 5;",y 的值将是 20
    }
}
```

如果方法中的局部变量的名字与成员变量的名字相同,那么方法就隐藏了成员变量。如果想在该方法中使用被隐藏的成员变量,必须使用关键字 this(在 5.8 节还会详细讲解 this 关键字)。例如:

```
class Tom {
    int x = 10,y;
    void f() {
        int x = 5;
        y = x + this.x;                    //y 得到的值是 15
    }
}
```

▶ 5.2.5 需要注意的问题

通过前面的学习我们知道,类体的内容由两部分构成:变量的声明和方法的定义。也就是说,对成员变量的操作只能放在方法中,方法可以对成员变量和该方法体中声明的局部变量进行操作。在声明成员变量时可以同时赋予初值,例如:

```
class A {
    int a = 12;
    float b = 12.56f;
}
```

但是不可以这样写:

```
class A {
   int a;
   float b;
   a = 12;                    //非法,这是赋值语句(语句不是变量的声明,只能出现在方法体中)
   b = 12.56f;                //非法
}
```

▶ 5.2.6 类的 UML 图

UML(Unified Modeling Language)图属于结构图,常被用于描述一个系统的静态结构。UML 图通常包含类(Class)的 UML 图、接口(Interface)的 UML 图、泛化关系(Generalization)的 UML 图、关联关系(Association)的 UML 图、依赖关系(Dependency)的 UML 图和实现关系(Realization)的 UML 图。

本节介绍类的 UML 图,后续章节会结合相应的内容介绍其余的 UML 图。图 5.1 是前面 5.2.2 节中 Ladder 类的 UML 图。

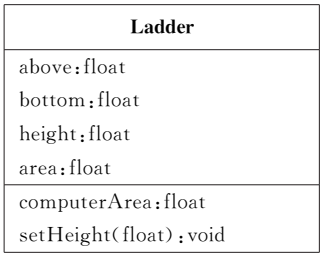

图 5.1 Ladder 类的 UML 图

在类的 UML 图中,使用一个长方形描述一个类的主要构成,将长方形垂直地分为三层。

顶部第 1 层是名字层,如果类的名字是常规字形,表明该类是具体类;如果类的名字是斜体字形,表明该类是抽象类

(抽象类在第 6 章讲述)。

第 2 层是变量层,也称属性层,列出类的成员变量及类型,格式是“变量名字:类型”。在用 UML 图表示类时,可以根据设计的需要只列出最重要的成员变量的名字。

第 3 层是方法层,也称操作层,列出类中的方法,格式是“方法名字(参数列表):类型”。在用 UML 图表示类时,可以根据设计的需要只列出最重要的方法。

视频讲解

5.3 构造方法与对象的创建

类是面向对象语言中最重要的一种数据类型,可以用它来声明变量。在面向对象语言中,用类声明的变量被称作对象。和基本数据类型不同,在用类声明对象后,还必须创建对象,即为声明的对象分配变量(确定对象具有的属性)。当使用一个类创建一个对象时,也称给出了这个类的一个实例。通俗地讲,类是创建对象的“模板”,没有类就没有对象。

构造方法和对象的创建密切相关,下面将详细讲解构造方法和对象的创建。

▶ 5.3.1 构造方法

构造方法是类中的一种特殊方法,当程序用类创建对象时需要使用它的构造方法。类中的构造方法的名字必须与它所在的类的名字完全相同,而且没有类型。允许在一个类中编写若干个构造方法,但必须保证它们的参数不同,即参数的个数不同,或者是参数的类型不同。

需要注意的是,如果类中没有编写构造方法,系统会默认该类只有一个构造方法,该默认的构造方法是无参数的,且方法体中没有语句。例如,5.2.2 节中的 Ladder 类就有一个默认的构造方法。

```
Ladder() {
}
```

如果类中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。例如,下列 Point 类有两个构造方法。

```
class Point {
    int x,y;
    Point() {
        x=1;
        y=1;
    }
    Point(int a,int b) {
       x=a;
       y=b;
    }
}
```

▶ 5.3.2 创建对象

创建一个对象包括两个步骤:对象的声明和为对象分配变量。

1. 对象的声明

一般格式为:

```
类的名字  对象名字;
```

例如：

```
Ladder ladder;
```

2. 为声明的对象分配变量

使用 new 运算符和类的构造方法为声明的对象分配变量，即创建对象。如果类中没有构造方法，系统会调用默认的构造方法，默认的构造方法是无参数的，且方法体中没有语句。

以下是两个详细的例子。

【例 5.1】

Example5_1.java

```
class XiyoujiRenwu {
    float height,weight;
    String head, ear,hand,foot, mouth;
    void speak(String s) {
       System.out.println(s);
    }
}
public class Example5_1 {
    public static void main(String args[]) {
        XiyoujiRenwu  zhubajie;              //声明对象
        zhubajie = new XiyoujiRenwu();       //为对象分配变量(使用 new 和默认的构造方法)
    }
}
```

【例 5.2】

Example5_2.java

```
class Point {
   int x,y;
    Point(int a,int b) {
       x = a;
       y = b;
    }
}
public class Example5_2 {
  public static void main(String args[]) {
     Point p1,p2;                            //声明对象 p1 和 p2
     p1 = new Point(10,10);                  //为对象分配变量(使用 new 和类中的构造方法)
     p2 = new Point(23,35);                  //为对象分配变量(使用 new 和类中的构造方法)
  }
}
```

注：如果类中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。例 5.2 提供了构造方法，下列创建对象是非法的。

```
p1 = new Point();
```

3. 对象的内存模型

使用前面的例 5.1 说明对象的内存模型。

1）声明对象时的内存模型

当用 XiyoujiRenwu 类声明一个变量 zhubajie，即对象 zhubajie 时，如例 5.1 中：

```
XiyoujiRenwu zhubajie;
```

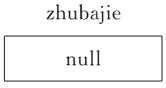

图 5.2　未分配变量的对象

内存模型如图 5.2 所示。

声明对象变量 zhubajie 后，zhubajie 的内存中还没有任何数据，称这时的 zhubajie 是一个空对象。空对象不能使用，因为它还没有得到任何“实体”，必须再进行为对象分配变量的操作，即为对象分配实体。

2）为对象分配变量后的内存模型

当系统见到

```
zhubajie = new XiyoujiRenwu();
```

时，就会做两件事。

（1）为 height、weight、head、ear、mouth、hand、foot 这些变量分配内存，即为 XiyoujiRenwu 类的成员变量分配内存空间，然后执行构造方法中的语句。如果成员变量在声明时没有指定初值，使用的构造方法也没有对成员变量进行初始化操作，那么，对于整型的成员变量，默认初值是 0；对于浮点型，默认初值是 0.0；对于 boolean 型，默认初值是 false；对于引用型，默认初值是 null。

（2）给出一条信息，已确保这些变量是属于对象 zhubajie 的，即这些内存单元将由 zhubajie 操作管理。为了做到这一点，new 运算符在为变量 height、weight、head、ear、mouth、hand、foot 分配内存后，将返回一个引用给对象变量 zhubajie。也就是返回一个“号码”（该号码包含代表这些成员变量内存位置的首地址等重要的有关信息）给 zhubajie，不妨认为这个引用就是 zhubajie 在内存中的名字，而且这个名字（引用）是 Java 系统确保分配给 height、weight、head、ear、mouth、hand、foot 的内存单元，将由 zhubajie 操作管理。称 height、weight、head、ear、mouth、hand、foot 分配的内存单元是属于对象 zhubajie 的实体，即这些变量是属于 zhubajie 的。所谓为对象分配内存，就是指为它分配变量，并获得一个引用，以确保这些变量由它来“操作管理”。为对象分配变量后，内存模型由声明对象时的模型图 5.2 变成如图 5.3 所示，箭头示意对象可以操作这些属于它的变量。

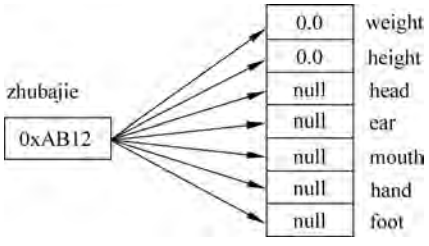

图 5.3　分配变量后的对象

3）创建多个不同的对象

一个类通过使用 new 运算符可以创建多个不同的对象，这些对象将被分配不同的内存空间，因此，改变其中一个对象的状态不会影响其他对象的状态。例如，可以在例 5.1 中创建两个对象：zhubajie、sunwukong。

```
zhubajie = new XiyoujiRenwu();
sunwukong = new XiyoujiRenwu();
```

当创建对象 zhubajie 时，XiyoujiRenwu 类中的成员变量 height、weight、head、ear、mouth、hand、foot 被分配内存空间，并返回一个引用给 zhubajie；当再创建一个对象 sunwukong 时，XiyoujiRenwu 类中的成员变量 height、weight、head、ear、mouth、hand、foot 再一次被分配内存空间，并返回一个引用给 sunwukong。sunwukong 的变量占据的内存空间和 zhubajie 的变量占据的内存空间是互不相同的位置。内存模型如图 5.4 所示。

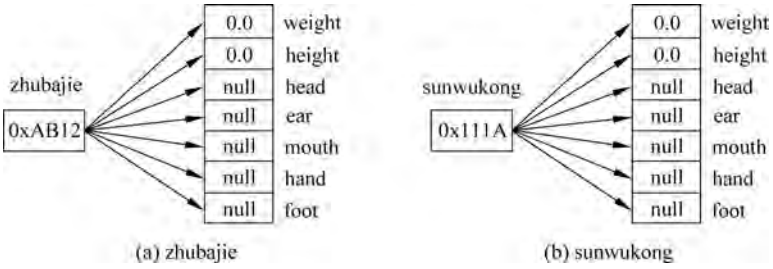

图 5.4 创建多个对象

▶ 5.3.3 使用对象

抽象的目的是产生类，而类的目的是创建具有属性和功能的对象。对象不仅可以操作自己的变量改变状态，而且可以调用类中的方法产生一定的行为。

通过使用运算符“.”，对象可以实现对自己变量的访问和方法的调用。

1. 对象操作自己的变量（改变属性的值）

对象创建之后，就有了自己的变量，即对象的实体。通过使用运算符“.”，对象可以实现对自己的变量的访问。访问格式为：

```
对象.变量;
```

2. 对象调用类中的方法（体现对象的功能）

对象创建之后，可以使用运算符“.”调用创建它的类中的方法，从而产生一定的行为功能。调用格式为：

```
对象.方法;
```

3. 体现封装

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。在讲述类的时候我们讲过类中的方法可以操作成员变量。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。

在例 5.3 中，主类的 main 方法使用 XiyoujiRenwu 创建两个对象：zhubajie、sunwukong，运行效果如图 5.5 所示。

【例 5.3】

Example5_3.java

```
class XiyoujiRenwu {
    float height,weight;
    String head, ear,hand,foot,mouth;
    void speak(String s) {
        head = "歪着头";
```

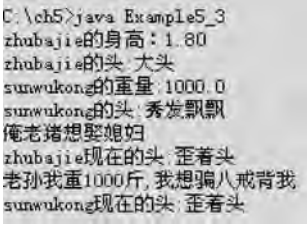

图 5.5 使用对象

```
        System.out.println(s);
    }
}
public class Example5_3 {
    public static void main(String args[]) {
        XiyoujiRenwu  zhubajie,sunwukong;                    //声明对象
        zhubajie = new XiyoujiRenwu();                       //为对象分配变量
        sunwukong = new XiyoujiRenwu();
        zhubajie.height = 1.80f;                             //对象给自己的变量赋值
        zhubajie.head = "大头";
        zhubajie.ear = "一双大耳朵";
        sunwukong.height = 1.62f;                            //对象给自己的变量赋值
        sunwukong.weight = 1000f;
        sunwukong.head = "秀发飘飘";
        System.out.println("zhubajie 的身高: " + zhubajie.height);
        System.out.println("zhubajie 的头:" + zhubajie.head);
        System.out.println("sunwukong 的重量:" + sunwukong.weight);
        System.out.println("sunwukong 的头:" + sunwukong.head);
        zhubajie.speak("俺老猪想娶媳妇");                     //对象调用方法
        System.out.println("zhubajie 现在的头:" + zhubajie.head);
        sunwukong.speak("老孙我重 1000 斤,我想骗八戒背我");   //对象调用方法
        System.out.println("sunwukong 现在的头:" + sunwukong.head);
    }
}
```

我们知道,类中的方法可以操作成员变量,当对象调用该方法时,方法中出现的成员变量就是指该对象的成员变量。在例 5.3 中,当对象 zhubajie 调用过方法 speak 之后,就将自己的头修改成“歪着头”;同样,对象 sunwukong 调用过方法 speak 之后,也将自己的头修改成“歪着头”。

▶ 5.3.4 对象的引用和实体

通过前面的学习我们已经知道,类是体现封装的一种数据类型,类声明的变量称作对象,对象负责存放引用,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。分配给对象的变量习惯地称作对象的实体。

1. 避免使用空对象

没有实体的对象称作空对象,空对象不能使用,即不能让一个空对象去调用方法产生行为。假如程序中使用了空对象,程序在运行时会出现 NullPointerException 异常。由于对象是动态地分配实体,所以 Java 的编译器对空对象不做检查。因此,在编写程序时要避免使用空对象。

2. 垃圾收集

一个类声明的两个对象如果具有相同的引用,那么二者就具有完全相同的实体,而且 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拥有(引用),如果发现这样的实体,就释放实体占有的内存。

以例 5.2 中的 Point 类为例,假如某个应用中使用 Point 类分别创建了两个对象 p1、p2。

```
Point p1 = new Point (5,15);
```

```
Point p2 = new Point(8,18);
```

那么内存模型如图 5.6 所示。

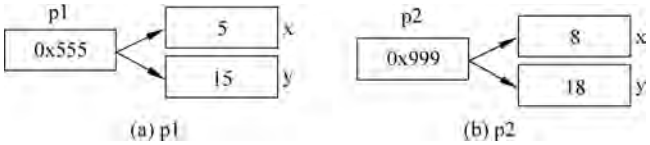

图 5.6 p1 和 p2 的引用不同

假如在程序中使用了如下的赋值语句：

```
p1 = p2
```

即把 p2 中的引用赋给了 p1，因此 p1 和 p2 本质上是一样的了。虽然在源程序中 p1 和 p2 是两个名字，但在系统看来它们的名字是一个：0x999，系统将取消原来分配给 p1 的变量（如果这些变量没有其他对象继续引用）。这时如果输出 p1.x 的结果将是 8，而不是 5。即 p1 和 p2 有相同的实体。内存模型由图 5.6 变成如图 5.7 所示。

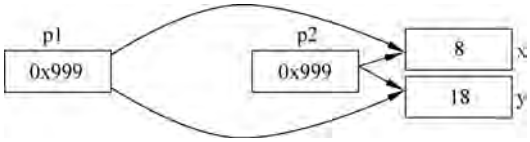

图 5.7 p1 和 p2 的引用相同

和 C++不同的是，在 Java 语言中，类有构造方法，但没有析构方法，Java 运行环境有“垃圾收集”机制，因此不必像 C++程序员那样，要时刻自己检查哪些对象应该使用析构方法释放内存。因此 Java 很少出现“内存泄露”，即由于程序忘记释放内存导致的内存溢出。

注：如果希望 Java 虚拟机立刻进行“垃圾收集”操作，可以让 System 类调用 gc()方法。


视频讲解

5.4 参数传值

方法中最重要的部分之一就是方法的参数，参数属于局部变量，当对象调用方法时，参数被分配内存空间，并要求调用者向参数传递值，即方法被调用时，参数变量必须有具体的值。

▶ 5.4.1 传值机制

在 Java 中，方法的所有参数都是“传值”的，也就是说，方法中参数变量的值是调用者指定的值的复制。例如，如果向方法的 int 型参数 x 传递一个 int 值，那么参数 x 得到的值是传递的值的复制。因此，方法如果改变参数的值，不会影响向参数“传值”的变量的值，反之亦然。参数得到的值类似生活中“原件”的“复印件”，那么改变“复印件”不影响“原件”，反之亦然。

▶ 5.4.2 基本数据类型参数的传值

对于基本数据类型的参数，向该参数传递的值的级别不可以高于该参数的级别。例如，不

可以向 int 型参数传递一个 float 值，但可以向 double 型参数传递一个 float 值。

在例 5.4 中有两个源文件：Circle.java 和 Example5_4.java，其中，Circle.java 中的 Circle 类负责创建对象，Example5_4.java 含有主类。在主类的 main 方法中使用 Circle 类来创建圆对象，该圆对象可以调用 setRadius(double r)设置自己的半径，因此，圆对象在调用 setRadius(double r)方法时，必须向方法的参数 r 传递值。程序运行效果如图 5.8 所示。

【例 5.4】

Circle.java

```
public class Circle {
    double radius,area;
    Circle(){
    }
    Circle(double r) {
       radius = r;
    }
    void setRadius(double r) {
       if(r>0){
          radius=r;
       }
    }
    double getRadius(){
       return radius;
    }
    double getArea(){
       area=3.14*radius*radius;
       return area;
    }
}
```

Example5_4.java

```
public class Example5_4 {
   public static void main(String args[]) {
       Circle circle=new Circle();
       double w=121.709;
       circle.setRadius(w);
       System.out.println("圆的半径："+circle.getRadius());
       System.out.println("圆的面积："+circle.getArea());
       System.out.println("更改向方法参数 r 传递值的 w 的值为 100");
       w=100;
       System.out.println("w="+w);
       System.out.println("圆的半径："+circle.getRadius());
    }
}
```

```
C:\ch5>java Example5_4
圆的半径：121.709
圆的面积：46513.07333834001
更改向方法参数r传递值的w的值为100
w=100.0
圆的半径：121.709
```

图 5.8　基本数据类型参数的传值

▶ 5.4.3　引用类型参数的传值

Java 的引用型数据包括前面学习的数组、刚刚学习的对象以及后面将要学习的接口。当参数是引用类型时，“传值”传递的是变量中存放的“引用”，而不是变量引用的实体。

需要注意的是，对于两个同类型的引用型变量，如果具有同样的引用，就会用同样的实体。因此，如果改变参数变量引用的实体，就会导致原变量的实体发生同样的变化；但是，改变参数中存放的“引用”不会影响向其传值的变量中存放的“引用”，反之亦然，如图 5.9 所示。

例 5.5 中涉及引用类型参数，请注意程序的运行效果。例 5.5 中除了使用例 5.4 中的 Circle 类外，还需要一个 Circular 类（刻画圆锥）和一个主类。程序在主类的 main 方法中首先使用 Circle 类创建一个“圆”对象 circle，然后使用 Circular 类创建一个圆锥对象 circular，在创建圆锥对象 circular 时，需要将先前 Circle 类的实例 circle，即“圆”对象的引用，传递给圆锥对象的成员变量 bottom。程序运行效果如图 5.10 所示。

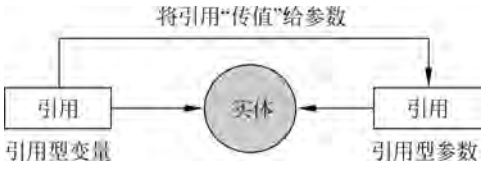

图 5.9 引用类型参数的传值

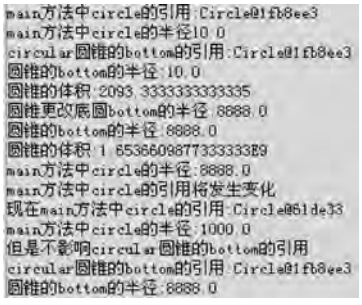

图 5.10 向参数传递对象的引用

【例 5.5】

Circular.java

```
public class Circular {
    Circle bottom;
    double height;
    Circular(Circle c,double h) {      //构造方法,将 Circle 类的实例的引用传递给 bottom
       bottom = c;
       height = h;
    }
    double getVolume() {
       return bottom.getArea() * height/3.0;
    }
    double getBottomRadius() {
       return bottom.getRadius();
    }
    public void setBottomRadius(double r){
       bottom.setRadius(r);
    }
}
```

Example5_5.java

```
public class Example5_5 {
   public static void main(String args[]) {
       Circle circle = new Circle(10);                          //【代码 1】
       System.out.println("main 方法中 circle 的引用:" + circle);
       System.out.println("main 方法中 circle 的半径" + circle.getRadius());
       Circular circular = new Circular(circle,20);             //【代码 2】
```

```
        System.out.println("circular 圆锥的 bottom 的引用:" + circular.bottom);
        System.out.println("圆锥的 bottom 的半径:" + circular.getBottomRadius());
        System.out.println("圆锥的体积:" + circular.getVolume());
        double r = 8888;
        System.out.println("圆锥更改底圆 bottom 的半径:" + r);
        circular.setBottomRadius(r);                          //【代码 3】
        System.out.println("圆锥的 bottom 的半径:" + circular.getBottomRadius());
        System.out.println("圆锥的体积:" + circular.getVolume());
        System.out.println("main 方法中 circle 的半径:" + circle.getRadius());
        System.out.println("main 方法中 circle 的引用将发生变化");
        circle = new Circle(1000);                            //重新创建 circle【代码 4】
        System.out.println("现在 main 方法中 circle 的引用:" + circle);
        System.out.println("main 方法中 circle 的半径:" + circle.getRadius());
        System.out.println("但是不影响 circular 圆锥的 bottom 的引用");
        System.out.println("circular 圆锥的 bottom 的引用:" + circular.bottom);
        System.out.println("圆锥的 bottom 的半径:" + circular.getBottomRadius());
    }
}
```

我们对例 5.5 中的 Example5_5.java 中的重要的、需要理解的代码给出了代码 1～代码 4 注释，以下结合对象的内存模型，对这些重要的代码给予讲解。

1）执行代码 1 后内存中的对象模型

执行代码 1：

```
Circle circle = new Circle(10);
```

后，内存中诞生了一个 circle 对象，内存中对象的模型如图 5.11 所示。

2）执行代码 2 后内存中的对象模型

执行代码 2：

```
Circular circular = new Circular(circle,20);
```

后，内存中又诞生了一个 circular 对象。执行代码 2 将 circle 对象的引用以“传值”方式传递给 circular 对象的 bottom，因此，circular 对象的 bottom 和 circle 对象就有同样的实体(radius)。内存中对象的模型如图 5.12 所示。

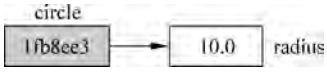

图 5.11　执行代码 1 后内存中的对象模型

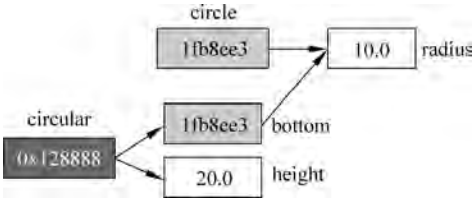

图 5.12　执行代码 2 后内存中的对象模型

3）执行代码 3 后内存中的对象模型

对于两个同类型的引用型变量，如果具有同样的引用，就会用同样的实体。因此，如果改变参数变量所引用的实体，就会导致原变量的实体发生同样的变化。

执行代码 3：

```
circular.setBottomRadius(r);
```

就使得 circular 的 bottom 和 circle 的实体(radius)发生了同样的变化,如图 5.13 所示。

4) 执行代码 4 后内存中的对象模型

执行代码 4:

```
circle = new Circle(1000);
```

使得 circle 的引用发生变化,重新创建了 circle 对象,即 circle 对象将获得新的实体(circle 对象的 radius 的值是 1000),但 circle 先前的实体不被释放,因为这些实体还是 circular 的 bottom 的实体。最初 circle 对象的引用是以传值方式传递给 circular 对象的 bottom 的,所以,circle 的引用发生变化并不影响 circular 的 bottom 的引用(bottom 对象的 radius 的值仍然是 8888)。对象的模型如图 5.14 所示。

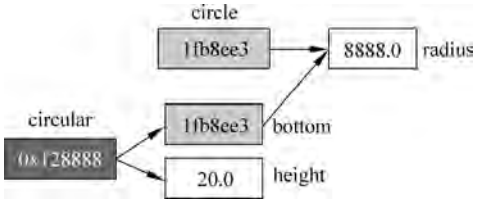

图 5.13 执行代码 3 后内存中的对象模型

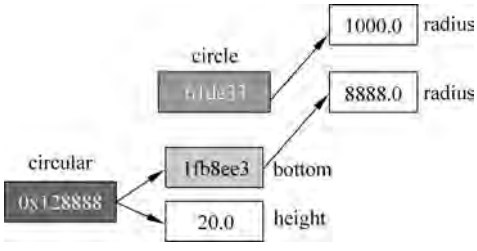

图 5.14 执行代码 4 后内存中的对象模型

5.5 对象的组合

视频讲解

我们已经知道,一个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。例如,前面的例 5.5 中的圆锥对象就将一个圆对象作为自己的成员,即圆锥有一个圆底。

▶ 5.5.1 由矩形和圆组合而成的图形

一个矩形和一个圆可以组合成各种形状的几何图形,如图 5.15 所示。

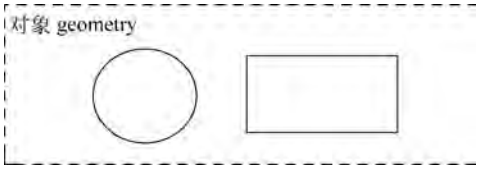

图 5.15 对象 geometry 是圆和矩形的组合

在例 5.6 中,一共编写了 4 个类,分成 4 个源文件:Rectangle.java、Circle.java、Geometry.java 和 Example5_6.java,需要将这 4 个源文件分别编辑,并保存在相同的目录(例如 C:\ch5)中。

- Rectangle.java 中的 Rectangle 类有 double 型的成员变量 x、y、width、height,分别用来表示矩形左上角的位置坐标以及矩形的宽和高。该类提供了修改 x、y、width、height 以及返回 x、y、width、height 的方法。

- Circle.java 中的 Circle 类有 double 型的成员变量 x、y、radius，分别用来表示对象的圆心坐标和圆的半径。该类提供了修改 x、y、radius 以及返回 x、y、radius 的方法。
- Geometry.java 中的 Geometry 类有 Rectangle 类型和 Circle 类型的成员变量，名字分别为 rect 和 circle。也就是说，Geometry 类创建的对象（几何图形）是由一个 Rectangle 对象和一个 Circle 对象组合而成。该类提供了修改 rect、circle 位置和大小的方法，提供了显示 rect 和 circle 位置关系的方法。
- Example5_6.java 含有主类，主类在 main 方法中用 Geometry 类创建对象 geometry，该对象调用相应的方法设置其中的圆的位置和半径、矩形的位置以及宽和高。

例 5.6 的运行效果如图 5.16 所示。

【例 5.6】

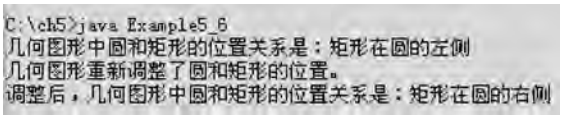

图 5.16　圆和矩形组合的对象

Rectangle.java

```
public class Rectangle {
    double x,y,width,height;
    public void setX(double a) {
       x = a;
    }
    public double getX() {
      return x;
    }
    public void setY(double b) {
       y = b;
    }
    public double getY(){
      return y;
    }
    public void setWidth(double w) {
       if(w > 0)
          width = w;
    }
    public double getWidth(){
       return width;
    }
    public void setHeight(double h) {
       if(h > 0)
          height = h;
    }
    public double getHeight() {
       return height;
    }
}
```

Circle.java

```
public class Circle {
    double x,y,radius;
    public void setX(double a) {
```

```
        x = a;
    }
    public double getX() {
        return x;
    }
    public void setY(double b){
        y = b;
    }
    public double getY() {
        return y;
    }
    public void setRadius(double r){
        if(r > 0 )
          radius = r;
    }
    public double getRadius(){
        return radius;
    }
}
```

Geometry. java

```
public class Geometry {
    Rectangle rect;
    Circle circle;
    Geometry(Rectangle rect,Circle circle){
        this.rect = rect;
        this.circle = circle;
    }
    public void setCirclePosition(double x,double y){
        circle.setX(x);
        circle.setY(y);
    }
    public void setCircleRadius(double radius){
        circle.setRadius(radius);
    }
    public void setRectanglePosition(double x,double y){
        rect.setX(x);
        rect.setY(y);
    }
    public void setRectangleWidthAndHeight(double w,double h){
        rect.setWidth(w);
        rect.setHeight(h);
    }
    public void showState(){
        double circleX = circle.getX();
        double rectX = rect.getX();
        if(rectX - rect.getWidth()>= circleX + circle.getRadius())
           System.out.println("矩形在圆的右侧");
        if(rectX + rect.getWidth()<= circleX - circle.getRadius())
```

```
            System.out.println("矩形在圆的左侧");
        }
    }
```

Example5_6.java

```
public class Example5_6 {
    public static void main(String args[]){
      Rectangle rect = new Rectangle();
      Circle circle = new Circle();
      Geometry geometry;
      geometry = new Geometry(rect,circle);
      geometry.setRectanglePosition(30,40);
      geometry.setRectangleWidthAndHeight(120,80);
      geometry.setCirclePosition(260,30);
      geometry.setCircleRadius(60);
      System.out.print("几何图形中圆和矩形的位置关系是: ");
      geometry.showState();                                    //显示圆和矩形的位置关系
      System.out.println("几何图形重新调整了圆和矩形的位置。");
      geometry.setRectanglePosition(220,160);
      geometry.setCirclePosition(40,30);
      System.out.print("调整后,几何图形中圆和矩形的位置关系是: ");
      geometry.showState();                                    //显示圆和矩形的位置关系
    }
}
```

▶ 5.5.2 关联关系和依赖关系的 UML 图

1. 关联关系

如果 A 类中成员变量是用 B 类声明的对象,那么 A 和 B 的关系是关联关系,称 A 关联于 B 或 A 组合了 B。如果 A 关联于 B,那么通过使用一条实线连 A 和 B 的 UML 图,实线的起始端是 A 的 UML 图,终点端是 B 的 UML 图,但终点端使用一条指向 B 的 UML 图的方向箭头表示实线的结束。

图 5.17 是例 5.5 中 Circular 类关联于 Circle 类的 UML 图。

2. 依赖关系

如果 A 类中某个方法的参数是用 B 类声明的对象,或者某个方法返回的数据类型是 B 类对象,那么 A 和 B 的关系是依赖关系,称 A 依赖于 B。如果 A 依赖于 B,那么通过使用一条虚线连 A 和 B 的 UML 图,虚线的起始端是 A 的 UML 图,终点端是 B 的 UML 图,但终点端使用一条指向 B 的 UML 图的方向箭头表示虚线的结束。

图 5.18 是 ClassRoom 依赖于 Student 的 UML 图。

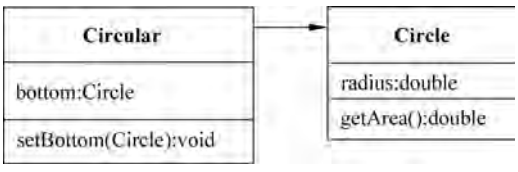

图 5.17 关联关系的 UML 图

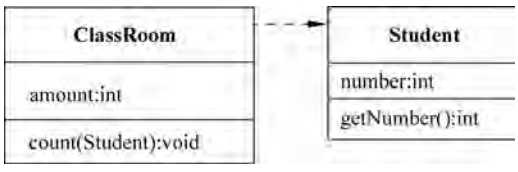

图 5.18 依赖关系的 UML 图

注：在 Java 中，习惯上将 A 关联于 B 也称作 A 依赖于 B，当需要强调 A 是通过方法参数依赖于 B 时，就在 UML 图中使用虚线连接 A 和 B 的 UML 图。


视频讲解

5.6 实例成员与类成员

▶ 5.6.1 实例变量和类变量的声明

在讲述类的时候我们讲过，类体中包括成员变量的声明和方法的定义，而成员变量又可细分为实例变量和类变量。在声明成员变量时，用关键字 static 给予修饰的称作类变量（类变量也称为 static 变量、静态变量）；否则称作实例变量。例如：

```
class Dog {
   float x;                                    //实例变量
   static int y;                               //类变量
}
```

Dog 类中，x 是实例变量，而 y 是类变量。需要注意的是，static 需放在变量的类型的前面。

▶ 5.6.2 实例变量和类变量的区别

1. 不同对象的实例变量互不相同

我们已经知道，一个类通过使用 new 运算符可以创建多个不同的对象，这些对象将被分配不同的（成员）变量，说得准确些就是：分配给不同的对象的实例变量占有不同的内存空间，改变其中一个对象的实例变量不会影响其他对象的实例变量。

2. 所有对象共享类变量

如果类中有类变量，当使用 new 运算符创建多个不同的对象时，分配给这些对象的这个类变量占有相同的一处内存，改变其中一个对象的这个类变量会影响其他对象的这个类变量。也就是说对象共享类变量。

3. 通过类名直接访问类变量

我们知道，当 Java 程序执行时，类的字节码文件被加载到内存，如果该类没有创建对象，类中的实例变量不会被分配内存。但是，类中的类变量在该类被加载到内存时，就被分配了相应的内存空间。如果该类创建对象，那么不同对象的实例变量互不相同，即分配不同的内存空间；而类变量不再重新分配内存，所有的对象共享类变量，即所有的对象的类变量是相同的一处内存空间，类变量的内存空间直到程序退出运行，才释放占有的内存。

类变量是与类相关联的数据变量，也就是说，类变量是和该类创建的所有对象相关联的变量，改变其中一个对象的类变量就同时改变了其他对象的这个类变量。因此，类变量不仅可以通过某个对象访问，也可以直接通过类名访问。实例变量仅仅是和相应的对象关联的变量，也就是说，不同对象的实例变量互不相同，即分配不同的内存空间，改变其中一个对象的实例变量不会影响其他对象的实例变量。实例变量可以通过对象访问，不可以使用类名访问。

在例 5.7 中，Ladder.java 中的 Ladder 类创建的梯形对象共享一个下底。程序运行效果如图 5.19 所示。

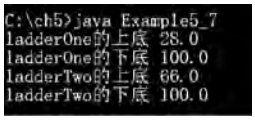

图 5.19 梯形共享一个下底

【例 5.7】

Ladder.java

```
public class Ladder {
    double 上底,高;                                    //实例变量
    static double 下底;                                 //类变量
    void 设置上底(double a) {
        上底 = a;
    }
    void 设置下底(double b) {
        下底 = b;
    }
    double 获取上底() {
       return 上底;
    }
    double 获取下底() {
       return 下底;
    }
}
```

Example5_7.java

```
public class Example5_7 {
   public static void main(String args[]) {
       Ladder.下底 = 100;                    //Ladder 的字节码文件被加载到内存,通过类名操作类变量
       Ladder ladderOne = new Ladder();
       Ladder ladderTwo = new Ladder();
       ladderOne.设置上底(28);
       ladderTwo.设置上底(66);
       System.out.println("ladderOne 的上底:" + ladderOne.获取上底());
       System.out.println("ladderOne 的下底:" + ladderOne.获取下底());
       System.out.println("ladderTwo 的上底:" + ladderTwo.获取上底());
       System.out.println("ladderTwo 的下底:" + ladderTwo.获取下底());
    }
}
```

例 5.7 从 Example5_7.java 中的主类的 main 方法开始运行,当执行

```
Ladder 下底 = 100;
```

时,Java 虚拟机首先将 Ladder 的字节码加载到内存,同时为类变量“下底”分配了内存空间,并赋值 100,如图 5.20 所示。

当执行

```
Ladder ladderOne = new Ladder();
Ladder ladderTwo = new Ladder();
```

图 5.20 为下底分配内存

时,实例变量“上底”和“高”都两次被分配内存空间,分别被对象 ladderOne 和 ladderTwo 引用,而类变量“下底”不再分配内存,直接被对象 ladderOne 和 ladderTwo 引用、共享,如图 5.21 所示。

注:类变量似乎破坏了封装性,其实不然,当对象调用实例方法时,该方法中出现的类变量也是该对象的变量,只不过这个变量和所有的其他对象共享而已。

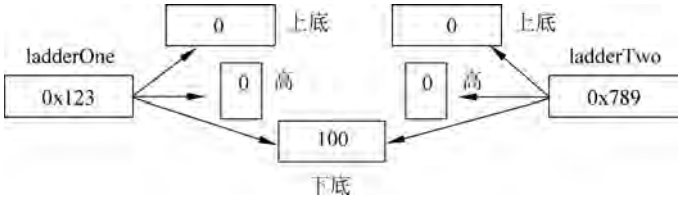

图 5.21 对象共享类变量

▶ 5.6.3 实例方法和类方法的定义

类中的方法也可分为实例方法和类方法。方法声明时，方法类型前面不加关键字 static 修饰的是实例方法，加 static 关键字修饰的是类方法(静态方法)。例如：

```
class A {
    int a;
    float max(float x,float y) {                    //实例方法
      …
    }
    static float jerry() {                          //类方法
      …
    }
    static void speak(String s) {                   //类方法
      …
    }
}
```

A 类中的 jerry 方法和 speak 方法是类方法，max 方法是实例方法。需要注意的是 static 需放在方法的类型的前面。

▶ 5.6.4 实例方法和类方法的区别

1. 对象调用实例方法

当类的字节码文件被加载到内存时，类的实例方法不会被分配入口地址，只有该类创建对象后，类中的实例方法才分配入口地址，从而实例方法可以被类创建的任何对象调用执行。需要注意的是，当创建第一个对象时，类中的实例方法就分配了入口地址，当再创建对象时，不再分配入口地址。也就是说，方法的入口地址被所有的对象共享，当所有的对象都不存在时，方法的入口地址才被取消。

实例方法中不仅可以操作实例变量，也可以操作类变量。当对象调用实例方法时，该方法中出现的实例变量就是分配给该对象的实例变量，该方法中出现的类变量也是分配给该对象的变量，只不过这个变量和所有的其他对象共享而已。

2. 类名调用类方法

对于类中的类方法，在该类被加载到内存时，就分配了相应的入口地址。从而类方法不仅可以被类创建的任何对象调用执行，也可以直接通过类名调用。类方法的入口地址直到程序退出才被取消。需要注意的是，实例方法不能通过类名调用，只能由对象来调用。

和实例方法不同的是，类方法不可以操作实例变量，这是因为在类创建对象之前，实例成

员变量还没有分配内存。

如果一个方法不需要操作实例成员变量就可以实现某种功能，可以考虑将这样的方法声明为类方法。这样做的好处是避免创建对象浪费内存。

在例 5.8 中，Sum 类中的 getContinueSum 方法是类方法。

【例 5.8】

Example5_8.java

```
class Sum {
    int x,y,z;
    static int getContinueSum(int start,int end) {
        int sum = 0;
        for(int i = start;i <= end;i++) {
          sum = sum + i;
        }
        return sum;
    }
}
public class Example5_8 {
    public static void main(String args[]) {
        int result = Sum.getContinueSum(0,100);
        System.out.println(result);
    }
}
```

视频讲解

5.7 方法重载与多态

Java 中存在两种多态：重载(Overload)和重写(Override)。重写是与继承有关的多态，将在第 6 章讨论。

方法重载是两种多态的一种。例如，让一个人执行“求面积”操作时，他可能会问你求什么面积。所谓功能多态性，是指可以向功能传递不同的消息，以便让对象根据相应的消息来产生相应的行为。对象的功能通过类中的方法来体现，那么功能的多态性就是方法的重载。方法重载的意思是一个类中可以有多个方法具有相同的名字，但这些方法的参数必须不同，即或者是参数的个数不同，或者是参数的类型不同。下面的 A 类中的 add 方法是重载方法。

```
class A {
    float add(int a,int b) {
        return a + b;
    }
    float add(long a,int b) {
        return a + b;
    }
    double add(double a,int b) {
        return a + b;
    }
}
```

注：方法的返回类型和参数的名字不参与比较，也就是说如果两个方法的名字相同，即使类型不同，也必须保证参数不同。

在例 5.9 中，People 类中的 computerArea 方法是重载方法。另外，例 5.9 除了 People、Tixing 和主类外，还用到了例 5.4 中的 Circle 类。程序运行效果如图 5.22 所示。

【例 5.9】

Tixing.java

```
public class Tixing {
    double above,bottom,height;
    Tixing(double a,double b,double h) {
        above = a;
        bottom = b;
        height = h;
    }
    double getArea() {
        return (above + bottom) * height/2;
    }
}
```

People.java

```
public class People {
    double computerArea(Circle c) {
        double area = c.getArea();
        return area;
    }
    double computerArea(Tixing t) {
        double area = t.getArea();
        return area;
    }
}
```

Example5_9.java

```
public class Example5_9 {
  public static void main(String args[]) {
        Circle circle = new Circle();
        circle.setRadius(196.87);
        Tixing ladder = new Tixing(3,21,9);
        People zhang = new People();
        System.out.println("zhang 计算圆的面积：");
        double result = zhang.computerArea(circle);
        System.out.println(result);
        System.out.println("zhang 计算梯形的面积：");
        result = zhang.computerArea(ladder);
        System.out.println(result);
    }
}
```

```
C:\ch5>java Example5_9
zhang计算圆的面积：
121699.48228600002
zhang计算梯形的面积：
108.0
```

图 5.22 方法重载

5.8 this 关键字

视频讲解

this 是 Java 的一个关键字，表示某个对象。this 可以出现在实例方法和构造方法中，但不可以出现在类方法中。

5.8.1 在构造方法中使用 this

this 关键字出现在类的构造方法中时，代表使用该构造方法创建的对象。

在例 5.10 中，People 类的构造方法中使用了 this。

【例 5.10】

People.java

```
public class People{
    int leg,hand;
    String name;
    People(String s){
        name = s;
        this.init();                          //可以省略 this,即将“this.init();”写成“init();”
    }
    void init(){
       leg = 2;
       hand = 2;
       System.out.println(name + "有" + hand + "只手" + leg + "条腿");
    }
    public static void main(String args[]){
      People bushi = new People("布什");      //创建 bushi 时,构造方法中的 this 就是对象 bushi
    }
}
```

5.8.2 在实例方法中使用 this

实例方法只能通过对象来调用，不能通过类名来调用。当 this 关键字出现在实例方法中时，代表正在调用该方法的当前对象。

实例方法可以操作类的成员变量，当实例成员变量在实例方法中出现时，默认的格式是：

```
this.成员变量;
```

当 static 成员变量在实例方法中出现时，默认的格式是：

```
类名.成员变量;
```

例如：

```
class A {
    int x;
    static int y;
    void f() {
       this.x = 100;
       A.y = 200;
    }
}
```

上述 A 类中的实例方法 f 中出现了 this，this 就代表使用 f 的当前对象。所以，“this.x”就表示当前对象的变量 x，当对象调用方法 f 时，将 100 赋给该对象的变量 x。因此，当一个对象调用方法时，方法中的实例成员变量就是指分配给该对象的实例成员变量，而 static 变量和其他对象共享。因此，通常情况下，可以省略实例成员变量名字前面的“this.”以及 static 变量前面

的“类名.”。

例如：

```
class A {
    int x;
    static int y;
    void f() {
        x = 100;
        y = 200;
    }
}
```

但是，当实例成员变量的名字和局部变量的名字相同时，成员变量前面的“this.”或“类名.”就不可以省略。

我们知道类的实例方法可以调用类的其他方法，对于实例方法调用的默认格式是：

```
this.方法;
```

对于类方法调用的默认格式是：

```
类名.方法;
```

例如：

```
class B {
    void f() {
       this.g();
        B.h();
    }
    void g() {
        System.out.println("ok");
    }
    static void h() {
        System.out.println("hello");
    }
}
```

在上述B类中的方法f中出现了this，this代表调用方法f的当前对象，所以，方法f的方法体中的this.g()就是当前对象调用方法g，也就是说，某个对象调用方法f的过程中，又调用了方法g。由于这种逻辑关系非常明确，一个实例方法调用另一个方法时可以省略方法名字前面的“this.”或“类名.”。

例如：

```
class B {
    void f() {
        .g();                              //省略 this
         h();                              //省略类名
    }
    void g() {
        System.out.println("ok");
    }
    static void h() {
```

```
            System.out.println("hello");
        }
    }
```

注：this 不能出现在类方法中，这是因为类方法可以通过类名直接调用，这时，可能还没有任何对象诞生。

视频讲解

5.9 包

包是 Java 语言中有效地管理类的一个机制。不同的 Java 源文件中可能出现名字相同的类，如果想区分这些类，就需要使用包名。

▶ 5.9.1 包语句

通过关键字 package 声明包语句。package 语句作为 Java 源文件的第一条语句，指明该源文件定义的类所在的包，即为该源文件中声明的类指定包名。package 语句的一般格式为：

```
package 包名;
```

如果源程序中省略了 package 语句，源文件中所定义命名的类被隐含地认为是无名包的一部分，只要这些类的字节码被存放在相同的目录中，那么它们就属于同一个包，但没有包名。

包名可以是一个合法的标识符，也可以是若干个标识符加“.”分隔而成，例如：

```
package sunrise;
package sun.com.cn;
```

▶ 5.9.2 有包名的类的存储目录

如果一个类有包名，那么就不能在任意位置存放它，否则虚拟机将无法加载这样的类。

程序如果使用了包语句，例如：

```
package tom.jiafei;
```

那么存储文件的目录结构中必须包含如下的结构：

```
…\tom\jiafei
```

例如：

```
C:\1000\tom\jiafei
```

并且要将源文件编译得到的类的字节码文件保存在目录 C:\1000\tom\jiafei 中（源文件可以任意存放）。

当然，可以将源文件保存在 C:\1000\tom\jiafei 中，然后进入 tom\jiafei 的上一层目录 1000 中编译源文件：

```
C:\1000 > javac tom\jiafei\源文件
```

那么得到的字节码文件默认地保存在当前目录 C:\1000\tom\jiafei 中。

▶ 5.9.3 运行有包名的主类

如果主类的包名是 tom.jiafei，那么主类的字节码一定存放在…\tom\jiafei 目录中，必须到 tom\jiafei 的上一层目录（即 tom 的父目录）中去运行主类。假设 tom\jiafei 的上一层目录是 1000，那么，必须用如下格式来运行：

```
C:\1000 > java tom.jiafei.主类名
```

即运行时，必须写主类的全名。因为使用了包名，主类全名是“包名.主类名”（就好比大连的全名是“中国.辽宁.大连”）。

例 5.11 中的 Student.java 和 Example5_11.java 使用包语句。

【例 5.11】

Student.java

```
package tom.jiafei;
public class Student{
    int number;
    Student(int n){
        number = n;
    }
    void speak(){
        System.out.println("Student 类的包名是 tom.jiafei,我的学号: " + number);
    }
}
```

Example5_11.java

```
package tom.jiafei;
public class Example5_11 {
    public static void main(String args[]){
        Student stu = new Student(10201);
        stu.speak();
        System.out.println("主类的包名也是 tom.jiafei");
    }
}
```

由于 Example5_11.java 用到了同一包中的 Student 类，所以在编译 Example5_11.java 时，需在包的上一层目录使用 javac 来编译。以下说明怎样编译和运行例 5.11。

1. 编译

将上述两个源文件保存到 C:\1000\tom\jiafei 中，然后进入 tom\jiafei 的上一层目录 1000 中编译两个源文件：

```
C:\1000 > javac tom\jiafei\Student.java
C:\1000 > javac tom\jiafei\Example5_11.java
```

编译通过后，C:\1000\tom\jiafei 目录中就会有相应的字节码文件 Student.class 和 Example5_11.class。

也可以进入 C:\1000\tom\jiafei 目录中，使用通配符 * 编译全部的源文件：

```
C:\1000\tom\jiafei > javac *.java
```

2. 运行

运行程序时必须到 tom\jiafei 的上一层目录 1000 中运行，例如：

```
C:\1000\java tom.jiafei.Example5_11
```

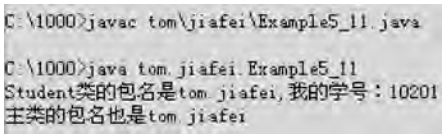

图 5.23　运行有包名的主类

例 5.11 的编译、运行效果如图 5.23 所示。

注意：Java 语言不允许用户程序使用 java 作为包名的第一部分，例如 java.bird 是非法的包名（发生运行异常）。

视频讲解

5.10　import 语句

一个类可能需要另一个类声明的对象作为自己的成员或方法中的局部变量，如果这两个类在同一个包中，当然没有问题，例如，前面的许多例子中涉及的类都是无名包，只要存放在相同的目录中，它们就是在同一个包中；对于包名相同的类，如例 5.11，它们必须按照包名的结构存放在相应的目录中。但是，如果一个类想要使用的那个类和它不在一个包中，它怎样才能使用这样的类呢？这正是 import 语句要帮助完成的使命。以下详细讲解 import 语句。

▶ 5.10.1　引入类库中的类

用户编写的类和类库中的类不在一个包中。如果用户需要类库中的类，就必须使用 import 语句。

在编写源文件时，除了自己编写类外，经常需要使用 Java 提供的许多类，这些类可能在不同的包中。在学习 Java 语言时，使用已经存在的类，避免一切从头做起，这是面向对象编程的一个重要方面。

为了能使用 Java 提供的类，可以使用 import 语句引入包中的类。在一个 Java 源程序中可以有多个 import 语句，它们必须写在 package 语句（假如有 package 语句的话）和源文件中类的定义之间。Java 为我们提供了大约 130 个包（在后续的章节将需要一些重要包中的类），例如：

java.lang：包含所有的基本语言类（见第 9、12 章）

javax.swing：包含抽象窗口工具集中的图形、文本、窗口 GUI 类（见第 11 章）

java.io：包含所有的输入输出类（见第 10 章）

java.util：包含实用类（见第 9 章）

java.sql：包含操作数据库的类（见第 14 章）

java.net：包含所有实现网络功能的类（见第 13 章）

如果要引入一个包中的全部类，可以用通配符号星号（*）来代替，例如：

```
import java.util.*;
```

表示引入 java.util 包中所有的类，而

```
import java.util.Date;
```

只是引入 java.util 包中的 Date 类。

例如，如果用户编写一个程序，并想使用 java.util 中的 Date 类创建对象来显示本地的时间，那么就可以使用 import 语句引入 java.util 中的 Date 类。在例 5.12 中，Example5_12.java 使用了 import 语句，运行效果如图 5.24 所示。

【例 5.12】

Example5_12.java

```
import java.util.Date;
public class Example5_12 {
    public static void main(String args[]) {
        Date date = new Date();
        System.out.println("本地机器的时间:");
        System.out.println(date);
    }
}
```

```
C:\ch5>java Example5_12
本地机器的时间
Fri Jan 01 21:47:04 CST 2010
```

图 5.24 引入类库中的类

注：① java.lang 包是 Java 语言的核心类库，它包含了运行 Java 程序必不可少的系统类，系统自动为程序引入 java.lang 包中的类（例如 System 类、Math 类等），因此不需要再使用 import 语句引入该包中的类。

② 如果使用 import 语句引入了整个包中的类，那么可能会增加编译时间，但绝对不会影响程序运行的性能，因为当程序执行时，只是将你真正使用的类的字节码文件加载到内存。

▶ 5.10.2 引入自定义包中的类

1. 有包名的源文件

包名路径左对齐。所谓包名路径左对齐，就是让源文件中的包名所对应的路径和它要用 import 语句引入的非类库中的类的包名所对应的路径的父目录相同。假如用户的源文件的包名是 hello.nihao，该源文件想引入的非类库中的包名是 sohu.com 的类，那么只需将两个包名所对应的路径左对齐，即让两个包名所对应的路径的父目录相同。例如，将用户的源文件和它准备用 import 语句引入的包名是 sohu.com 的类分别保存在：

```
C:\ch5\hello\nihao
```

和

```
C:\ch5\sohu\com
```

中，即 hello\nihao 和 sohu\com 的父目录相同，都是 C:\ch5。

2. 无包名的源文件

包名路径和源文件左对齐。假如用户的源文件没有包名，该源文件想引入的非类库中的包名是 sohu.com 的类，那么只需让源文件中 import 语句要引入的非类库中的类的包名路径的父目录和用户的源文件所在的目录相同，即包名路径和源文件左对齐。例如，将用户的源文件和它准备用 import 语句引入的包名是 sohu.com 的类分别保存在：

```
C:\ch5\
```

和

```
C:\ch5\sohu\com
```

中，即 sohu\com 的父目录和用户的源文件所在目录都是 C:\ch5。

编写一个有价值的类是令人高兴的事情，可以将这样的类打包（自定义包），形成有价值的“软件产品”，供其他软件开发者使用。

在例 5.13 中，Triangle.java 含有一个 Triangle 类，该类可以创建“三角形”对象，一个需要三角形的用户可以使用 import 语句引入 Triangle 类。将例 5.13 中的 Triangle.java 源文件保存到 C:\ch5\tom\jiafei 中（将 tom\jiafei 目录放在文件夹 ch5 中）。

如下编译 Triangle 类：

```
C:\ch5> javac tom\jiafei\Triangle.java
```

【例 5.13】

Triangle.java

```
package tom.jiafei;
public class Triangle {
    double sideA,sideB,sideC;
    boolean isTriangle;
    public Triangle(double a,double b,double c) {
       sideA = a;
       sideB = b;
       sideC = c;
       if(a + b > c&&a + c > b&&c + b > a) {
          isTriangle = true;
       }
       else {
          isTriangle = false;
       }
    }
    public void 计算面积() {
       if(isTriangle) {
           double p = (sideA + sideB + sideC)/2.0;
           double area = Math.sqrt(p * (p - sideA) * (p - sideB) * (p - sideC)) ;
           System.out.println("是一个三角形,面积是:" + area);
       }
       else {
           System.out.println("不是一个三角形,不能计算面积");
       }
    }
    public void 修改三边(double a,double b,double c) {
       sideA = a;
       sideB = b;
       sideC = c;
       if(a + b > c&&a + c > b&&c + b > a) {
          isTriangle = true;
       }
       else {
          isTriangle = false;
       }
    }
}
```

在例 5.14 中，Example5_14.java 中的主类（无包名）使用 import 语句引入 tom.jiafei 包中的 Triangle 类，以便创建三角形，并计算三角形的面积。将 Example5_14.java 保存在 C:\ch5 目录中（因为 ch5 下有 tom\jiafei 子目录）。程序运行效果如图 5.25 所示。

【例 5.14】

Example5_14.java

```
import tom.jiafei.Triangle;
public class Example5_14 {
    public static void main(String args[]) {
        Triangle tri = new Triangle(6,7,10);
        tri.计算面积();
        tri.修改三边(3,4,5);
        tri.计算面积();
    }
}
```

```
C:\ch5>java Example5_14
是一个三角形，面积是：20.862465970833866
是一个三角形，面积是：6.0
```

图 5.25 引入自定义包中的类

视频讲解

5.11 访问权限

我们已经知道，当用一个类创建了一个对象之后，该对象可以通过“.”运算符操作自己的变量、使用类中的方法，但对象操作自己的变量和使用类中的方法是有一定限制的。

▶ 5.11.1 何谓访问权限

所谓访问权限，是指对象是否可以通过“.”运算符操作自己的变量或通过“.”运算符使用类中的方法。访问限制修饰符有 private、protected 和 public，都是 Java 的关键字，用来修饰成员变量或方法。以下来说明这些修饰符的具体作用。

需要特别注意的是，在编写类的时候，类中的实例方法总是可以操作该类中的实例变量和类变量；类方法总是可以操作该类中的类变量，与访问限制符没有关系。

▶ 5.11.2 私有变量和私有方法

用关键字 private 修饰的成员变量和方法称为私有变量和私有方法。例如：

```
class Tom {
    private float weight;                    //weight 是 private 的 float 型变量
    private float f(float a,float b) {       //方法 f 是 private 方法
       return a + b;
    }
}
```

当在另外一个类中用类 Tom 创建了一个对象后，该对象不能访问自己的私有变量和私有方法。例如：

```
class Jerry {
    void g() {
       Tom cat = new Tom();
       cat.weight = 23f;                     //非法
```

```
        float sum = cat.f(3,4);                    //非法
    }
}
```

如果 Tom 类中的某个成员是私有类变量(静态成员变量),那么在另外一个类中也不能通过类名 Tom 来操作这个私有类变量。如果 Tom 类中的某个方法是私有的类方法,那么在另外一个类中也不能通过类名 Tom 来调用这个私有的类方法。

当用某个类在另外一个类中创建对象后,如果不希望该对象直接访问自己的变量,即通过“.”运算符来操作自己的成员变量,就应当将该成员变量访问权限设置为 private。面向对象编程提倡对象应当调用方法来改变自己的属性,类应当提供操作数据的方法,这些方法可以经过精心的设计,使得对数据的操作更加合理,如例 5.15 所示。

【例 5.15】

Student.java

```
public class Student {
   private int age;
   public void setAge(int age) {
      if(age >= 7&&age <= 28) {
          this.age = age;
      }
   }
   public int getAge() {
      return age;
   }
}
```

Example5_15.java

```
public class Example5_15 {
   public static void main(String args[]) {
      Student zhang = new Student();
      Student geng = new Student();
      zhang.setAge(23);
      System.out.println("zhang 的年龄: " + zhang.getAge());
      geng.setAge(25);
      //zhang.age = 23;或 geng.age = 25;都是非法的,因为 zhang 和 geng 已经不在 Student 类中
      System.out.println("geng 的年龄: " + geng.getAge());
    }
}
```

▶ 5.11.3 共有变量和共有方法

用 public 修饰的成员变量和方法被称为共有变量和共有方法。例如:

```
class Tom {
    public float weight;                    //weight 是 public 的 float 型变量
    public float f(float a,float b) {       //方法 f 是 public 方法
        return a + b;
    }
}
```

当在任何一个类中用类 Tom 创建了一个对象后，该对象能访问自己的 public 变量和类中的 public 方法。例如：

```
class Jerry {
    void g() {
        Tom cat = new Tom();
         cat.weight = 23f;                    //合法
         float sum = cat.f(3,4);              //合法
    }
}
```

如果 Tom 类中的某个成员是 public 类变量，那么在另外一个类中也可以通过类名 Tom 来操作 Tom 类中的这个成员变量。如果 Tom 类中的某个方法是 public 类方法，那么在另外一个类中也可以通过类名 Tom 来调用 Tom 类中的这个 public 类方法。

▶ 5.11.4 友好变量和友好方法

不用 private、public、protected 修饰符的成员变量和方法被称为友好变量和友好方法。例如：

```
class Tom {
     float weight;                            //weight 是友好的 float 型变量
     float f(float a,float b) {               //方法 f 是友好方法
        return a + b;
     }
}
```

当在另外一个类中用类 Tom 创建了一个对象后，如果这个类与 Tom 类在同一个包中，那么该对象能访问自己的友好变量和友好方法。在任何一个与 Tom 在同一包的类中，也可以通过 Tom 类的类名访问 Tom 类的友好成员变量和友好方法。

假如 Jerry 与 Tom 是同一个包中的类，那么下述 Jerry 类中的 cat.weight、cat.f(3,4)都是合法的：

```
class Jerry {
    void g() {
        Tom cat = new Tom();
        cat.weight = 23f;                     //合法
        float sum = cat.f(3,4);               //合法
    }
}
```

在源文件中编写命名的类总是在同一包中的。如果源文件使用 import 语句引入了另外一个包中的类，并用该类创建了一个对象，那么该类的这个对象将不能访问自己的友好变量和友好方法。

▶ 5.11.5 受保护的成员变量和方法

用 protected 修饰的成员变量和方法被称为受保护的成员变量和受保护的方法。例如：

```
class Tom {
    protected float weight;                   //weight 是 protected 的 float 型变量
```

```
    protected float f(float a,float b) {          //方法 f 是 protected 方法
        return a + b;
    }
}
```

当在另外一个类中用类 Tom 创建了一个对象后，如果这个类与 Tom 类在同一个包中，那么该对象能访问自己的 protected 变量和 protected 方法。在任何一个与 Tom 类在同一个包中的类中，也可以通过 Tom 类的类名访问 Tom 类的 protected 类变量和 protected 类方法。

假如 Jerry 与 Tom 是同一个包中的类，那么下述 Jerry 类中的 cat. weight、cat. f(3,4)都是合法的：

```
class Jerry {
    void g() {
        Tom cat = new Tom();
        cat.weight = 23f;                         //合法
        float sum = cat.f(3,4);                   //合法
    }
}
```

注：在后面讲述子类时，将讲述“受保护(protected)”和“友好”之间的区别。

▶ 5.11.6 public 类与友好类

类声明时，如果在关键字 class 前面加上 public 关键字，就称这样的类是一个 public 类。例如：

```
public class A
{ …
}
```

可以在任何另外一个类中，使用 public 类创建对象。如果一个类不加 public 修饰，例如：

```
class A
{…
}
```

这样的类被称作友好类，那么另外一个类中使用友好类创建对象时，要保证它们是在同一包中。

注：

(1) 不能用 protected 和 private 修饰类。

(2) 访问限制修饰符按访问权限从高到低的排列顺序是 public、protected、友好的、private。

视频讲解

5.12 基本类型的类包装

Java 的基本数据类型包括 byte、int、short、long、float、double、char。Java 同时也提供了基本数据类型相关的类，实现了对基本数据类型的封装。这些类在 java. lang 包中，分别是 Byte、Integer、Short、Long、Float、Double 和 Character 类。

5.12.1 Double 类和 Float 类

Double 类和 Float 类实现了对 double 和 float 基本型数据的类包装。

可以使用 Double 类的构造方法：

```
Double(double num)
```

创建一个 Double 类型的对象；使用 Float 类的构造方法：

```
Float(float num)
```

创建一个 Float 类型的对象。Double 对象调用 doubleValue()方法可以返回该对象含有的 double 型数据；Float 对象调用 floatValue()方法可以返回该对象含有的 float 型数据。

5.12.2 Byte 类、Short 类 、Integer 类、Long 类

下述构造方法分别创建 Byte、Integer、Short 和 Long 类型的对象：

```
Byte(byte num)
Short(short num)
Integer(int num)
Long(long num)
```

Byte、Short、Integer 和 Long 对象分别调用 byteValue ()、shortValue()、intValue()和 longValue ()方法返回该对象含有的基本型数据。

5.12.3 Character 类

Character 类实现了对 char 基本型数据的类包装。

可以使用 Character 类的构造方法：

```
Character(char c)
```

创建一个 Character 类型的对象。Character 对象调用 charValue()方法可以返回该对象含有的 char 型数据。

5.13 可变参数

视频讲解

可变参数(variable argument)是 JDK 1.5 新增的功能。可变参数是指在声明方法时不给出参数列表中从某项直至最后一项参数的名字和个数，但这些参数的类型必须相同。可变参数使用“…”表示若干个参数，这些参数的类型必须相同，最后一个参数必须是参数列表中的最后一个参数。例如：

```
public void f(int … x)
```

那么，方法 f 的参数列表中，从第 1 个至最后一个参数都是 int 型，但连续出现的 int 型参数的个数不确定。称 x 是方法 f 的参数列表中的可变参数的“参数代表”。

再如：

```
public void g(double a,int … x)
```

那么，方法 g 的参数列表中，第 1 个参数是 double 型，第 2 个至最后一个参数是 int 型，但连续出现的 int 型参数的个数不确定。称 x 是方法 g 的参数列表中的可变参数的“参数代表”。

参数代表可以通过下标运算来表示参数列表中的具体参数，即 x[0]、x[1]、……、x[m]分别表示 x 代表的第 1 个至第 m 个参数。例如，对于上述方法 g，x[0]、x[1]就是方法 g 的整个参数列表中的第 2 个参数和第 3 个参数。对于一个参数代表，如 x，x.length 等于 x 代表的参数的个数。参数代表非常类似自然语言中的“等”，英语中的 and so on。

对于类型相同的参数，如果参数的个数需要灵活的变化，那么使用参数代表可以使方法的调用更加灵活。例如，如果需要经常计算若干个整数的和，如：

```
1203 + 78 + 556,1 + 2 + 3 + 4 + 5,31 + 202 + 1101 + 1309 + 257 + 88
```

由于整数的个数经常需要变化，又无规律可循，那么就可以使用下列带可变参数的方法计算它们的和：

```
int getSum (int … x);
```

那么，

```
getSum (203,178,56,2098);
```

就可以返回 203、178、56、2098 的和。

在例 5.16 中，有两个 Java 源文件，分别是 Computer.java 和 Example5_16.java，其中，Computer 类中的 getSum()方法使用了参数代表，可以计算若干个整数的和。

【例 5.16】

Computer.java

```
public class Computer {
   public int getSum(int... x) {                    //x 是可变参数的参数代表
      int sum = 0;
      for(int i = 0;i < x.length;i++) {
         sum = sum + x[i];
      }
      return sum;
   }
}
```

Example5_16.java

```
public class Example5_16 {
   public static void main(String args[]) {
      Computer computer = new Computer();
      int result = computer.getSum(203,178,56,2098);
      System.out.println("1203,178,56,2098 的和:" + result);
      result = computer.getSum(66,12,5,89,2,51);
     System.out.println("66,12,5,89,2,51 的和:" + result);
   }
}
```

对于可变参数，Java 也提供了增强的 for 语句，允许按如下方式使用 for 语句遍历参数代表所代表的参数。

```
for(声明循环变量: 参数代表) {
    …
}
```

上述 for 语句的作用是对循环变量依次取参数代表所代表的每一个参数的值。因此，可以将上述例 5.16 中 Computer 类中的 for 循环语句更改为：

```
for(int param:x) {
    sum = sum + param;
}
```


视频讲解

5.14 var 局部变量

从 Java SE 10(JDK 10)版本开始，增加了“局部变量类型推断”这一新功能。

可以使用 var 在方法体内声明局部变量(局部变量知识点见 4.2.4 节)，而且必须显式地指定初值(初值不可以是 null)，那么编译器就可以推断出 var 所声明的变量的类型，即确定该变量的类型。var 不是真正意义的动态变量(运行时刻确定类型)，var 声明的变量也是在编译阶段就确定了类型。需要注意的是，方法的参数和方法的返回类型不可以用 var 来声明。

在类的类体中，不可以用 var 声明成员变量(成员变量知识点见 5.2.3 节)，var 是保留类型名称，但不是 Java 的关键字。从 JDK 10 开始，var 仍然也可用作变量或方法的名字。但是，var 不能再用作类或接口名称(如果您维护的代码需要使用 JDK 10 之后的环境，就需要修改类名或接口名是 var 的那部分代码)。

在例 5.17 中，Tom 类中的方法 f 使用 var 声明了局部变量。

【例 5.17】

Example5_17.java

```
import java.util.Date;
class Tom {
    void f(double m) {
        var width = 108;          //var 声明变量 width 并推断出是 int 型
        var height = m;           //var 声明变量 height 并推断出是 double 型
        var date = new Date(); //var 声明变量 date 并推断出是 Date 型
       //width = 3.14; 非法,因为 width 的类型已经确定为 int 型
       //var str ; 非法,没有显式地指定初值,无法推断 str 的类型
       //var what = null; 非法,无法推断 what 的类型
       System.out.printf("%d,%f,%s\n",width,height,date);
    }
}
public class Example5_17 {
    public static void main(String args[]){
        var tom = new Tom(); //var 声明变量 tom 并推断出是 Tom 型
        tom.f(6.18);
    }
}
```

注：从上下文推断局部变量类型可减少所需的代码量，但并不提高运行效率。编者认为，过多地使用 var 声明局部变量将增加软件测试维护人员的工作量，因为软件测试维护人员在阅读代码时，就要学着像编译器那样去推断局部变量的类型。

5.15 上机实践

▶ 5.15.1 机动车

1. 实验目的

本实验的目的是让学生使用类来封装对象的属性和功能。

2. 实验要求

编写一个 Java 应用程序，该程序中有两个类：Vehicle(用于刻画机动车)和 User(主类)。Vehicle 类有一个 double 型变量 speed，用于刻画机动车的速度；有一个 int 型变量 power，用于刻画机动车的功率。Vehicle 类中定义了 speedUp(int s)方法，体现机动车有加速功能；定义了 speedDown()方法，体现机动车有减速功能；定义了 setPower(int p)方法，用于设置机动车的功率；定义了 getPower()方法，用于获取机动车的功率。在主类 User 的 main 方法中用 Vehicle 类创建对象，并让该对象调用方法设置功率，演示加速和减速功能。

3. 程序模板

请按照模板要求，将【代码】替换为 Java 程序代码。

Vehicle.java

```
public class Vehicle {
    【代码 1】//声明 double 型变量 speed,刻画速度
    【代码 2】//声明 int 型变量 power,刻画功率
    void speedUp(int s) {
       【代码 3】    //将参数 s 的值与成员变量 speed 的和赋值给成员变量 speed
    }
    void speedDown(int d) {
       【代码 4】    //将成员变量 speed 与参数 d 的差赋值给成员变量 speed
    }
    void setPower(int p) {
       【代码 5】    //将参数 p 的值赋值给成员变量 power
    }
    int getPower() {
       【代码 6】    //返回成员变量 power 的值
    }
    double getSpeed() {
       return speed;
    }
}
```

User.java

```
public class User {
   public static void main(String args[]) {
      Vehicle car1,car2;
```

```
        【代码 7】//使用 new 运算符和默认的构造方法创建对象 car1
        【代码 8】//使用 new 运算符和默认的构造方法创建对象 car2
        car1.setPower(128);
        car2.setPower(76);
        System.out.println("car1 的功率是:" + car1.getPower());
        System.out.println("car2 的功率是:" + car2.getPower());
        【代码 9】//car1 调用 speedUp 方法将自己的 speed 的值增加 80
        【代码 10】//car2 调用 speedUp 方法将自己的 speed 的值增加 100
        System.out.println("car1 目前的速度:" + car1.getSpeed());
        System.out.println("car2 目前的速度:" + car2.getSpeed());
        car1.speedDown(10);
        car2.speedDown(20);
        System.out.println("car1 目前的速度:" + car1.getSpeed());
        System.out.println("car2 目前的速度:" + car2.getSpeed());
    }
}
```

4. 实验指导

1) 避免使用空对象

创建一个对象时,成员变量被分配内存空间,这些内存空间称为该对象的实体或变量;而对象中存放着引用,以确保这些变量由该对象操作使用。空对象不能使用,即不能让一个空对象去调用方法产生行为。假如程序中使用了空对象,在运行时会出现 NullPointerException 异常。对象是动态地分配实体,Java 的编译器对空对象不做检查。因此,在编写程序时要避免使用空对象。

2) 参考答案

【代码 1】: double speed;

【代码 2】: int power;

【代码 3】: speed = speed + s;

【代码 4】: speed = speed - d;

【代码 5】: power = p;

【代码 6】: return power;

【代码 7】: car1 = new Vehicle();

【代码 8】: car2 = new Vehicle();

【代码 9】: car1.speedUp(80);

【代码 10】: car2.speedUp(100);

5. 实验后的练习

(1) 改进 speedUP 方法,使 Vehicle 类的对象加速时 speed 值不能超过 200。

(2) 改进 speedDown 方法,使 Vehicle 类的对象减速时 speed 值不能小于 0。

(3) 增加一个刹车方法 void brake(),Vehicle 类的对象调用它能将 speed 的值变为 0。

▶ 5.15.2 电视机

1. 实验目的

本实验的目的是让学生掌握对象的组合以及参数传递。

2. 实验要求

编写一个 Java 应用程序，模拟家庭买一台电视，即家庭将电视作为自己的一个成员，并通过调用一个方法将某个电视的引用传递给自己的电视成员。有三个源文件：TV.java、Family.java 和 MainClass.java。其中，TV.java 中的 TV 类负责创建“电视”对象；Family.java 中的 Family 类负责创建“家庭”对象；MainClass.java 是主类。在主类的 main 方法中首先使用 TV 类创建一个对象 haierTV，然后使用 Family 类再创建一个对象 zhangSanFamily，并将先前 TV 类的实例 haierTV 的引用传递给 zhangSanFamily 对象的成员变量 homeTV。

3. 程序模板

请按照模板要求，将【代码】替换为 Java 程序代码。

TV.java

```
public class TV {
    int channel;  //电视频道
    void setChannel(int m) {
       if(m>=1){
          channel=m;
       }
    }
    int getChannel(){
       return channel;
    }
    void showProgram(){
       switch(channel) {
          case 1 : System.out.println("综合频道");
                   break;
          case 2 : System.out.println("经济频道");
                   break;
          case 3 : System.out.println("文艺频道");
                   break;
          case 4 : System.out.println("国际频道");
                   break;
          case 5 : System.out.println("体育频道");
                   break;
          default : System.out.println("不能收看"+channel+"频道");
       }
    }
}
```

Family.java

```
public class Family {
    TV homeTV;
    void buyTV(TV tv) {
       【代码 1】                    //将参数 tv 赋值给 homeTV
    }
    void remoteControl(int m) {
       homeTV.setChannel(m);
    }
    void seeTV() {
       homeTV.showProgram();      //homeTV 调用 showProgram()方法
    }
}
```

MainClass. java

```
public class MainClass {
   public static void main(String args[]) {
       TV haierTV = new TV();
       【代码 2】                    //haierTV 调用 setChannel(int m)方法,并向参数 m 传递 5
       System.out.println("haierTV 的频道是" + haierTV.getChannel());
       Family zhangSanFamily = new Family();
       【代码 3】           //zhangSanFamily 调用 void buyTV(TV tv)方法,并将 haierTV 传递给参数 TV
       System.out.println("zhangSanFamily 开始看电视节目");
       zhangSanFamily.seeTV();
       int m = 2;
       System.out.println("zhangSanFamily 将电视更换到" + m + "频道");
       zhangSanFamily.remoteControl(m);
       System.out.println("haierTV 的频道是" + haierTV.getChannel());
       System.out.println("zhangSanFamily 再看电视节目");
       zhangSanFamily.seeTV();
    }
}
```

4. 实验指导

1) 黑盒复用

通过组合对象来复用方法也称“黑盒复用”,因为当前对象只能委托所包含的对象调用其方法,这样一来,当前对象对所包含的对象的方法的细节是一无所知的。

2) 参考答案

【代码 1】: homeTV = tv;

【代码 2】: haierTV.setChannel(5);

【代码 3】: zhangSanFamily.buyTV(haierTV);

5. 实验后的练习

(1) 省略【代码 2】程序能否通过编译? 若能通过编译,程序输出的结果是怎样的?

(2) 在主类的 main 方法的最后增添下列代码,并解释运行效果。

```
Family lisiFamily = new Family();
lisiFamily.buyTV(haierTV);
lisiFamily.seeTV();
```

▶ 5.15.3 森林

1. 实验目的

类变量是与类相关联的数据变量,而实例变量是仅仅和对象相关联的数据变量。不同的对象的实例变量将被分配不同的内存空间,如果类中有类变量,那么所有对象的这个类变量都分配给相同的一处内存,改变其中一个对象的这个类变量会影响其他对象的这个类变量。也就是说,对象共享类变量。类中的方法可以操作成员变量,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,方法中出现的类变量也是该对象的变量,只不过这个变量和所有的其他对象共享而已。实例方法可操作实例成员变量和静态成员变量;静态方法只能操作静态成员变量。

本实验的目的是让学生掌握类变量与实例变量,以及类方法与实例方法的区别。

2. 实验要求

编写程序模拟两个村庄共同拥有一片森林。编写一个 Village 类，该类有一个静态的 int 型成员变量 treeAmount 用于模拟森林中树木的数量。在主类 MainClass 的 main 方法中创建两个村庄，一个村庄改变了 treeAmount 的值，另一个村庄查看 treeAmount 的值。程序运行参考效果如图 5.26 所示。

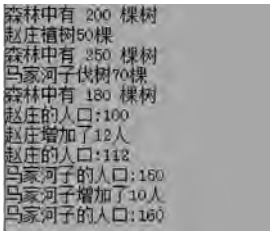

图 5.26　村庄共享森林

3. 程序模板

请按照模板要求，将【代码】替换为 Java 程序代码。

Village.java

```
class Village {
    static int treeAmount;      //模拟森林中树木的数量
    int peopleNumber;           //村庄的人数
    String name;                //村庄的名字
    Village(String s) {
        name = s;
    }
    void treePlanting(int n){
        treeAmount = treeAmount + n;
        System.out.println(name + "植树" + n + "棵");
    }
    void fellTree(int n){
        if(treeAmount - n >= 0){
           treeAmount = treeAmount - n;
           System.out.println(name + "伐树" + n + "棵");
        }
        else {
           System.out.println("无树木可伐");
        }
    }
    static int lookTreeAmount() {
       return treeAmount;
    }
    void addPeopleNumber(int n) {
       peopleNumber = peopleNumber + n;
       System.out.println(name + "增加了" + n + "人");
   }
}
```

MainClass.java

```
public class MainClass {
   public static void main(String args[]) {
        Village zhaoZhuang,maJiaHeZi;
        zhaoZhuang = new Village("赵庄");
        maJiaHeZi = new Village("马家河子");
        zhaoZhuang.peopleNumber = 100;
        maJiaHeZi.peopleNumber = 150;
        【代码 1】//用类名 Village 访问 treeAmount,并赋值 200
        int leftTree = Village.treeAmount;
        System.out.println("森林中有 " + leftTree + " 棵树");
        【代码 2】//zhaoZhuang 调用 treePlanting(int n),并向参数传值 50
```

```
        leftTree =【代码 3】//maJiaHeZi 调用 lookTreeAmount()方法得到树木的数量
        System.out.println("森林中有 " + leftTree + " 棵树");
        【代码 4】maJiaHeZi 调用 fellTree(int n),并向参数传值 70
        leftTree = Village.lookTreeAmount();
        System.out.println("森林中有 " + leftTree + " 棵树");
        System.out.println("赵庄的人口:" + zhaoZhuang.peopleNumber);
        zhaoZhuang.addPeopleNumber(12);
        System.out.println("赵庄的人口:" + zhaoZhuang.peopleNumber);
        System.out.println("马家河子的人口:" + maJiaHeZi.peopleNumber);
        maJiaHeZi.addPeopleNumber(10);
        System.out.println("马家河子的人口:" + maJiaHeZi.peopleNumber);
    }
}
```

4. 实验指导

1) 关于共享变量

对象共享类变量,在【代码 1】之前已经有了 zhaoZhuang 对象,这个时候,【代码 1】用“Village.treeAmount=200;”或“zhaoZhuang.treeAmount=200;”替换都是正确的。

2) 参考答案

【代码 1】: `Village.treeAmount = 200;`

【代码 2】: `zhaoZhuang.treePlanting (50);`

【代码 3】: `maJiaHeZi.lookTreeAmount();`

【代码 4】: `zhaoZhuang.fellTree (70);`

5. 实验后的练习

【代码 2】是否可以写成“Village.treePlanting(50);”?

5.16 课外读物

课外读物均来自编者的教学辅助微信公众号 java-violin,扫描二维码即可学习观看。

(1) 守株待兔。

(2) 调虎离山。

(3) 请女朋友吃海鲜。

(4) 击鼓传花。


课外读物(1)


课外读物(2)


课外读物(3)


课外读物(4)

习题

1. 类中的实例变量在什么时候会被分配内存空间?
2. 什么叫方法的重载? 构造方法可以重载吗?
3. 类中的实例方法可以操作类变量(static 变量)吗? 类方法(static 方法)可以操作实例

变量吗？

4. 类中的实例方法可以用类名直接调用吗？

5. 简述类变量和实例变量的区别。

6. 下列类声明错误的是_____。

A. class A　　B. public class A

C. protected class A　　D. private class A

7. 下列 A 类的类体中【代码 1】～【代码 5】哪些是错误的？

```
class Tom {
  private int x = 120;
  protected int y = 20;
  int z = 11;
  private void f() {
     x = 200;
     System.out.println(x);
  }
  void g() {
     x = 200;
     System.out.println(x);
  }
}
public class A {
  public static void main(String args[]) {
      Tom tom = new Tom();
       tom.x = 22;                    //【代码 1】
       tom.y = 33;                    //【代码 2】
       tom.z = 55;                    //【代码 3】
       tom.f();                       //【代码 4】
       tom.g();                       //【代码 5】
   }
}
```

8. 请说出 A 类中 System.out.println 的输出结果。

```
class B
{   int x = 100,y = 200;
    public void setX(int x)
    {   x = x;
    }
    public void setY(int y)
    { this.y = y;
    }
    public int getXYSum()
    { return x + y;
    }
}
public class A
{   public static void main(String args[])
    {   B b = new B();
        b.setX( - 100);
        b.setY( - 200);
```

```
        System.out.println("sum = " + b.getXYSum());
    }
}
```

9. 请说出 A 类中 System.out.println 的输出结果。

```
class B {
    int n;
    static int sum = 0;
    void setN(int n) {
      this.n = n;
    }
    int getSum() {
      for(int i = 1;i <= n;i++)
         sum = sum + i;
       return sum;
    }
}
public class A {
    public static void main(String args[]) {
       B b1 = new B(),b2 = new B();
       b1.setN(3);
       b2.setN(5);
       int s1 = b1.getSum();
       int s2 = b2.getSum();
       System.out.println(s1 + s2);
    }
}
```

10. 请说出 E 类中 System.out.println 的输出结果。

```
class A {
    double f(int x,double y) {
       return x + y;
    }
    int f(int x,int y) {
       return x * y;
    }
}
public class E {
    public static void main(String args[]) {
       A a = new A();
       System.out.println(a.f(10,10));
       System.out.println(a.f(10,10.0));
  }
}
```