第 3 章　Java 语言面向对象程序设计

本章主要内容：类是面向对象程序的主体，也是对程序代码进行模块化管理的一个重要手段。Java 语言采用包来管理类和类名空间。本章主要介绍使用 Java 语言实现面向对象程序设计的基本概念和相关语法，包括对象的封装和信息的隐藏、类的继承、程序设计的多态、Java 语言的单一继承机制、实现多重继承功能的接口概念、完整的 Java Application 程序的基本结构和基本规范，并简要介绍 Java 类库中几个比较常用的类。

3.1　面向对象程序设计

3.1.1　程序设计思想：结构化与面向对象

自高级程序设计语言诞生以后，使用高级程序设计语言开发软件就成为一个新兴的项目领域，并且得到了蓬勃发展。大约在 20 世纪 70 年代，随着软件开发的规模越来越大，参与开发项目的人越来越多，程序代码动辄几百万行，开发者们发现相互之间的协调越来越难，在软件出现问题时，查找错误甚至比开发一个软件还要困难。因此，人们开始注意研究软件设计的方法和思想。结构化程序设计和面向对象程序设计的思想先后产生。

实际上，早在 20 世纪 60 年代后期就出现了类和对象的概念，将类作为语言机制来封装数据和对数据的相关操作。20 世纪 70 年代前期，随着 Smalltalk 语言的出现，面向对象的方法逐渐被引入软件开发中。自 80 年代中期到 90 年代，面向对象的研究重点已经从语言方面转移到设计方法学方面，尽管还不成熟，但已经陆续提出了一些面向对象的开发方法和设计技术。目前，面向对象的方法和思想已经成为软件工程领域中的重要设计思想。

面向对象程序设计是建立在“对象”概念基础上的方法学，是将描述客观事物的数据和处理数据的方法封装为一体的设计方式，程序代码与客观实体具有更贴切的对应关系。采用面向对象程序设计方法带来了以下 3 个方面的好处。

一是可以将一个大的程序分解为若干个小的相对独立的程序，降低了关联度和耦合度。这样一来，就规避了大型程序设计中的检查难、协调难问题，毕竟一个小的程序在出现问题时更容易复查。

二是促进了软件重用技术的应用和发展。在不同的软件开发过程中，总会有一些功能是重复出现的，可以使用标准的开发方法创建程序。在面对一个新的项目时，可以使用现有的方法作为模块来搭建项目。对于连续开发多个项目的情形来说，这种方法往往能够加快开发进度。

三是避免重复开发。在软件开发中经常会遇到这样的事情，有些功能模块会重复出现，采用面向对象程序设计方法可以将这样的部分提取出来，实现一次开发、多次使用。

Java 语言是一种完全的面向对象的程序设计语言，它是在 C++语言基础上改进而成的，

但它比 C++语言更彻底地实现了面向对象程序设计方式。Java 语言用类来实现对代码的管理，实现了完全的面向对象的程序设计。Java 语言对代码采用二级模块管理：一级模块是类；二级模块是方法。类在 Java 语言中的作用有两个：一个作用是作为面向对象程序设计的主体概念；另一个作用是作为组织程序代码的基本单位，即程序代码的一级模块。

3.1.2 封闭：对象、类和消息

在面向对象程序设计方法中，对象就是描述事物的数据与处理数据的方法的集合。面向对象程序设计实现了对象的封装，实现了模块化和信息隐藏，有利于程序的可移植性和安全性。类是对客观事物的抽象，是在程序中统一定义的对象，定义了所有对象所共有的内容，是程序中若干个对象所具有的共性，是对象的模板。将类实例化即可生成对象。在程序中，各个对象之间采用消息进行交互。消息是对象之间的交互方式和交互内容。消息包括消息的接收者、接收对象应采用的应对方法和执行方法所需要的参数 3 个方面的内容。

3.1.3 继承

继承是面向对象程序设计方法中的重要内容，通过类的继承可以实现程序代码的重复利用，也可以在原有代码的基础上进行改进。

Java 语言不支持类的多重继承，只支持类的单一继承，使类的继承机制大大简化。学习过 C++语言的人都有一个体会，该语言的多重继承语法非常复杂和烦琐，往往令人十分头疼。Java 语言在继承机制上的改进，避免了 C++语言中因多重继承而导致的弊病，以及因此所定义的很多关联概念，使得 Java 语言的语法更为简洁和清晰。由于实行了类的单一继承，因此每个经过继承所得到的派生类只有一个信息来源，继承关系十分明确，使得 Java 语言的类按照其继承层次形成了一个树状的继承关系图，不妨称其为“类树”。在摒弃类的多重继承的同时，Java 语言支持接口，可以通过接口之间的多重继承和接口在类中的实现完成多重继承的功能。

3.1.4 多态

多态是面向对象程序设计方法中最具有灵活性和技巧性的内容。通过多态，程序员可以使程序具有很多灵活的适应性，完成各种具有自适应能力的设计。Java 语言实现多态的手段有：方法重载和方法重写、用子类的构造方法初始化父类的对象实例、抽象类和抽象方法、接口等。在 Java 语言中，多态的内容与类继承的内容是交叉、融合的。

3.2 类与对象

3.2.1 类声明

在 Java 语言中，一个基本的类声明格式如下：

```
[public][abstract|final] class classname
```

```
{
classbody
}
```

其中，class 是类声明关键字，任何类声明都需要使用这个关键字。classname 代表这个类的名称，是一个符合 Java 语法要求的标识符。前面用一对方括号标记的部分是可选项，采用不同的可选项可代表不同的含义。如果这个类被定义为程序的主类，就需要在类的前面使用 public 关键字修饰。对于 Java Application 程序而言，主类中的主方法是程序执行的入口点和出口点。我们在第 1 章和第 2 章中已经见过比较简单的 Java 语言程序的例子，其中的类都是程序的主类。在第二个方括号中，我们采用一条竖线将两个关键字 abstract 和 final 分开，表示这两个关键字修饰的意义是不相容的，在这个位置上只能二选一，不能同时使用。如果使用 abstract 关键字修饰，这个类就被定义成了抽象类；如果使用 final 关键字修饰，这个类就被定义成了最终类。这两个概念将在 3.5 节中介绍。classbody 代表类的类体，类体是类的定义的实际内容部分。在类体中，包含对变量成员和方法成员的声明和定义，还可以对所有的成员分别设定访问权限，以限定其他对象对它的访问。Java 语言的访问权限将在 3.4 节中介绍。

3.2.2 变量成员

在 Java 语言的类中，变量是类的属性的载体。

变量成员的基本声明格式如下：

```
[public|protected|private][static][final][transient][volatile]
type variableName;
```

其中，type 代表一种数据类型；variableName 代表作为变量成员名的标识符，只要符合 Java 语法要求的标识符，都可以作为变量成员的名称。如同第 2 章中已经介绍过的，推荐采用英文单词序列作为类变量成员的名称，习惯上首单词全部小写，后面的单词首字母大写。

变量成员的类型可以是基本数据类型，也可以是引用数据类型，其名称在同一个类的变量成员中必须是唯一的，不能与其他变量成员同名，但可以和同一个类内的某个方法成员使用同一个名称。在一个类中可以根据需要声明多个变量成员，也可以没有变量成员，但是实际工作中很少出现没有变量成员的情况。

我们可以为变量成员设定访问权限，如 public、protected、private 等均是设定访问权限的关键字，后面将对它们进行专门介绍。变量成员可以用 static、final、transient、volatile 等关键字来修饰。static 的含义将在 3.4 节中介绍；final 用来声明一个标识符常量，其含义将在 3.5 节中介绍；transient 用来声明一个暂时性变量；volatile 用来声明一个由多个并发线程共享的变量。

在声明的基础上，变量成员在使用之前必须有创建的过程，也要有初始化的过程。显式地给出变量成员的初始值，称为显式初始化。如果类的变量成员没有进行显式初始化，则系统会对其进行自动初始化，自动初始化的规则参见表 2.4。

3.2.3 方法成员

在 Java 语言的类中，方法是类的行为的载体。

方法成员的基本声明格式如下：

```
[public|protected|private][static][abstract|final][native][synchronized]
returnType methodName([paramList])
{
    methodbody
}
```

其中，returnType 代表方法的返回类型，可以是基本数据类型，也可以是引用数据类型，当方法成员无返回值时，还可以是 void 型。methodName 代表方法的名称，是一个符合 Java 语法要求的标识符，推荐使用以动词开头的英文单词序列作为方法名称，并且首单词全部小写，后面的单词首字母大写。paramList 代表参数列表，在 Java 语言的方法定义中，允许定义有参数的方法，也允许定义无参数的方法。参数列表中的参数可以是各种数据类型，包括基本数据类型和引用数据类型，参数的个数由程序员根据需要来确定。参数列表中的每个参数都必须明确声明数据类型，参数声明之间用逗号分隔。在一个类中可以通过声明定义任意个方法成员。在实际工作中，一个类中可能会出现几十个甚至上百个方法成员。

我们可以为方法成员设定访问权限，如 public、protected、private 等均是设定访问权限的关键字，后面将对它们进行专门介绍。可以使用 static，final 和 abstract 中的一个，以及 native 和 synchronized 等关键字来修饰方法成员。static、final 和 abstract 的含义将在 3.4 节、3.5 节中讲解；native 用来声明一个本地方法；synchronized 用来声明一个并行方法。

methodbody 代表方法体。方法体是方法的实现，其中可以包含局部变量的声明和合法的 Java 执行语句。从软件设计上来说，方法最重要的任务是描述一个算法的计算逻辑。事实上，方法体是 Java 语言程序代码的实质部分，程序中所有的执行动作都是在方法体中实现的。一个方法的方法体就是一段完成一定功能的程序，这段程序的设计方式通常遵循结构化程序设计的方式。结构化程序设计是面向对象程序设计的基本前提，这一点恰恰体现在方法体的定义上。至于在方法体中需要定义什么样的内容，则是由程序员根据设计意图来确定的。按照面向对象程序设计的指导思想，应该尽量把软件的功能分得细一些，最好是根据一个功能定义一个方法，避免将超过一个以上的功能用一个方法实现。

方法的返回类型与方法体内部的语句之间应该有一个关联关系。Java 语言对方法体有一个规定：如果方法的返回类型不为 void 型，则在方法体中必须包含 return 语句，用于返回一个与方法的返回类型相同的值，并且在程序的执行流程中，必须保证有一条 return 语句被执行；如果方法的返回类型为 void 型，则在方法体中不必包含 return 语句。总而言之，如果一个方法是有返回类型的，就必须能够向外传递一个与其返回类型相同的值；如果一个方法没有返回类型，则无须向外传递任何值。有返回类型而没有执行 return 语句和返回类型为 void 型而返回了一个值，都算是一种语法错误。

return 语句在方法中的使用方式如下：

```
return expression;
```

其中，expression 代表一个表达式，可以是任何符合 Java 语法要求的表达式，也可以

是一个简单的变量。表达式的计算结果的数据类型应该与方法的返回类型一致。这种方式的 return 语句既可以向调用方法的语句返回一个确切的值，又可以返回控制。

return 语句的另一种使用方式如下：

```
return;
```

此时的 return 语句不返回任何数值，只向调用方法的语句返回控制。

3.2.4 声明的作用域

作用域是程序设计的一个重要概念。所谓声明的作用域，就是程序中可以通过正常的方式引用所声明的程序实体的代码范围。Java 语言的作用域分为类级、方法级、语句块级、语句级，具体来说，有以下基本规则。

（1）在类体中声明的变量成员和方法成员的作用域是整个类。可以在类中直接访问在该类中声明的变量成员和方法成员，以及该类从其父类继承的方法。

（2）在方法中声明的参数的作用域是整个方法体；在方法中所有语句之外声明的变量的作用域是整个方法体。

（3）在语句块中声明的局部变量的作用域是该语句块。

（4）在语句中声明的变量的作用域是该语句。例如，在第 2 章程序清单 2.5 的程序中，在 for 循环语句中声明的循环控制变量的作用域是该 for 循环语句。

特别提示，在 Java 语言中没有全局作用域。

3.2.5 主类和主方法

在 Java 语言程序中，可以定义一个特殊的类，称为程序的主类。主类需要在类声明的前面用 public 关键字修饰。对于 Java Application 程序而言，主类中的主方法是程序执行的入口点和出口点；对于 Java Applet 程序而言，主类不具有这个特性。

Java Application 程序的主类中有一个特殊的方法成员，称为主方法。主方法的名称与类型声明都是固定的。例如：

```
public static void main(String args[])
{
    methodbody
}
```

主方法是在 Java Application 程序执行时第一个被访问的方法，也是最后一个被退出的方法。主方法是公有方法、静态方法、无返回值的方法。

主方法只能出现一个，不允许重载。

3.2.6 构造方法

在 Java 语言的类中，还有一个特殊的方法成员，称为构造方法。构造方法是用来初始化类对象的。构造方法的方法名与类名相同，并且无返回类型，构造方法只能使用 new 关键字调用。我们可以在定义类时像定义一般的方法一样定义构造方法，如果在定义类时没

有定义构造方法，则 Java 系统会自动提供默认的无参数构造方法。所以，无论程序中是否显式地定义了构造方法，在类中都将存在构造方法。构造方法在类定义中经常被定义多个，即它可以是重载的。也可以将构造方法的方法体定义为空，当程序执行到空的构造方法时，将自动调用该类的父类的构造方法。

3.2.7 finalize()方法

finalize()方法是在 Object 类中定义的，其作用是释放对象实例所占用的系统资源，一般会在自动垃圾收集之前由系统自动调用。由于 Object 类是所有 Java 类的父类，所以 finalize()方法会被所有的 Java 类继承，程序员无须在类中显式地定义该方法，一般也无须重写该方法。Object 类的详细情况将在本章后面专门讲授。

3.2.8 方法重载

方法重载（Overloading）是 Java 语言实现多态的手段之一。

方法重载是指在一个类中可以定义多个名称相同的方法，这些同名方法的访问权限可以相同，返回类型可以相同，但是参数列表不能相同。参数列表的不同体现在参数的个数不同，或者虽然参数的个数相同，但是参数中至少有一个处于同一位置上的参数的类型不同。在调用方法时，系统通过使用参数数量和类型的不同组合来确定调用的是哪一个方法。重载的方法的方法体一般是不同的，相当于定义了使用同样的方法名的不同方法。在实际工作中，方法重载一般用于采用不同的途径完成相同或相似目的的多个方法的定义。

注意，Java 系统无法分辨名称相同、参数列表相同，而返回类型不同的两个方法成员，一旦出现这种情况，将导致一个编译错误。

一般来说，在类中声明的方法成员都允许进行重载，常见的方法重载是构造方法的重载。有兴趣的读者可以查阅一下 Java 类库，看看在类库中的 Java 类中是如何进行方法重载的，只要稍加留意就会发现，方法重载，特别是构造方法的重载，在 Java 类库中非常普遍。

下面介绍一个类的声明和定义的例子。

【例 3.1】利用 Java 类库中的 Point 类定义三角形的例子。

具体的程序如程序清单 3.1 所示。

程序清单 3.1

```
//Example 1 of Chapter 3

import java.awt.Point;

class Triangle {

    //定义三角形的 3 个顶点
    protected Point X1,X2,X3;

    //无参数的构造方法
```

```
public Triangle()
{
    //空的构造方法，隐含访问 Object 类的构造方法
}

//有参数的构造方法
public Triangle(Point a,Point b,Point c)
{
    X1 = a;
    X2 = b;
    X3 = c;
}

//设置第一个顶点
public void setX1(Point a)
{
    X1 = a;
}

//获取第一个顶点
public Point getX1()
{
    return X1;
}

//设置第二个顶点
public void setX2(Point b)
{
    X2 = b;
}

//获取第二个顶点
public Point getX2()
{
    return X2;
}

//设置第三个顶点
public void setX3(Point c)
{
    X3 = c;
}

//获取第三个顶点
```

```
    public Point getX3()
    {
        return X3;
    }

    //获取三角形的字符串表示
    public String toString()
    {
        return "["+X1.x+","+X1.y+"]"+"\n"+"["+X2.x+","+X2.y+"]"+"\n"+
"["+X3.x+","+X3.y+"]";
    }

    //计算三角形的面积
    public double getTriangleArea()
    {
        //定义三角形的 3 条边
        double a,b,c;
        //定义三角形的 3 条边之和的一半
        double s;
        //定义三角形的面积
        double S;

        a=Math.sqrt((X1.x-X2.x)*(X1.x-X2.x)+(X1.y-X2.y)*(X1.y-X2.y));
        b=Math.sqrt((X2.x-X3.x)*(X2.x-X3.x)+(X2.y-X3.y)*(X2.y-X3.y));
        c=Math.sqrt((X1.x-X3.x)*(X1.x-X3.x)+(X1.y-X3.y)*(X1.y-X3.y));

        s=(a+b+c)/2;

        //计算三角形的面积
        S=Math.sqrt(s*(s-a)*(s-b)*(s-c));

        //返回三角形的面积
        return S;
    }

}
```

我们看到，程序的开头使用 import 语句引入了 Java 类库中的 Point 类。这个类是一个关于平面上的点的类，其中包含两个 int 型变量成员，作为两个平面坐标。关于 import 语句，我们将在 3.3 节中介绍。在程序中定义了一个 Triangle 类作为对三角形的描述，其中包含 3 个 Point 类的对象成员，作为三角形的 3 个顶点。在定义构造方法时，实现了构造方法的重载，定义了一个无参数的构造方法，其中不包含任何可执行语句。在这种情况下，程序将默认使用 Triangle 类的父类的构造方法，即 Object 类的构造方法。在另一个构造方法

中，将 3 个实参的值传递给了类的 3 个成员。Triangle 类中一共定义了 8 个方法成员，有 3 个以 set 命名的方法成员负责对 3 个对象成员进行设置，还有 3 个以 get 命名的方法成员负责获取 3 个对象成员的值，这种使用专门的方法成员来设置和获取变量成员的设计方法是一种规范的程序设计方法，后面还会介绍。方法成员 toString()负责将类的内容用字符串表示出来，在 Object 类中已经有定义。作为 Object 类的子类，Triangle 类定义 to String()方法实际上是在重写 Object 类的这个方法。方法成员 getTriangleArea()负责计算一个 Triangle 类所描述的三角形的面积，其中使用了海伦公式来计算三角形的面积。查看 getTriangleArea()方法的程序段，我们会发现，这是一个典型的结构化程序设计的例子。

关于例 3.1 还有以下几点说明。

（1）在程序代码中多次出现了 return 语句的使用实例，请读者注意每一条 return 语句所返回的数据类型与方法的返回类型之间的严格对应关系。

（2）在方法成员 getTriangleArea()中，我们有意识地使用了两个局部变量 s 和 S。虽然它们是一个字母，但是由于大小写不同，因此 Java 语言将它们看作不同的标识符。

（3）在程序代码中多处使用了注释信息，这是为了更好地理解程序，再次建议读者在编写程序时坚持这种好习惯。

（4）在方法成员 getTriangleArea()中，我们使用了 Math.sqrt()方法来计算平方根。本章的 3.9 节将专门介绍 Math 类。

3.2.9 对象

定义类是为了给出一个生成实例的模板，而对象就是类在程序中的实例化。一个类在被定义之后，只有生成了对象实例才可以被程序使用。在程序代码中，对象实例要经历生成、使用和清除 3 个阶段。

对象实例的生成包括声明、实例化和初始化 3 个步骤，格式如下：

```
type objectName = new type([paramList])
```

其中，type 代表某个已经存在的类的类型；objectName 代表一个对象实例标识符，与前面已经介绍的类和方法一样，对象实例标识符也应该是符合 Java 语法要求的标识符；new 关键字用于分配存储空间，完成实例化；type 构造方法用于执行初始化工作。

调用构造方法的结果是将该类的一个引用赋给对象实例标识符。

在程序中使用对象实例的目的是访问对象实例中的变量和方法，这都是通过取成员运算符“.”来实现的，其格式如下：

```
objectName.variable                    //引用变量
objectName.methodName([paramList])     //引用方法
```

当一个对象实例被使用完之后，其使命也就结束了，应该被清除以释放其占用的资源，这称为对象实例的清除。Java 运行时系统通过自动垃圾收集机制可以周期性地清除无用对象，释放内存。

例 3.2 在例 3.1 的基础上将 Triangle 类进行实例化，并进行了一些简单的应用计算。

【例 3.2】利用对象实例访问方法成员的例子。

具体的程序如程序清单 3.2 所示。程序的执行结果如图 3.1 所示。

程序清单 3.2

```
//Example 2 of Chapter 3

import javax.swing.JOptionPane;
import java.text.DecimalFormat;
import java.awt.Point;

class Triangle {

    //定义三角形的 3 个顶点
    protected Point X1,X2,X3;

    //无参数的构造方法
    public Triangle()
    {
        //空的构造方法，隐含访问 Object 类的构造方法
    }

    //有参数的构造方法
    public Triangle(Point a,Point b,Point c)
    {
        X1 = a;
        X2 = b;
        X3 = c;
    }

    //设置第一个顶点
    public void setX1(Point a)
    {
        X1 = a;
    }

    //获取第一个顶点
    public Point getX1()
    {
        return X1;
    }

    //设置第二个顶点
    public void setX2(Point b)
    {
        X2 = b;
    }
```

```
        //获取第二个顶点
        public Point getX2()
        {
            return X2;
        }

        //设置第三个顶点
        public void setX3(Point c)
        {
            X3 = c;
        }

        //获取第三个顶点
        public Point getX3()
        {
            return X3;
        }

        //获取三角形的字符串表示
        public String toString()
        {
            return "["+X1.x+","+X1.y+"]"+"\n"+"["+X2.x+","+X2.y+"]"+"\n"+
"["+X3.x+","+X3.y+"]";
        }

        //计算三角形的面积
        public double getTriangleArea()
        {
            //定义三角形的 3 条边
            double a,b,c;
            //定义三角形的 3 条边之和的一半
            double s;
            //定义三角形的面积
            double S;

            a=Math.sqrt((X1.x-X2.x)*(X1.x-X2.x)+(X1.y-X2.y)*(X1.y-X2.y));
            b=Math.sqrt((X2.x-X3.x)*(X2.x-X3.x)+(X2.y-X3.y)*(X2.y-X3.y));
            c=Math.sqrt((X1.x-X3.x)*(X1.x-X3.x)+(X1.y-X3.y)*(X1.y-X3.y));

            s=(a+b+c)/2;

            //计算三角形的面积
            S=Math.sqrt(s*(s-a)*(s-b)*(s-c));
```

```
        //返回三角形的面积
        return S;
    }

}

public class TriangleTest {

    public static void main(String[] args)
    {
        String output = "";

        //定义顶点
        Point a1,a2,a3;
        Point b1,b2,b3;
        a1 = new Point(0,0);
        a2 = new Point(30,0);
        a3 = new Point(30,40);
        b1 = new Point(10,10);
        b2 = new Point(40,50);
        b3 = new Point(0,100);

        //定义三角形
        Triangle t1,t2;
        t1 = new Triangle();
        t1.setX1(a1);
        t1.setX2(a2);
        t1.setX3(a3);

        t2 = new Triangle(b1,b2,b3);

        DecimalFormat twoDigits = new DecimalFormat("0.00");

        output += "第一个三角形的顶点为：\n"+t1.toString();
        output += "\n第一个三角形的面积为："+twoDigits.format(t1.getTriangleArea());
        output += "\n"+"第二个三角形的顶点为：\n"+t2.toString();
        output += "\n第二个三角形的面积为："+twoDigits.format(t2.getTriangleArea());

        JOptionPane.showMessageDialog(null,output);
        System.exit(0);

    }
}
```

图 3.1　程序清单 3.2 的程序的执行结果

在例 3.2 中，建立了一个主类 TriangleTest，其中包含主方法。在主方法中声明了两个 Triangle 类的对象实例；第一个对象实例先通过无参数的构造方法进行了实例化，再通过方法成员设置了类成员的初始值；第二个对象实例通过重载的有参数的构造方法进行了实例化并为类成员赋予了初始值。通过这两个对象实例，实现了对方法成员的访问，分别得到了两个三角形的字符表达和面积的数值。在使用 Triangle 类的构造方法之前，先使用 Point 类的构造方法对 6 个局部变量进行了初始化，读者应该能够理解其中的道理。另外，程序中使用了 Java 类库中的一个 DecimalFormat 类，其作用是定义一个十进制浮点数的输出格式。

3.2.10　源程序文件的组织与存储

按照上面的语法，可以编写 Java 语言的源程序代码，并且在源程序代码编写完成后，可以以系统文件的形式将其存储到外部设备上。类是 Java 语言程序代码存储的基本单元。一个文件中可以包含一个类，也可以包含多个类。当源程序文件被存储时，如果文件中包含主类，则必须以主类名作为文件基本名，以.java 作为文件扩展名；如果文件中不包含主类，则文件的基本名无限制，以.java 作为文件扩展名。

源程序代码在执行之前需要进行编译，并且在编译之后得到 Java 字节码文件。Java 字节码文件的基本名与源程序文件的基本名相同，但以.class 为扩展名。如果源程序文件中包含多个类，则每个类会各自生成一个独立的字节码文件，并且字节码文件的基本名与类名相同。

编程技巧提示 3.1　学会使用开发工具的在线提示

Java 基础类库为用户提供了大量的实用类，使用户在进行程序设计时可以借鉴使用。初学者可以多了解、多关注，并且学会在自己的程序中使用类库。而且用户不必记忆 Java 基础类库的内容，因为目前的开发工具都有在线提示功能。用户在安装开发工具软件之后，只要简单配置一下就可以使用这个方便、快捷的功能。

编程技巧提示 3.2　学会阅读编译提示信息

无论是使用命令行还是开发工具编译程序，系统在遇到程序中的错误时，都会输出相应的错误提示信息。错误提示信息一般会说明错误的类型、错误在源代码中的位置和错误原因等。虽然这些提示信息不能准确地指导程序员如何修改，但它们还是非常有价值的。

用户应该逐渐学会通过编译提示信息分析程序中的错误，找到错误发生的原因。

编程常见错误提示 3.1　变量成员的声明和赋值相混淆

初学者易犯的一个错误是：在程序中直接对变量成员进行赋值操作而忘记先声明变量成员。这通常是初学者把编写 C 语言程序的习惯带到编写 Java 语言程序中了，并且认为对变量成员进行赋值的操作中隐含了对变量成员的声明。用户应该尽量养成先声明、后赋值使用的习惯。

编程常见错误提示 3.2　定义构造方法时写返回类型

Java 语言的语法规定构造方法没有返回类型，要求构造方法不能给出返回类型。然而很多初学者容易忘记这个规则，在定义构造方法时下意识地写上了返回类型或 void 型。

编程常见错误提示 3.3　在方法体中重复定义变量成员

变量成员作为类的成员，其声明的作用域是整个类体。如果在方法体中声明了与类的变量成员同名的变量，则变量成员的作用将被覆盖，在方法体中访问到的将是在方法体中定义的局部变量。这可能会引起数据传递的错误。

3.3　包

在进行大型软件的开发时，可能会有很多人参与，写出很多的 Java 类，这些类的名称可能会出现重复。为了便于管理数目众多的类，也为了解决类的命名冲突问题，Java 引入了包的机制，以提供类的多重命名空间，同时负责类名空间的管理。将因不同目的而开发的类放在不同的包中，即使出现相同的类名，也可以很好地管理它们。包可以有一定的层次，这种层次实际上对应着外部存储器上的目录结构。有兴趣的读者可以参考 Java API 文档或者自己机器上安装的 JDK。在程序代码中，有两条与包的概念有关的执行语句：package 语句用于实现在程序中定义的类的存储；import 语句用于调用已经在包中存在的类。

3.3.1　package 语句

package 语句被放在 Java 源程序文件的第一行，用于指明该文件中定义的类被存放在哪个包中。程序中可以没有 package 语句，此时的类将被存放到当前包中。显然，在每个程序文件中，最多只能有一条 package 语句。

package 语句的格式如下：

```
package pkg1.[pkg2.[pkg3.…]];
```

其中，pkg1、pkg2 等代表不同的包的层次。程序员可以在外部存储器上根据需要定义不同的包，将所开发的类根据开发项目或使用目的存储到不同的包中。这也是一个管理数据和代码的好习惯。

3.3.2　import 语句

import 语句被放在 Java 源程序文件的 package 语句之后，以及 Java 类定义和接口定

义之前。package 语句和 import 语句都应该被放在类定义和接口定义模块之外。import 语句的作用是引入包中已经存在的类，以供程序使用，包括 JFC 标准类库和开发包中的类，其格式如下：

```
import pkg1[.pkg2…].classname|*;
```

其中，classname 代表所要引入的类的类名，用于引入包中类名指代的类；*是一个通配符，用于引入包中的所有类。当需要在同一个包中引入多个类时，无须针对每个类写一条 import 语句，只需写一条 import 语句，并使用*通配符代替具体的类名即可。如果需要从多个包中引入多个类，则需要使用多条 import 语句逐一引入。

Java 语言的各个版本都提供了相当数量的类，构成了 Java 类库。这些类用来定义 Java 语言的各种基本功能，是使用不同的包的层次来管理的，称为 Java API 包，有兴趣的读者可以查阅 Java API 文档，了解其中的内容。

3.4 成员的访问

在完成类的定义之后，就可以对类中定义的类成员，包括变量成员和方法成员进行访问了。一般而言，在类的内部可以自由地访问变量成员和方法成员，没有什么限制；在类的外部访问类的成员时，会受到访问权限的限制。另外，对于具有 static 属性的类成员来说，Java 语言对其访问方式和访问范围有一些特别的规定。

3.4.1 变量成员和方法成员的访问

按照面向对象程序设计的思想，把描述事物属性的信息与处理这些信息的方法封装在对象中，实现了信息的隐藏。实际上，类中的方法就是用来访问类中的属性的。在同一个类中，方法成员可以直接使用变量名来任意地访问变量成员，其形式就像访问一个普通的变量一样；在类的外部，对变量成员的访问要通过类的对象实例来实现。具体的形式如下：

```
objectName.variableName                    //引用变量
```

在类的内部，可以使用方法名实现在一个方法中对其他方法成员的访问；在类的外部，对方法成员的访问也要通过类的对象实例来实现。具体的形式如下：

```
objectName.methodName([paramList])      //引用方法
```

有一种程序设计方法是值得推荐的，即可以为类中的每一个变量成员都定义一个读取方法和设置方法，通常这样的方法被称为 setter 方法和 getter 方法，这也是软件工程理论一向强调的程序设计方法。这种程序设计方法的作用是，即使对于被设置为 private 访问权限的变量成员而言，只要将相应的 setter 方法和 getter 方法设置成 public 型，就可以在程序中通过调用 setter 方法和 getter 方法很容易地访问变量成员。同时这种程序设计方法避免了对变量成员的直接访问，更有利于隐藏对象信息。

3.4.2 形参和实参

下面讨论关于方法成员的访问的另一个问题——参数的传递。前文在介绍方法的定义时说过，在 Java 语言的方法定义中，允许定义有参数的方法，也允许定义无参数的方法。在方法定义时所使用的参数称为形参。在访问有参数的方法时，要给出与形参的个数和类型都一致的参数，这样的参数称为实参。访问的过程包含实参向形参传递值的动作。程序在执行时将使用实参代替形参，并把实参的值传递给方法，以便对实参进行预期的处理和计算。

Java 语言对参数的数量和出现顺序有非常严格的要求，在这一点上比 C++语言要严格得多。在程序中使用实参访问方法成员时要特别注意，一定要把实参的数量和顺序写正确，否则将会出现一个语法错误。在调用重载的方法时更要注意，一定要把实参的数量和顺序与将要调用的那个方法成员的形参严格对应，如果没有严格对应，则很可能会调用同名的重载方法，从而得不到预期的结果，而这种错误往往较难检查出来。

这里有一个重要的内容必须介绍，就是参数提升。所谓参数提升，是当进行方法访问时，如果实参的数据类型与形参的数据类型不一致，则需要将给定的实参强行转换为适当的数据类型，以便与形参的数据类型一致，再传递给方法进行计算。参数提升是在程序中自动进行的。将实参进行强制数据类型转换的规则称为提升规则。按照提升规则，可以在不损失数据的前提下，将一种数据类型转换为另一种数据类型。提升规则适用于两种情况：一种情况是在表达式中包含两个及两个以上的基本类型运算数，需要两个运算数的数据类型一致；另一种情况是在访问方法时，要将作为实参的基本数据类型的数据传递给方法的形参，需要实参和形参的数据类型一致。提升规则的基本思想是将数据存储长度较短的数据提升为数据存储长度较长的数据，这样可以保证在数据提升的过程中不会损失数据的存储精度。

Java 语言的 8 种基本数据类型允许进行的提升如表 3.1 所示，没有被收录在表中的基本数据类型转换都是不被允许的。

表 3.1　Java 语言的 8 种基本数据类型允许进行的提升

基本数据类型	允许进行的提升
boolean	不允许
char	int、long、float 和 double
byte	short、int、long、float 和 double
short	int、long、float 和 double
int	long、float 和 double
long	float 和 double
float	double
double	无

从表 3.1 中可以看到，boolean 型的数据不允许进行提升；double 型的数据由于使用 64 位存储，已经是最高数据存储长度的数据而无法提升了。参数提升规则的意义在于，在使用实参访问定义了形参的方法时，即使实参的数据类型与形参的数据类型不严格一致，也可以成功地进行访问。不过我们已经看到了，这种“不严格一致”只允许很小的差别，提升规则的数据类型转换是十分有限的，所以在进行方法访问时要特别注意，一定要把实参的数据类型与形参的数据类型匹配好。

3.4.3 this

在 Java 语言程序中，有时需要对当前的对象实例进行一些操作，而操作的代码往往位于对象实例的内部，这时需要使用一个特别的方式来指代当前的对象实例。

在 Java 语言中，每个对象实例都可以使用 this 关键字作为其自身的引用，这是 this 关键字的第一种用法。另外，在方法体中，还可以使用 this 关键字引用当前对象，调用对象成员。其格式如下：

```
this.variableName                    //引用变量
this.methodName([paramList])         //引用方法
```

此时，this 关键字代表当前对象，用于调用当前对象的成员。

关于 this 关键字的用法，不在这里给出例子了。在后面的章节中，我们会频繁地看到带有这两种使用形式的程序。

3.4.4 访问权限

Java 语言规定了 4 种类成员的访问权限，分别限定了允许对其进行访问的其他对象的范围。这 4 种访问权限分别为 private、protected、public 和 friendly，前面 3 种访问权限需要在程序中使用关键字明确声明，如果没有明确声明，则系统会默认定义为第四种访问权限，即 friendly 型。注意：friendly 并不是 Java 关键字，也无须在程序中出现。这 4 种访问权限的限定范围如表 3.2 所示。

表 3.2　4 种访问权限的限定范围

	同一个类中	同一个包中	不同包中的子类	不同包中的非子类
private	√			
protected	√	√	√	
public	√	√	√	√
friendly	√	√		

private 是私有型，凡是被声明为私有型的成员是不允许在类的外部被访问的，而只能在类的内部被访问，这样的成员通常是需要对外实行信息隐藏的部分。所以，在类以外的任何地方采用对象实例的引用来访问私有型成员都是不被允许的。另外，凡是被声明为私有型的成员都不能被其子类所继承。关于继承将在 3.5 节中介绍。

protected 是介于 private 型和 public 型的一种访问权限，具有这种访问权限的成员可以被继承，也可以被其子类访问，而且除了子类，还可以被同一个包中的类访问。

public 是公有型，是为了让外界对类进行了解而定义的。公有型成员是类中的开放部分，可以被任何类访问。无论是否在同一个包中，也无论是否有继承关系，其他的类都可以访问公有型成员，可以在类的内部直接访问，也可以在类的外部使用对象实例的引用访问。公有型成员可以被其子类继承。

friendly 是注重位置关系而不注重继承关系的一种访问类型。可以在类的内部和同一个包中对这种类型的成员进行访问，但是这种类型的成员不能被其子类访问。这种类型的成员也可以被继承。

3.4.5 static 属性：类变量成员和类方法成员

在类的定义中，还可以使用 static 关键字修饰其变量成员和方法成员，使其具有 static 属性。成员在具有 static 属性之后，其被访问的方式和范围将发生一些变化。

3.4.5.1 类变量成员和实例变量成员

使用 static 关键字声明的变量成员称为类变量成员；不使用 static 关键字声明的变量成员称为实例变量成员。类变量成员有时也称为静态变量成员，其声明格式如下：

```
static type classVariableName;
```

实例变量成员通常都有各自的内存区。对于不同的对象实例而言，其对应的变量成员有不同的值。实例变量必须在生成对象实例后通过对象实例名来访问。一旦使用 static 关键字声明了变量成员，该变量成员就成了类变量成员，其值存储在类定义共享内存区，将由多个对象实例共享同一个值。而类变量除了可以通过对象实例名访问，还可以通过类名直接访问。

3.4.5.2 类方法成员和实例方法成员

使用 static 关键字声明的方法成员称为类方法成员；不使用 static 关键字声明的方法成员称为实例方法成员。类方法成员有时也称为静态方法成员，其声明格式如下：

```
static returnType classMethodName([paramList])
{
    methodbody
}
```

与实例方法成员相比，类方法成员发生了较大的变化。一般的实例方法既可以访问实例变量和实例方法，又可以访问类变量和类方法；而类方法则只能访问类变量和类中的其他类方法，不能访问实例变量和实例方法。在访问实例方法时，必须在生成对象实例后通过对象实例名来访问；而类方法可以在不生成对象实例的情况下，通过类名直接访问。这可能就是将其称为“类”方法成员的原因。另外，在类方法成员中不能使用 this 关键字和 super 关键字。

结合本章前面所介绍的内容和本节的内容，可以看出，访问类的成员的方式有 3 种：第一种是在类的内部使用成员名直接访问；第二种是在类的外部使用对象实例名调用成员名访问；第三种是在类的外部使用类名调用成员名访问。

例 3.3 说明了类变量成员和类方法成员的使用，具体的程序如程序清单 3.3 所示。

【例 3.3】类变量成员和类方法成员的使用实例。

程序清单 3.3

```
//Example 3 of Chapter 3

public class SoftwareNumber
{
    private int serialNumber;
    public static int counter=0;
```

```
    public SoftwareNumber()
    {
        counter++;
        serialNumber=counter;
    }

    public static int getTotalNumber()
    {
        return counter;
    }

    public int getSerialNumber()
    {
        return serialNumber;
    }

}
```

在例 3.3 的程序代码中，在 SoftwareNumber 类中同时定义了实例变量成员和类变量成员，也同时定义了实例方法成员和类方法成员。其中示范了在实例方法成员 SoftwareNumber()中访问类变量成员 counter 和实例变量成员 serialNumber 的方式，也示范了在类方法成员 getTotalNumber()中访问类变量成员 counter 的方式，这都是系统允许的访问方式。但是如果尝试在类方法成员 getTotalNumber()中访问实例变量成员 serialNumber，则会发生错误。

编程技巧提示 3.3　谨慎定义静态的类方法

使用 static 关键字声明的方法成员是静态的类方法。在程序设计中，如果不是为了一些特定的用途，尽量不要使用 static 关键字修饰方法成员，因为这样会带来两个不便：静态的类方法在功能更新时不够灵活；静态的类方法在访问变量成员时有权限限制。

编程常见错误提示 3.4　调用构造方法没有使用 new 关键字

初学者易犯的一个错误是在调用构造方法初始化对象实例时，习惯性地使用调用一般方法的方式直接调用构造方法给对象实例赋值，却忘记使用 new 关键字。

编程常见错误提示 3.5　调用方法时没有严格对应实参与形参

很多 Java 语言的初学者都是学过 C 语言的，在编写 Java 语言程序时，对于 Java 语言对方法参数的顺序和类型的要求未能深入理解，依然习惯性地忽略实参与形参的严格对应关系。

3.5　父类、子类和继承

Java 语言支持继承机制，允许在已有的类的基础上派生新的类。在继承关系中，被继承的类称为父类（Superclass）、超类，通过继承得到的类称为子类（Subclass）、派生类。子

类可以继承父类的属性和行为的载体——变量和方法，还可以在子类中添加新的变量成员和方法成员，修改原有的变量成员定义，重写方法成员。Java 语言只支持单一继承，不支持多重继承。在 Java 语言中，所有的类都是通过直接或间接地继承 Object 类而得到的，也就是说，在 Java 语言的“类树”中，Object 类是这棵树的“根”，Object 类是 Java 语言中唯一没有父类的类。

所以，面向对象程序设计的继承机制既可以“遗传”，也可以“变异”。

3.5.1 创建子类

在 Java 语言中，创建子类的过程是在程序的类声明中通过 extends 子句声明父类来实现的，其格式如下：

```
[public][abstract|final] class subclassName extends superclassName
{
    classbody
}
```

其中，subclassName 代表子类的类名；superclassName 代表已经存在的作为父类的一个类的类名；extends 为继承关键字，其后面的部分称为 extends 子句。经过这样的声明，subclassName 类就是 superclassName 类的子类了。子类可以继承父类中访问权限为 protected、public 和 friendly 的成员，但不能继承访问权限为 private 的成员。

Java 语言规定，如果在程序中声明类时没有使用 extends 子句明确指明该类是哪个类的子类，则该类将被认为是 Object 类的子类。这样一来，用户在程序中所定义的类实际上都将加入 Java 语言的“类树”中。

3.5.2 变量成员的隐藏和方法重写

通过继承关系，即使子类中不进行任何关于变量成员和方法成员的定义，其中也会含有变量成员和方法成员，这是因为在父类中已经定义过的变量成员和方法成员只要访问权限不是 private，就会通过继承关系被传递到子类中。子类除了继承父类中的变量成员和方法成员，还可以在自己的类体中定义新的变量成员和方法成员。这样一来，子类中的变量成员和方法成员将包括两部分：一部分是继承得到的；另一部分是新定义的。

在子类中定义新的变量成员有一种特别的方式：变量成员的名称可以和从父类中继承的变量成员的名称相同，但其数据类型可以和原来的数据类型不一致，这称为变量成员的隐藏。在子类中定义新的方法成员的名称、参数列表、返回类型时，也可以和从父类中继承的方法成员的名称、参数列表、返回类型相同，这称为方法重写。重写的方法可以根据需要对父类中原有的方法进行功能改造，即修改其方法体，还可以对其访问权限进行修改。但是重写的方法不能使用比被重写的方法更严格的访问权限，也就是说，新定义的方法的访问权限或者和原来方法的访问权限一样，或者比原来方法的访问权限宽。例如，如果原来的访问权限是 friendly，新的访问权限就必须是 friendly、protected 或 public，不能是 private；如果原来的访问权限是 protected，新的访问权限就必须是 protected 或 public，不能是 friendly 或 private。

在子类中定义与父类中变量同名的变量时，父类中的同名变量将被隐藏。在子类中重写的方法也将把从父类中继承的方法隐藏，这种现象称为方法重写或方法覆盖（Overriding）。方法重写也是 Java 语言多态性的一个体现。如果把子类从父类中继承变量成员和方法成员看作“遗传”，那么在子类中定义新的变量成员和方法成员，修改父类中的变量成员定义和方法重写就可以被看作继承机制中的“变异”。

3.5.3 super

为了准确地分辨父类中原有的成员和子类中修改定义的成员，Java 语言通过 super 关键字来实现对父类中被隐藏成员的访问。super 关键字的使用有 3 种格式：

```
super.variableName                    //访问父类变量
super.methodName([paramList])         //调用父类方法成员
super([paramList])                    //调用父类构造方法
```

这样一来，无论是父类中原有的成员还是子类中修改定义的成员，都可以在程序中被准确地调用，实现了运行时多态。

例 3.4 说明了类继承的一般方式和 super 关键字的使用。

【例 3.4】定义一个“产权”类，要求其中有业主姓名、面积和办理日期。在“产权”类的基础上派生一个“带车库的产权”类，要求增加车库序号信息。

具体的程序如程序清单 3.4 所示。程序的执行结果如图 3.2 所示。

程序清单 3.4

```
//Example 4 of Chapter 3

import javax.swing.JOptionPane;
import java.util.GregorianCalendar;

class PropertyRight
{
    protected String Owner;
    protected double Area;
    protected GregorianCalendar DateofPurchase;

    public PropertyRight(String n,double s,GregorianCalendar d)
    {
        Owner = n;
        Area = s;
        DateofPurchase = d;
    }

    public String putoutDateofPurchase()
    {
        return DateofPurchase.get(GregorianCalendar.YEAR) + "年"
```

```
                + DateofPurchase.get(GregorianCalendar.MONTH) + "月"
                + DateofPurchase.get(GregorianCalendar.DAY_OF_MONTH) + "日";
        }

        public String getDetails()
        {
            return "所有人：" + Owner +"，面积：" + Area + "，办理日期：" +
putoutDateofPurchase();
        }
    }

    class PropertyRightWithGarage extends PropertyRight
    {
        protected String GarageNumber;

        public PropertyRightWithGarage(String n,double s,GregorianCalendar
d,String number)
        {
            super(n,s,d);
            GarageNumber = number;
        }

        public String getDetails()
        {
            return super.getDetails() + "，车库序号：" + GarageNumber;
        }

    }

    public class PropertyRightTest
    {
        public static void main(String args[])
        {
            String output = "";

            PropertyRight a,b,c;
            PropertyRightWithGarage d,e;

            GregorianCalendar date1 = new GregorianCalendar(2001,4,28);
            GregorianCalendar date2 = new GregorianCalendar(2001,2,20);
            GregorianCalendar date3 = new GregorianCalendar(2001,8,18);
            GregorianCalendar date4 = new GregorianCalendar(2011,3,13);
            GregorianCalendar date5 = new GregorianCalendar(2011,5,22);
```

```
        a = new PropertyRight("李嘉庆",200.0,date1);
        b = new PropertyRight("张寿常",165.0,date2);
        c = new PropertyRight("王文昌",200.0,date3);

        d = new PropertyRightWithGarage("边立志",137.0,date4,"8181");
        e = new PropertyRightWithGarage("邵东志",145.0,date5,"4285");

        output += "\n业主一的信息：" + a.getDetails();
        output += "\n业主二的信息：" + b.getDetails();
        output += "\n业主三的信息：" + c.getDetails();
        output += "\n业主四的信息：" + d.getDetails();
        output += "\n业主五的信息：" + e.getDetails();

        JOptionPane.showMessageDialog(null,output);
        System.exit(0);

    }
}
```

图 3.2　程序清单 3.4 的程序的执行结果

在例 3.4 的程序代码中，先定义了 PropertyRight 类，其中包含 3 个保护型变量成员和 2 个方法成员，随后在 PropertyRight 类的基础上派生定义了 PropertyRightWithGarage 类，增加了一个新的保护型变量成员，共含有 4 个变量成员。在 PropertyRightWithGarage 类中重写了 PropertyRight 类的方法成员 getDetails()。在 PropertyRightWithGarage 类的构造方法中，我们看到了 super 关键字的一种使用方法，此时使用的是 PropertyRightWithGarage 类的父类 PropertyRight 的构造方法。在 PropertyRightWithGarage 类的方法成员 getDetails()中，我们看到了 super 关键字的另一种使用方法，此时的 super 关键字代表父类，使用的是父类的方法成员 getDetails()。在这个例子中，还可以看到 Java 语言多态性的一个表现：代码中分别使用 5 个对象实例调用了 getDetails()方法，代码相同，却得到了不同的运行结果。在调用 a、b、c 三个对象实例时，输出了信息窗中呈现的前三行；在调用 d、e 两个对象实例时，输出了信息窗中呈现的后两行。这是因为 a、b、c 是父类的对象实例，调用的是父类的方法成员；而 d、e 是子类的对象实例，调用的是子类的方法成员。在程序中使用了一个 Java 类库中用来处理时间的 GregorianCalendar 类。

3.5.4 final 属性：final 类和 final 方法

具有 final 属性的类是不能被继承的类，称为最终类；具有 final 属性的方法是不能被重写的方法，称为最终方法。使用 final 关键字修饰类和方法，可以分别得到最终类和最终方法。我们已经在前面的类定义和方法定义中看到了 final 关键字的使用方法，这里就不再给出使用格式了。将某一个类定义为最终类，是希望这个类不再被继承；将某一个方法定义为最终方法，是希望在类的继承中不要再改写这个方法。最终类中的所有方法都隐含地具有最终方法的属性，但是最终方法不一定只包含在最终类中，实际上，在很多普通的类中也包含最终方法。

final 属性的实质是不让所描述的语言成分再发生变化，不让它们“进化”。

3.5.5 abstract 属性：abstract 类和 abstract 方法

abstract 属性与 final 属性的性质相反，具有 abstract 属性的类是必须被继承的类，称为抽象类；具有 abstract 属性的方法是必须被重写的方法，称为抽象方法。使用 abstract 关键字修饰类和方法，可以分别得到抽象类和抽象方法。我们也已经在前面的类定义和方法定义中看到了 abstract 关键字的使用方法，这里也不再给出使用格式了。抽象类不能被实例化，只有其子类才能被实例化，所以抽象类只有被继承之后，才能被程序使用。抽象方法是只给出方法声明而没有给出方法体的方法，即抽象方法不提供方法实现，只有在子类中重新定义其方法体，才有实际意义。抽象类中不一定含有抽象方法，但含有抽象方法的类一定是抽象类。因为同一个抽象类可能被不同的新类所继承，同一个抽象方法在被继承之后，也可能被实现为不同内容的方法，所以抽象类和抽象方法也是 Java 语言实现多态的手段之一。

abstract 属性的实质是强制性地让所描述的语言成分发生变化，让它们必须“进化”。

由于 final 属性和 abstract 属性的性质相反，无法同时在一个类或方法上出现，所以 final 关键字和 abstract 关键字不能同时使用在一个类或一个方法上。

3.5.6 类继承机制在程序设计中的作用

类的继承可以让子类直接获取在父类中定义的内容，并直接在子类中使用。在父类中定义的方法就像在子类中编写的一样，使用时没有访问权限等限制，这种使用是几乎无障碍的。从软件工程角度来讲，类继承机制是软件复用技术的一种，由于被复用的代码是对使用方开放的，因此这种复用方式属于“白盒复用”。类继承机制允许在已有程序代码的基础上迅速开展新的程序设计工作，并且提供了对原有程序代码进行修改和补充的方法。

3.6 接口

Java 语言不支持类的多重继承，而是使用接口实现多重继承的功能。在接口中只定义方法，而不给出方法体，所以其中的方法实际上是抽象方法。另外，在接口中只允许定义常量，不允许定义变量。可以说接口是抽象方法与常量的集合，从本质上来讲，接口是一

种特殊的抽象类。接口与接口之间允许多重继承，由于在接口中只定义方法，而不给出方法体，因此接口之间的多重继承只是方法集合的合并，即子接口的方法集合是其所有父接口的方法集合的并集，即使继承了多个相同的方法，也不会引起方法在执行时的混淆。接口需要通过在类中实现来实现其程序功能，并且一个类可以实现多个接口。利用这种接口之间的继承和接口与类之间的实现，Java 语言明显地改进了 C++语言中的多重继承机制，实现了比 C++语言中的多重继承机制更强的功能。类的继承和接口的实现可以同时出现在一个类定义中。

接口也是 Java 语言实现多态的重要手段之一。同一个接口在不同的实现中可以实现不同的功能。

3.6.1 接口的定义

在 Java 语言中，接口是通过 interface 关键字来声明的，基本的接口声明格式如下：

```
[public]interface interfaceName
{
    interfaceBody
}
```

其中，interfaceName 代表接口的名称，通常由用户使用一个符合 Java 语法要求的标识符来作为接口名。一般来说，接口名采用带有-able 或-ible 词尾的英文单词。接口具有 public 访问权限，所以即使在程序代码中没有写出 public，也不影响其访问权限。

3.6.2 接口体的定义

接口体包含常量定义和方法声明两部分。接口体中定义的常量具有 public、static 和 final 属性。接口体中的方法只进行方法声明，无须提供方法体，用分号结尾，具有 public、abstract 属性。

【例 3.5】一个简单的接口的定义。

具体的程序如程序清单 3.5 所示。

程序清单 3.5

```
//Example 5 of Chapter 3

interface Collection
{
    int  MAX_NUM = 100;
    void  add(Object  obj);
    void  delete(Object  obj);
    Object  find(Object  obj);
    int  currentCount( );
}
```

3.6.3 接口的继承

Java 语言允许在已有接口的基础上定义新的接口，这是通过接口的继承机制实现的。Java 语言不允许类之间的多重继承，但允许接口之间的多重继承。

接口之间的继承与类之间的继承一样，也是使用 extends 子句来实现的。被继承的接口称为父接口。在 extends 子句中，可以写出当前接口所要继承的所有父接口，如果父接口多于一个，则使用逗号隔开，具体的格式如下：

```
[public] interface subInterfaceName extends superInterfaceNameList
{
    interfacebody
}
```

子接口可以继承各个父接口的全部成员，而对于相同的成员来说，子接口中只保留一个副本。也可以说，子接口的方法成员的集合是各个父接口的方法成员的集合的并集。

3.6.4 接口的实现

在接口中只声明了方法成员，而没有给出方法体，这样还不能在程序中使用。要想使用接口中声明的方法成员，就必须在实现接口的类中给出方法体，这个过程称为接口的实现。在类的声明中，使用 implements 子句来实现某个或某些接口，其格式如下：

```
[public][abstract|final] class className [extends superclassName]
    implements interfaceNameList
{
    classbody
}
```

在一个类中可以实现多个接口，多个接口的名称可以写在 implements 子句中，并使用逗号隔开。在接口中定义的方法都必须在实现接口的类中给出方法体，只要给出了方法体，接口就实现了。即使没有必要给出方法体的方法，按照语法要求，在形式上也必须给出一个由一对大括号“{”和“}”标出的空方法体。类的继承和接口的实现可以同时出现在一个类定义中，所以可以在一个类定义中同时出现 extends 子句和 implements 子句。extends 子句和 implements 子句之间可以连续依次写，不需要使用逗号或其他符号隔开。

实际上，利用抽象类也可以实现接口的部分功能。二者的差别在于，抽象类的子类除继承抽象类之外，不能再继承其他的类，而实现接口的类可以同时实现多个接口，还可以再继承一个类。显然这种方法能实现更多的内容。

这里不再给出实现接口的例子，在第 5 章介绍事件监听器时，会给出很多实现一个或多个接口的程序。

3.6.5 接口代码的存储

与类一样，一个接口是一个代码存储的基本单元。接口体的源程序代码在存储时，一般与类存储在同一个源程序文件中，接口体的代码与类的代码并列。存储文件的命名规则

依然遵循存储类的代码时的命名规则，接口体的源程序不影响文件命名。

当包含接口体的代码的源程序进行编译时，除了每个类会各自生成一个字节码文件，接口体也会生成一个专有的字节码文件。文件的基本名与接口名相同，并以.class 为扩展名。

3.6.6 Java 8 对接口定义的扩展修订

Java 8 修订了接口的定义，主要的改动有以下两个。

（1）允许在接口中定义非抽象的方法实现，并使用 default 关键字修饰该方法。该方法称为扩展方法，默认扩展方法可以在实现接口的子类上直接使用。例如：

```
interface Shape
{
    double calculate(double a);
    default double perimater(double u)
    {
        ……
    }
}
```

在此接口的基础上定义实现类：

```
public class NewShape
{
    public double calculate(double a)
    {
        ……
    }
}
```

之后，可以在代码中直接使用接口中定义的扩展方法，例如：

```
NewShape ns;
ns.perimater(2.205);
```

（2）允许在接口中定义非抽象的方法实现，并使用 static 关键字修饰该方法，然后在程序中直接使用接口名调用该方法，例如：

```
interface Shape
{
    double calculate(double a);
    public static double perimater(double u)
    {
        ……
    }
}
```

之后，可以在代码中直接使用接口中定义的方法，例如：

```
Shape.perimater(2.205);
```

编程常见错误提示 3.6　在实现接口的方法时未声明 public 访问权限

无论接口的方法声明中是否写了 public 关键字，所有的方法都具有公有属性。在实现接口的类中，如果实现接口中声明的方法成员没有明确使用 public 关键字修饰，则意味着将这个方法的访问权限定义为默认的 friendly 了，访问权限会比原方法更弱。这个错误将导致一个编译错误，从而在编译程序时无法通过。

3.7　多态性的讨论

3.7.1　多态性的概念

多态性是面向对象程序设计中非常有实用价值的技术，也是一项比较复杂的技术。多态性与继承机制中的方法重载、方法重写、接口、抽象类等概念密切相关，利用多态性，可以设计和实现具有良好的可扩展性的系统。

多态性是指在一个继承层次结构或接口实现结构中，一个对象实例在代码叙述形式相同的情形下，可以依据其所指向的对象类型调用不同的执行方法，其后台的执行逻辑就是代码的动态加载。程序在编译时仅对语法进行检验，只有在执行时才会根据对象实例的类型与具体的执行方法实现关联，这种机制称为运行时绑定或动态绑定（Dynamic Binding）。

3.7.2　继承层次结构中对象间的关系

在 Java 语言的语法中，存在着类与类之间的常规继承方式，存在着在某一个类中实现接口的继承方式，还存在着为了使抽象类成为可用类而进行的继承方式。这 3 种继承方式如图 3.3 所示。

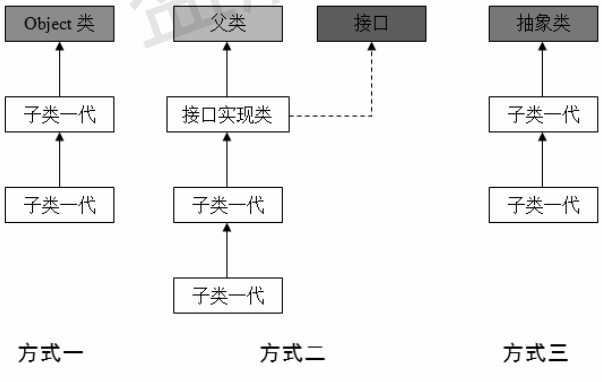

图 3.3　Java 语言中的 3 种继承方式

下面简单比较一下接口与抽象类。在接口中，定义的所有方法都是没有实现的，都具有抽象的属性；而在抽象类中，可能包含部分已经实现了的方法和部分没有实现的方法。如果抽象类中不存在已经实现了的方法供子类继承，则可以使用接口定义替代。如果抽象类的子类依然包含抽象方法且没有实现方法，则该子类仍然需要被声明为抽象类。

在 Java 语言的继承关系中，有一个十分关键且常用的概念：子类对象可以被视为父类的对象。利用这个重要概念，可以在程序中实现很多灵活、丰富的操作。这个概念是 Java

语言多态性的一个重要体现。

从类的继承关系中可以看到，在通常情况下，子类可以继承父类的全部非私有成员，而在子类中还可以定义自己的新成员，所以子类的成员集合比父类的成员集合要大一些。下面两种多态性应用是可以从逻辑上讲清楚的。

（1）允许将父类的对象实例用子类的构造方法实例化，或者将子类的引用赋给父类的对象实例，但是反之不可以。

（2）当访问一个有引用数据类型形参的方法时，可以使用该引用数据类型的子类的对象实例作为实参访问该方法。

第二种应用在本质上与第一种应用是一样的，都是将子类中从父类中继承的成员传递给访问的方法，都是将父类的对象实例中的成员用子类的构造方法初始化。由于子类中的成员数量比父类中的成员数量多，因此子类的构造方法是有能力完成这个工作的。

如果反过来，将子类的对象实例用父类的构造方法实例化，则将产生一个编译错误。

这种现象可以使用类型转换的思想来认识，对 Java 引用数据类型的强制类型转换称为造型（Casting），而造型仅被允许在有继承关系的引用数据类型之间进行。从子类到父类的类型转换可以自动进行，而从父类到子类的类型转换必须通过造型来实现，无继承关系的引用数据类型之间的转换及造型是非法的。为了保证程序中的造型能够正确执行，在造型前可以使用运算符“instanceof”测试一个对象的类型是否有继承关系，只有当测试结果为真时，才进行造型。

例 3.6 对例 3.4 中的程序进行了扩充，定义了一个 Property 接口，并由 PropertyRight 类实现了该接口。在主类中，分别声明了接口的几个对象实例、父类的几个对象实例和子类的几个对象实例，分别验证了几种初始化方式。

【例 3.6】用子类的构造方法初始化父类的对象实例，并将子类的引用赋给父类的对象实例。

具体的程序如程序清单 3.6 所示。程序的执行结果如图 3.4 所示。

程序清单 3.6

```
//Example 6 of Chapter 3

import javax.swing.JOptionPane;
import java.util.GregorianCalendar;

interface Property
{
    public String putoutDateofPurchase();
    public String getDetails();
}

class PropertyRight implements Property
{
    protected String Owner;
    protected double Area;
```

```
        protected GregorianCalendar DateofPurchase;

        public PropertyRight(String n,double s,GregorianCalendar d)
        {
            Owner = n;
            Area = s;
            DateofPurchase = d;
        }

        public String putoutDateofPurchase()
        {
            return DateofPurchase.get(GregorianCalendar.YEAR) + "年"
                + DateofPurchase.get(GregorianCalendar.MONTH) + "月"
                + DateofPurchase.get(GregorianCalendar.DAY_OF_MONTH) + "日";
        }

        public String getDetails()
        {
            return  "所有人："  +  Owner  +"，面积："  +  Area  +  "，办理日期："  +
putoutDateofPurchase();
        }
    }

    class PropertyRightWithGarage extends PropertyRight
    {
        protected String GarageNumber;

        public PropertyRightWithGarage(String n,double s,GregorianCalendar
d,String number)
        {
            super(n,s,d);
            GarageNumber = number;
        }

        public String getDetails()
        {
            return super.getDetails() + "，车库序号：" + GarageNumber;
        }

    }

    public class PropertyRightTest2
    {
        public static void main(String args[])
```

```
{
    String output = "";

    Property p1,p2,p3,p4;
    PropertyRight pr_a,pr_b,pr_c,pr_d;
    PropertyRightWithGarage prg_e,prg_f;

    GregorianCalendar date1 = new GregorianCalendar(2001,4,28);
    GregorianCalendar date2 = new GregorianCalendar(2001,2,20);
    GregorianCalendar date3 = new GregorianCalendar(2001,8,18);
    GregorianCalendar date4 = new GregorianCalendar(2011,3,13);
    GregorianCalendar date5 = new GregorianCalendar(2011,5,22);

    //语法允许的初始化
    p1 = new PropertyRight("测试者",200.0,date1);
    //语法允许的初始化
    p2 = new PropertyRightWithGarage("测试者",145.0,date5,"4285");
    pr_a = new PropertyRight("李嘉庆",200.0,date1);
    pr_b = new PropertyRight("张寿常",165.0,date2);
    //语法允许的初始化
    pr_c = new PropertyRightWithGarage("邵东志",145.0,date5,"4285");

    prg_e = new PropertyRightWithGarage("边立志",137.0,date4,"8181");
    //prg_f = new PropertyRight("王文昌",200.0,date3);语法不允许的初始化

    p3 = pr_a;                  //语法允许的引用赋值
    p4 = prg_e;                 //语法允许的引用赋值
    pr_d = prg_e;               //语法允许的引用赋值
    //prg_f = pr_b;               语法不允许的引用赋值

    output += "\n" + p1.getClass().getName();
    output += "\n" + p1.getDetails();
    output += "\n" + p2.getClass().getName();
    output += "\n" + p2.getDetails();
    output += "\n" + p3.getClass().getName();
    output += "\n" + p3.getDetails();
    output += "\n" + p4.getClass().getName();
    output += "\n" + p4.getDetails();
    output += "\n" + pr_a.getClass().getName();
    output += "\n" + pr_a.getDetails();
    output += "\n" + pr_b.getClass().getName();
    output += "\n" + pr_b.getDetails();
    output += "\n" + pr_c.getClass().getName();
    output += "\n" + pr_c.getDetails();
```

```
            output += "\n" + pr_d.getClass().getName();
            output += "\n" + pr_d.getDetails();
            output += "\n" + prg_e.getClass().getName();
            output += "\n" + prg_e.getDetails();

            JOptionPane.showMessageDialog(null,output);
            System.exit(0);

        }
    }
```

图 3.4　程序清单 3.6 的程序的执行结果

在程序清单 3.6 的程序中，演示了使用接口实现类的构造方法初始化接口的对象实例的操作；演示了使用接口实现类的子类的构造方法初始化接口的对象实例的操作；演示了将接口实现类的引用和接口实现类的子类的引用赋值给接口的对象实例的操作；演示了使用子类的构造方法初始化父类的对象实例的操作；演示了将子类的引用赋值给父类的对象实例的操作。这些操作都是系统允许的。但是使用父类的构造方法初始化子类的对象实例的操作和将父类的引用赋值给子类的对象实例的操作都是系统不允许的，编译时将产生一个编译错误。

无论使用哪一种语法允许的初始化方法进行初始化，在程序执行对象实例时，都将调用对应的方法。然而调用哪个类的方法取决于对象实例被初始化为哪个类的对象实例。那么，对象实例被初始化为哪个类的对象实例，是取决于对象的类型声明呢？还是取决于对象的初始化操作呢？其实这个答案存在于每个对象实例中，可以通过 getClass()方法来获取答案。在程序清单 3.6 的程序中出现了形如 p1.getClass().getName()的语句，该语句就是用来判定这个事情的。getClass()方法是在 Object 类中定义的，其功能是返回一个 Class 类，

该类是对象实例对应的类；getName()方法是在 Class 类中定义的，其功能是返回此 Class 对象所表示的实体的名称。可以从图 3.4 的执行结果中清楚地看到，上述问题的答案是：在声明时可以使用父类的类型声明，甚至可以使用接口的类型声明，对象实例被初始化为哪个类的对象实例取决于对象的初始化操作。各个对象实例的初始化类型与对象实例动态绑定的方法呈现严格、准确的对应关系，这正是 Java 语言多态性程序设计的形式特征。

编程技巧提示 3.4　使用 equals()方法比较两个对象实例

equals()方法是 Object 类定义的方法，可以用来比较两个对象实例。如果两个对象实例是同一个类的实现，并且内容相同，则认为它们是相等的，返回逻辑值 true。equals()方法的使用方式是用一个对象实例调用 equals()方法，将另一个参加比较的对象实例作为 equals()方法的参数放到小括号中。除了可以比较两个对象实例，equals()方法还可以用来比较两个数据，并且这种方法更安全。初学者通常都知道使用运算符“==”比较两个数据是否相等，但是在使用运算符“==”比较两个对象实例时，比较的是两个对象实例的引用是否相同，而不看其内容。

3.8　内部类与嵌套类

3.8.1　内部类与嵌套类的概念

在例 3.4 中我们已经看到，一个程序中可以定义多个类，类与类之间是平等的关系，它们在程序中并列摆放，每个类都是一个代码的模块。除了正常情形，Java 语言还允许创建内部类和嵌套类：如果一个类可以声明在另一个类的类体内，则称为内部类；如果声明在另一个类的类体内的类还具有静态属性，则称为嵌套类。

3.8.2　内部类与嵌套类的两种实现形式

内部类可以以一个完整的类的形式出现在另一个类中，此时包容内部类的类相对而言就称为外部类。内部类的类体与外部类的方法成员并列摆放。内部类与外部类之间存在一个特殊的关系：系统允许内部类直接访问外部类的变量成员和方法成员。另外，如果在内部类中使用了 this 关键字，则此时的 this 指向内部类的当前对象而不是外部类的当前对象。如果要使用外部类的 this 引用，则要在 this 关键字之前加上外部类的类名。

还可以在类的某个方法中声明内部类，这种形式的内部类可以定义类名，也可以不定义类名而形成无名类，即不必为所定义的类给定标识符，称为匿名内部类。在方法中声明的内部类具有 3 个代码层次：外部类、外部类方法、内部类。方法中的内部类可以访问其外部类的对象实例的变量成员和方法成员，也可以访问声明它的方法中的具有 final 属性的局部变量，但是不能访问声明它的方法中的其他局部变量。在匿名内部类中，也可以像普通的类一样实现接口、继承类。

使用包含内部类与嵌套类的类在编译程序时，外部类和内部类将生成各自的字节码文件。外部类的字节码文件的基本名与普通类的生成规则相同，而内部类的字节码文件的基本名由外部类的类名与内部类的类名使用“$”连起来构成，如 OuterClassName$InnerClassName.class。

匿名内部类的字节码文件的基本名类似于 OuterClassName$#.class，其中的“#”为数字，如果程序中有多个匿名内部类，则从 1 开始，每遇到一个匿名内部类，该数字就递增 1。

内部类与嵌套类主要用于事件处理，将实现事件监听器的类写在类的内部，便于访问程序中的某些特定变量。具体实例参见第 5 章的例 5.8 和例 5.9。

3.9 Java 类库中常用类的介绍

本节将简要介绍 Java 类库中几个常用的类。在这些类中，有的已经在前面的例题中出现过，有的则是在设计 Java 语言程序时非常常见和常用的。在学习 Java 语言时，有必要对这些类进行一个初步的了解。关于这些类的具体信息，请读者参阅相关的 Java API 文档。

3.9.1 Object 类

前文已经介绍过，无论是 Java 类库中已经提供的类，还是用户自己定义的类，Java 语言的所有类都可以遵循继承关系，构成一个树状的继承关系图。我们可以将其称为“继承树”或“类树”。在这棵“树”中，处于根的位置的类就是 Object 类，它被存放在 Java 类库的 java.lang 包中。由于 Object 类是所有 Java 类的父类，因此所有 Java 类都将继承该类定义的成员，所有 Java 对象都可以使用该类的成员。Object 类是 Java 语言中唯一一个没有父类的类。

Java 类库中的 java.lang 包是其最基本的一个包，其中定义的类都是关于 Java 语言的最基本的语言内容，在 Java 语言程序中使用该包中的类无须用 import 语句引入，直接使用即可。后面还会介绍该包中的几个类。

程序清单 3.7 中的代码就是 Object 类的定义原型。

程序清单 3.7

```
public class java.lang.Object
{
    public Object();                          //构造方法
    protected Object clone();                 //建立当前对象的副本
    public boolean equals(Object obj);        //比较对象
    protected void finalize();                //释放资源
    public final Class getClass();            //求对象对应的类
    public int hashCode();                    //求哈希码值
    public final void notify();               //唤醒当前线程
    public final void notifyAll();            //唤醒所有线程
    public String toString();                 //返回当前对象的字符串
    public final void wait();                 //使线程等待
    public final void wait(long timeout);
    public final void wait(long timeout, int nanos);
}
```

其中，timeout 为最长等待时间，单位为毫秒；nanos 为附加时间，单位为纳秒，取值范围为 0～999999。

3.9.2 System 类

System 类是 java.lang 包中一个非常常用的实用类，是一个极其重要的类，在前面的章节中已经多次出现过。System 类不能被实例化，包含很多有用的静态字段和静态方法，可以通过类名直接访问。System 类提供了 Java 的标准输入流、标准输出流和标准错误输出流，对外部定义的属性和环境变量的访问，数组元素的快速复制等功能。

System 类中有 3 个静态变量成员：作为标准错误输出流的 PrintStream 类内建对象 err，作为标准输入流的 InputStream 类内建对象 in，作为标准输出流的 PrintStream 类内建对象 out。利用这 3 个内建对象可以实现 Java 语言程序的标准输入和标准输出。System 类中的 exit()方法可以终止当前正在运行的 Java 虚拟机，在前面的例题中已经多次使用过。getProperties()方法、getProperty()方法可以获取当前的系统属性，setProperties()方法、setProperty()方法可以设置当前的系统属性，getSecurityManager()方法可以获取系统安全属性，setSecurityManager()方法可以设置系统安全属性。

3.9.3 Class 类

Class 类是 java.lang 包中一个很特别的类，也是一个极其重要的类。System 类不能被实例化，而 Class 类的实例表示正在运行的 Java 应用程序中的类和接口，也就是说，程序中正在使用的具体的类都是 Class 类的对象实例。Class 类中两个常用的方法成员如下：

```
String  getName()                    //以 String 的形式返回此 Class 对象所表示
                                     //的实体名称
public T newInstance()               //创建此 Class 对象所表示的类的新实例
```

3.9.4 Math 类

在例 3.1 和例 3.2 中，我们已经见到了计算平方根的方法，就是 Math 类的方法成员 sqrt()。Math 类也是定义在 java.lang 包中的，它将基本的数学操作，如绝对值计算、指数计算、对数计算、幂计算、平方根计算和三角函数计算的方法都封装于其中。Math 类中还定义了两个常用的数学常量：圆周率 π 和自然对数的底 e。Math 类中的成员都被声明为静态的，可以通过类名直接访问。利用 Math 类中的方法可以完成程序中常见的数学计算功能。

3.9.5 基本数据类型封装类

在 java.lang 包中，Java 语言为每一种基本数据类型都定义了一个封装类，共有 8 个，分别是 Boolean、Character、Byte、Short、Integer、Long、Float 和 Double 类，并且将逻辑类型、字符类型、4 种整数类型和 2 种浮点类型的基本数据分别进行了封装。除 Boolean 类和 Character 类之外，其他 6 个类都派生自 Number 抽象类。这 8 个类都具有 final 属性，即不可以再被继承。在程序中可以将基本数据类型的数据存储为相应的基本数据类型封装类

的对象实例，这样就可以利用基本数据类型封装类，像操作对象实例一样地操作基本数据类型的数据，特别是可以多态地操作基本数据类型的数据。

在 Number 抽象类中，专门定义了基本数据类型之间的转换方法：

```
byte byteValue()                    //以 byte 形式返回指定的数值
double doubleValue()                //以 double 形式返回指定的数值
float floatValue()                  //以 float 形式返回指定的数值
int intValue()                      //以 int 形式返回指定的数值
long longValue()                    //以 long 形式返回指定的数值
short shortValue()                  //以 short 形式返回指定的数值
```

由于 Byte、Short、Integer、Long、Float 和 Double 这 6 个类都派生自 Number 抽象类，并且各自实现了上述方法，因此只要调用各个类中的相应方法，就可以实现这 6 个数据类型之间的转换。需要留意的是，其中有些情形可能会涉及舍入或取整。

在这 8 个基本数据类型封装类中，定义了解析方法成员，可以把表示基本数据类型变量的字符串解析为相应的数值；还定义了字符串转换方法成员，可以把各种数据类型的数值转换为表示指定数值的字符串。这种解析和转换方式在向程序中的变量进行数值的输入/输出操作中，具有安全性的特点，可以使程序更为稳定、可靠。8 个基本数据类型封装类的方法汇总如下。

Boolean 类：

```
static boolean parseBoolean(String s)  //将字符串参数解析为 boolean 值
String toString()                      //返回表示此 boolean 值的 String 对象
Static String toString(boolean b) //返回一个表示指定 boolean 值的新 String 对象
```

Character 类：

```
String toString()          //返回表示此 char 值的 String 对象
static String toString(char c)     //返回一个表示指定 char 值的新 String 对象
```

Byte 类：

```
static byte parseByte(String s)    //将字符串参数解析为有符号的十进制 byte 值
String toString()                  //返回表示此 byte 值的 String 对象
static String toString(byte b)     //返回一个表示指定 byte 值的新 String 对象
```

Short 类：

```
static short parseShort(String s)  //将字符串参数解析为有符号的十进制 short 值
String toString()                  //返回表示此 short 值的 String 对象
static String toString(short s)    //返回一个表示指定 short 值的新 String 对象
```

Integer 类：

```
static int parseInt(String s)      //将字符串参数解析为有符号的十进制 int 值
String toString()                  //返回表示此 int 值的 String 对象
static String toString(int i)      //返回一个表示指定 int 值的新 String 对象
```

Long 类：

```
static long parseLong(String s)    //将字符串参数解析为有符号的十进制 long 值
String toString()                  //返回表示此 long 值的 String 对象
static String toString(long i)     //返回一个表示指定 long 值的新 String 对象
```

Float 类：

```
static float parseFloat(String s)   //返回一个新的 float 值，该值被初始化为
                                    //用指定 String 表示的值
String toString()                   //返回这个 Float 对象的字符串表示形式
static String toString(float f)     //返回 float 参数的字符串表示形式
```

Double 类：

```
static double parseDouble(String s) //返回一个新的 double 值，该值被初始化为
                                    //用指定 String 表示的值
String toString()                   //返回 Double 对象的字符串表示形式
static String toString(double d)    //返回 double 参数的字符串表示形式
```

3.9.6 数组操作工具类 Arrays

Arrays 类是一个实用类，定义了用来操作数组的各种方法。这些方法都具有静态属性，并且针对不同数据类型的参数给出了不同的重载版本，主要的方法如下：

```
copyOf()                            //将一个数组中的值复制到新数组
copyOfRange()                       //将一个数组中的值按范围复制到新数组
asList()                            //返回一个支持数组的固定大小的列表
binarySearch()                      //用二进制搜索算法在数组中搜索指定值
equals()                            //如果两个数组相等，则返回 true
hashCode()                          //基于指定数组的内容返回哈希码
sort()                              //对数组成员按数字升序进行排序
toString()                          //返回指定数组内容的字符串表示形式
```

3.9.7 String 类和 StringBuffer 类

前文已经介绍过，在 Java 语言中没有字符串数据类型，而是把字符串作为对象来处理。java.lang 包中的 String 类和 StringBuffer 类都可以用来表示一个字符串：String 类用于处理不变的字符串；StringBuffer 类则用于处理可变的字符串。例如，在一个字符串中插入字符或者将两个及两个以上的字符串连接起来，最好使用 StringBuffer 类来处理，一旦处理完成，创建了一个新的字符串，就可以使用 toString()方法将其转换为 String 类对象实例。另外，在 java.util 包中还有一个与字符串处理有关的 StringTokenizer 类。该类允许应用程序将字符串分解为语言符号，可以用来处理自然语言，这里就不详细介绍了。

可以采用以下方式创建一个字符串：

```
String(byte[] bytes)
String(byte[] bytes,int offset,int length)
String(byte[] ascii,int hibyte,int offset,int count)
String(char[] value)
String(char[] value,int offset,int count)
String(String original)
String(StringBuffer buffer)
```

在 String 类中，以下几个方法是比较常用的：

```
charAt()                    //返回指定索引处的 char 值
compareTo()                 //按字典顺序比较两个字符串
equals()                    //比较此字符串与指定的对象
indexOf()                   //返回指定字符在此字符串中第一次出现处的索引
length()                    //返回此字符串的长度
```

在 StringBuffer 类中，以下几个方法是比较常用的：

```
append()                    //将参数的字符串表示形式追加到此序列
delete()                    //移除此序列的子字符串中的字符
insert()                    //将参数的字符串表示形式插入此序列中
setCharAt()                 //将给定索引处的字符设置为给定参数
```

3.9.8 Calendar 类和 GregorianCalendar 类

早期的 Java 版本定义了一个 Date 类，该类是一个描述日期和时间的常用类，它把日期和时间解释为年、月、日、小时、分钟和秒值，还允许格式化和分析日期字符串。为了处理时间功能的国际化，从 Java 1.1 版开始，Date 类中的一些方法逐渐被 Calendar 类和 GregorianCalendar 类中的方法取代，目前 Date 类已经被废弃。这两个类都被存放在 java.util 包中。

Calendar 类是处理日期和时间的类，它是一个抽象类，为特定瞬间与一组日历字段之间的转换提供了一些方法，并为操作日历字段提供了一些方法。GregorianCalendar 类是 Calendar 类的直接子类，也是它的实现类，提供了世界上大多数国家使用的标准日历系统。Calendar 类中定义的静态字段对年、月、日、小时、分钟和秒值，以及月份、星期等有详尽的描述，其中的方法在经由 GregorianCalendar 类实现之后，可以用来处理与日期和时间有关的工作。在程序中，如果要生成与日期和时间有关的对象实例，则必须使用 GregorianCalendar 类生成。

3.10 Java Application 程序的完整结构

前文对 Java 语言中的类的概念进行了介绍，本节将对把类作为组织程序代码的基本单位的作用进行介绍，对 Java Application 程序的基本结构进行总结，同时对 Java Application 程序的代码管理进行总结。

完整的 Java Application 程序的基本结构是：一个程序可以被分成若干个文件，其中可以有一个主类，并且最多只能有一个主类；一个文件中可以含有若干个类和接口，每个类中包含若干个变量成员和方法成员，每个方法中包含若干个变量声明和若干条执行语句。在同一个文件中，类和接口出现的先后顺序对程序的运行没有影响；在同一个类中，变量成员和方法成员出现的先后顺序对程序的运行没有影响。但是按照书写习惯，通常把接口写到类的前面，把变量成员写到方法成员的前面。

1. 完整的 Java 语言程序文件的格式

```
package packageName;                    //指定文件中定义的类所在的包，0 个或 1 个
import packageName.(className|*);       //指定引入的类，0 个、1 个或多个
```

```
public classDefinition                    //主类定义，0 个或 1 个
interfaceDefinition and classDefinition //接口和类定义，0 个、1 个或多个
```

2．完整的类定义格式

```
[public][abstract|final] class className [extends superclassName]
    [implements interfaceNameList]
{
  [public|protected|private][static][final][transient][volatile]type
    variableName;              //变量成员定义，0 个、1 个或多个
    [public|protected|private][static][final|abstract][native][synchronized]
    returnType methodName([paramList])[throws exceptionList]
  {
    statements
  }                            //方法成员定义，0 个、1 个或多个
}
```

有一个特别的类，称为程序的主类，需要在类声明的前面用 public 关键字修饰。对于 Java Application 程序而言，主类中的主方法是程序执行的入口点和出口点。

3．完整的接口定义格式

```
[public]interface interfaceName[extends superInterfaceList]
{
    type constantName = Value;          //常量成员定义，0 个、1 个或多个
    returnType methodName([paramList]);  //方法成员声明，0 个、1 个或多个
}
```

4．3 个特别的类方法成员

1）main()方法

main()方法，即主方法。主方法只能在 Java Application 程序的主类中定义。主方法的名称、参数列表、访问权限、静态属性、返回类型是固定的，在编写程序时都不能改变。其声明格式如下：

```
public static void main(String args[])
{
    methodbody
}
```

主方法是在 Java Application 程序执行时第一个被访问的方法，也是最后一个被退出的方法。主方法是公有方法、静态方法、无返回值的方法。

2）构造方法

前文已经介绍过，构造方法的方法名就是类名，它没有返回类型，通常是 public 访问权限的。构造方法要求的固定声明格式如下：

```
className([paramList])
{
    methodbody
}
```

3）finalize()方法

finalize()方法的声明格式如下：

```
protected void finalize() throws throwable
{
    methodbody
}
```

finalize()方法是在 Object 类中定义的，这个方法在继承过程中会被其子类所继承，因此所有 Java 类中都含有这个方法。在通常情况下，finalize()方法不需要在类中重新定义。

5．Java 文件的存储

在存储 Java 源程序文件时，如果文件中有主类，则文件的基本名必须和主类名一致，否则没有限制，文件的扩展名为.java。Java 源程序文件在经过编译后，得到的是 Java 字节码文件。在存储字节码文件时，其基本名与源程序文件名一致，但文件的扩展名为.class。如果文件中含有多个类和接口，则每个类和接口都各自生成一个专有的字节码文件，并且文件的基本名是类名或接口名。

由于 Java 语言规定没有指明父类的类是 Object 类的子类，因此用户通过写程序来定义类的过程也可以被看作定义 Object 类的子类的过程。用户可以直接使用现有类中的方法成员来完成程序的功能，从而留给用户自己定义代码的工作就比较少了，这是面向对象程序设计所带来的好处之一。在进行 Java 语言面向对象程序设计时，提倡使用已有的方法，这样设计出来的程序可能会比完全由程序员自己编写代码所设计的程序要可靠。

综上所述，我们已经对 Java 语言中面向对象程序设计的思想和面向对象程序设计实现的载体——类进行了介绍，并且对使用类的方式管理程序代码模块的功能进行了介绍。至此，我们已经将 Java 语言面向对象程序设计的语法规则介绍完了。根据这些语法规则，相信读者已经可以设计出完整的 Java 语言程序了。

作为本章的结束，例 3.7 给出了一个具有比较完整的程序功能的典型案例。

【例 3.7】多态性应用案例。

某超市中有以下几大类商品：包装物，每位顾客限购一件，金额固定为包装物的单价；按单价计算金额的一般商品，金额为单价乘以数量；优惠商品，按单价计算金额的一般商品，但是在超过一定数量之后，超出部分按 9 折计价；需带包装购买的商品，包装物 1 件，商品数量不限，金额为包装物金额与商品金额之和。该超市希望编写一个 Java 语言程序，用于计算顾客购买的每一种商品所应付的金额。

具体的程序如程序清单 3.8 所示。程序的执行结果如图 3.5 所示。

程序清单 3.8

```
//Example 7 of Chapter 3

import javax.swing.JOptionPane;

// 定义 Merchandise 抽象类

abstract class Merchandise
```

```
{
    private String Name;
    private String SerialNumber;

    public Merchandise( String name, String number )
    {
        Name = name;
        SerialNumber = number;
    }

    public void setName( String name )
    {
        Name = name;
    }

    public String getName()
    {
        return Name;
    }

    public void setSerialNumber( String number )
    {
        SerialNumber = number;
    }

    public String getSerialNumber()
    {
        return SerialNumber;
    }

    public String toString()
    {
        return getName() + ",SerialNumber: " + getSerialNumber();
    }

    public abstract double paying();

}// 结束定义 Merchandise 抽象类

// 定义 Wrappage 类

class Wrappage extends Merchandise
{
    private double wrappagePrice;
```

```
        public Wrappage( String name, String number, double nowwrappageprice )
        {
            super( name, number );
            setWrappagePrice( nowwrappageprice );
        }

        public void setWrappagePrice( double nowwrappageprice )
        {
            wrappagePrice = nowwrappageprice < 0.0 ? 0.0 : nowwrappageprice;
        }

        public double getWrappagePrice()
        {
            return wrappagePrice;
        }

        // 重写父类的方法
        public String toString()
        {
            return "包装物: " + super.toString();
        }

        // 实现父类的抽象方法
        public double paying()
        {
            return getWrappagePrice();
        }
    } // 结束定义 Wrappage 类

    // 定义 GeneralMerchandise 类

    class GeneralMerchandise extends Merchandise
    {
        private double price;
        private double count;

        public GeneralMerchandise( String name, String number, double
nowprice, double nowcount )
        {
            super( name, number );
            setPrice( nowprice );
            setCount( nowcount );
        }
```

```
    public void setPrice( double nowprice )
    {
        price = nowprice < 0.0 ? 0.0 : nowprice;
    }

    public double getPrice()
    {
        return price;
    }

    public void setCount( double nowcount )
    {
        count = nowcount < 0.0 ? 0.0 : nowcount;
    }

    public double getCount()
    {
        return count;
    }

    // 重写父类的方法
    public String toString()
    {
        return "一般商品: " + super.toString();
    }

    // 实现父类的抽象方法
    public double paying()
    {
        return getPrice() * getCount();
    }
} // 结束定义 GeneralMerchandise 类

// 定义 RebateMerchandise 类

class RebateMerchandise extends Merchandise
{
    private double price;
    private double count;
    private int rebateCount;

    public RebateMerchandise( String name, String number, double nowprice,
        double nowcount, int nowrebatecount )
```

```
{
    super( name, number );
    setPrice( nowprice );
    setCount( nowcount );
    setRebateCount( nowrebatecount );
}

public void setPrice( double nowprice )
{
    price = nowprice < 0.0 ? 0.0 : nowprice;
}

public double getPrice()
{
    return price;
}

public void setCount( double nowcount )
{
    count = nowcount < 0.0 ? 0.0 : nowcount;
}

public double getCount()
{
    return count;
}

public void setRebateCount( int nowrebatecount )
{
    rebateCount = nowrebatecount < 0 ? 0 : nowrebatecount;
}

public int getRebateCount()
{
    return rebateCount;
}

// 重写父类的方法
public String toString()
{
    return "打折商品: " + super.toString();
}

// 实现父类的抽象方法
```

```
    public double paying()
    {
        if ( count <= rebateCount )
            return price * count;
        else
            return price * rebateCount + price * ( count - rebateCount ) * 0.9;
    }
}// 结束定义 RebateMerchandise 类

// 定义 WrappageMerchandise 类

class WrappageMerchandise extends GeneralMerchandise
{
    private double wrappagePrice;

    public WrappageMerchandise( String name, String number, double nowprice,
            double nowcount, double nowwrappageprice )
    {
        super( name, number, nowprice, nowcount );
        setWrappagePrice( nowwrappageprice );
    }

    public void setWrappagePrice( double nowwrappageprice )
    {
        wrappagePrice = nowwrappageprice < 0.0 ? 0.0 : nowwrappageprice;
    }

    public double getWrappagePrice()
    {
        return wrappagePrice;
    }

    // 重写父类的方法
    public String toString()
    {
        return "带包装的商品: " + super.getName() + " " + ",SerialNumber: "
               + super.getSerialNumber();
    }

    // 实现父类的抽象方法
    public double paying()
    {
    return getPrice() * getCount() + getWrappagePrice();
    }
```

```
}// 结束定义 WrappageMerchandise 类

// 定义主类

public class PayingCalculation
{
    public static void main( String[] args )
    {
        // 生成 Merchandise 数组
        Merchandise merchandise[] = new Merchandise[4];

        // 用子类的构造方法实例化
        merchandise[ 0 ] = new Wrappage( "包装袋", "100011231", 1.00 );
        merchandise[ 1 ] = new GeneralMerchandise( "非常可乐", "205433501",
3.50, 12 );
        merchandise[ 2 ] = new WrappageMerchandise( "大米", "502000125",
1.50, 20, 1.00 );
        merchandise[ 3 ] = new RebateMerchandise( "苹果", "320171005",
4.80, 5.5, 5 );

        String output = "";

        // 循环语句中出现运行时多态应用
        for ( int i = 0; i < merchandise.length; i++ )
        {
            output += "\n";
            output += merchandise[ i ].toString();

            // 确定哪个元素是 WrappageMerchandise 类的对象实例
            if ( merchandise[ i ] instanceof WrappageMerchandise )
            {
                // 强制类型转换后赋值给新的对象实例
                WrappageMerchandise currentM = ( WrappageMerchandise )
merchandise[ i ];
                output += "\n 包装袋的价格是：" + currentM.getWrappagePrice();
            } // 结束 if 语句

            output += "\n 需要支付：" + merchandise[ i ].paying() + "\n";

        } // 结束 for 语句

        JOptionPane.showMessageDialog( null, output );
        System.exit( 0 );
```

```
    }// 结束定义主方法

}// 结束定义主类
```

图 3.5　程序清单 3.8 的程序的执行结果

在上面的案例中，总共定义了 6 个类，首先定义了一个 Merchandise 抽象类，作为程序中所需要的类的共同父类，其中含有一个抽象方法。然后在 Merchandise 抽象类的基础上，通过继承定义了 Wrappage、GeneralMerchandise、RebateMerchandise、WrappageMerchandise 等几个实用类，这几个类的继承层次关系如图 3.6 所示。

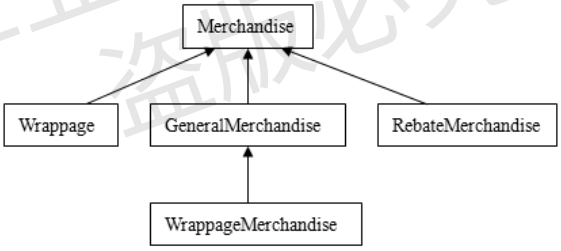

图 3.6　程序清单 3.8 的程序中的几个类的继承层次关系

这几个类中所定义的类的全部功能都通过主类 PayingCalculation 得到了使用，并通过窗口实现了输出。在案例中使用了本章介绍过的继承和多态的多种知识，其中通过抽象类定义公有属性，并在此基础上派生所需要的实用类的设计方法，这在 Java 面向对象程序设计中是比较常用的设计方法。对于类的每一个私有型变量成员，都设计一个设置方法和一个获取方法，这也是一种规范化的 Java 语言程序设计方法。至于在案例中如何使用多态和使用多少种多态的方法，有兴趣的读者可以仔细阅读程序并详细总结一下。程序清单 3.8 中使用了很多由英文单词构成的标识符，建议读者在编写自己的程序时，也尽量使用接近自然语言的字符串作为程序中的标识符，这样做可以实现一种“望词见义”的效果，有利于理解程序的内容和算法。

编程技巧提示 3.5　尽量把程序的业务内容封装在类中

在编写程序时，应该避免把主类和主方法的内容写得过多，较好的方式是把程序的业务内容封装到专门的类中，并且这种封装越是细分，写出的程序就越具有可重用性。而在主类中，只需生成封装类的对象实例，然后调用适当的方法即可完成程序功能。

本章知识点

★　面向对象程序设计思想是结构化程序设计思想的发展和延伸，是目前软件工程领域中的重要设计思想。

★　面向对象程序设计主要体现在对象的封装、类的继承和程序设计的多态上。

★　面向对象程序设计的 3 个主要好处是项目的分解、代码的重用和避免程序的重复开发。

★　Java 语言是一种完全的面向对象的程序设计语言，其中完全实现了对象的封装、类的继承和程序设计的多态。

★　Java 语言对代码采用二级模块管理：一级模块是类；二级模块是方法。

★　类在 Java 语言中的作用有两个：一个作用是作为面向对象程序设计的主体概念；另一个作用是作为组织程序代码的基本单位，即程序代码的一级模块。

★　对象是描述事物的数据与处理数据的方法的集合。面向对象程序设计实现了对象的封装，实现了模块化和信息隐藏。

★　类是统一定义的对象，对象是类的具体实现。

★　对象之间采用消息进行交互。消息包括消息的接收者、接收对象应采用的应对方法和执行方法所需要的参数 3 个方面的内容。

★　继承是面向对象程序设计方法中的重要内容，通过类的继承可以实现程序代码的重复利用。

★　Java 语言只支持单一继承，不支持多重继承，其多重继承的功能是通过接口之间的多重继承和在类中实现接口实现的。

★　多态是面向对象程序设计方法中的重要内容。通过多态，程序可以实现丰富多彩的设计内容。

★　Java 语言中有多种实现多态的手段：方法重载和方法重写、用子类的构造方法初始化父类的对象实例、抽象类和抽象方法、接口等。

★　类包含类头和类体，类体中包含变量成员和方法成员，变量成员是类的属性的载体，方法成员是类的行为的载体。

★　Java 语言中的方法体是由一段程序代码构成的，是 Java 语言程序的实质性执行成分。

★　如果一个方法是有返回类型的，就必须能够向外传递一个与返回类型相同的值，如果返回类型为空，就不能向外传递值。

★　声明的作用域就是程序中可以通过正常的方式引用所声明的程序实体的代码范围。Java 语言的作用域分为类级、方法级、语句块级、语句级。

★　用来初始化类对象的方法称为构造方法。构造方法是一种特殊的方法成员，其名

称与类名相同，没有返回类型，只能使用 new 关键字调用。

★ 任何类中都有 finalize()方法，其作用是释放系统资源，一般在程序中无须重写这个方法。

★ 在一个类中可以定义多个相同名称、相同访问权限、相同返回类型，但参数列表不同的方法，称为方法重载。

★ 对象是类的实例化。在程序代码中，对象实例的使用包括生成、使用和清除 3 个阶段；对象实例的生成包括声明、实例化和初始化；对象实例的使用是为了访问其成员；对象实例的清除是释放其占用的资源，这是由自动垃圾收集机制完成的。

★ 包的作用有两个：管理类和管理类名。

★ package 语句让程序将生成的类存储到指定的包中；import 语句让包中现有的类被程序所用。

★ 在类的内部对成员的访问没有什么限制；在类的外部对成员的访问会受到访问权限的限制。

★ 从形式上总结，对成员的访问有 3 种形式：直接使用成员名访问、使用对象实例名访问、使用类名访问。

★ 在方法定义时所使用的参数称为形参；在方法访问时所使用的参数称为实参。实参的数量和类型要与形参一致，Java 语言可以通过参数提升对与形参类型不一致的实参进行有限的类型转换，转换的规则称为提升规则。

★ this 是指代当前对象实例的关键字，可以用其访问当前对象实例的成员。

★ Java 语言中规定了 4 种类的成员的访问权限，分别为 private、protected、public、friendly。

★ 具有 static 属性的成员称为类成员。类成员的访问方式比较简单，但是类成员的访问功能比较弱。

★ 继承机制允许在已有类的基础上派生新的类，被继承的类称为父类，通过继承而得到的类称为子类。

★ 在子类中可以继承父类的属性和行为的载体——变量和方法，还可以在子类中添加新的变量成员和方法成员，修改原有的变量成员定义，重写方法成员。

★ Java 语言只支持单一继承，不支持多重继承。

★ 创建子类的过程是通过 extends 子句实现的，其中声明了当前类的父类。如果在声明类时没有使用 extends 子句，则其父类为 Object 类。

★ 子类中可以定义与父类中原有的变量成员同名的变量成员，也可以重写父类中原有的方法成员，此时父类中原有的变量成员和方法成员将被隐藏。

★ 使用 super 关键字可以访问父类中被隐藏的变量成员和方法成员，也可以访问父类的构造方法。

★ 在 Java 语言的继承关系中，有一个十分关键且常用的概念：子类对象可以被视为父类的对象。

★ 具有 final 属性的类是不能被继承的类，称为最终类；具有 final 属性的方法是不能被重写的方法，称为最终方法。

★ 具有 abstract 属性的类是必须被继承的类，称为抽象类；具有 abstract 属性的方法

是必须被重写的方法，称为抽象方法。

★ Object 类是 Java 语言的所有其他类的父类，被存放在 java.lang 包中。

★ java.lang 包是定义 Java 语言基本内容的类的共用包。在使用这个包中的类时，无须使用 import 语句引入。

★ 接口是实现 Java 语言多重继承功能的重要内容，具有抽象类的属性。接口也是 Java 语言实现多态的重要手段。

★ 在多个接口的基础上可以定义新的接口，即接口允许多重继承。

★ 在实现接口的类中必须给出接口中声明的方法的方法体。实现接口需要使用 implements 子句，并在其中写出被实现的接口的名称。一个类可以实现多个接口。

★ Java 语言允许将一个类声明在另一个类的类体中，称为内部类。如果这个内部类还具有静态属性，则称为嵌套类。

★ 内部类可以是定义在外部类中的一个完整类，也可以是声明在外部类的方法中的类。方法中的类通常不必定义标识符，称为匿名内部类。

★ 内部类在编译时会独立生成一个字节码文件，其基本名为外部类名加“$”再加内部类名。匿名内部类在编译时生成的字节码文件的基本名为外部类名加“$”再加一个从 1 开始的数字。

★ 在 java.lang 包中，Java 语言为每一种基本数据类型都定义了一个封装类，分别是 Boolean、Character、Byte、Short、Integer、Long、Float 和 Double 类。利用这些类可以像处理一般对象实例一样地处理基本类型数据。

★ System 类是 java.lang 包中的实用类，可以用来进行标准输入/输出，获取和设置系统属性等操作。

★ String 类和 StringBuffer 类都可以用来进行字符串的有关处理工作。

★ Math 类中封装了进行数学计算的方法，可以通过类名直接访问其中的方法并进行相关的数学计算。

★ Date 类、Calendar 类和 DateFormat 类可以用来处理日期和时间。

★ 完整的 Java Application 程序的基本结构是：一个程序可以被分成若干个文件，其中可以有一个主类，并且最多只能有一个主类，一个文件中可以含有若干个类和接口，每个类中包含若干个变量成员和方法成员，每个方法中包含若干个变量和若干条执行语句。

★ 主方法只能在主类中定义。主方法的名称、参数列表、访问权限、静态属性、返回类型是固定的，在编写程序时都不能改变。主方法是 Java Application 程序执行时的入口点，也是程序执行结束时的出口点。

习题 3

3.1 总结一下 Java 语言中的面向对象程序设计思想都体现在哪些地方，以及封闭、继承和多态的思想都是如何实现的。

3.2 什么是类？什么是对象？

3.3 类的变量成员和方法成员各自体现了类的什么内容？

3.4 在 Java 语言的继承机制中，子类除继承父类中的内容之外，还可以通过哪些方式

改变和扩充其从父类中继承的内容？

3.5　什么是包？包的作用是什么？

3.6　什么是 final 类和 final 方法？什么是 abstract 类和 abstract 方法？

3.7　关键字 this 和 super 的含义是什么？使用方式是什么样的？

3.8　什么是方法重载？什么是方法重写？简要说明二者的区别。

3.9　Java 语言中的 4 种访问权限是如何界定的？

3.10　什么是实例变量成员？什么是类变量成员？

3.11　什么是实例方法成员？什么是类方法成员？

3.12　什么是接口？接口的多重继承机制是如何实现的？

3.13　接口中定义的方法是怎样实现的？

3.14　什么是内部类？内部类是如何定义的？

3.15　回顾一下例 3.7，在程序清单 3.8 的程序代码中，GeneralMerchandise 类在实现父类 Merchandise 的方法成员 paying()时是采用 getPrice()方法和 getCount()方法获取私有成员 price 和 count 的值的，而在 RebateMerchandise 类中，在实现 paying()方法时是直接采用变量名获取 price 和 count 的值的，请问这是为什么？这两种方式有什么差别？

3.16　同样是在程序清单 3.8 的程序代码中，在主类的循环语句中有一个表达式：

```
merchandise[i] instanceof WrappageMerchandise
```

请问按照程序中的实际情况，当程序执行到这个循环语句时，上面的这个表达式可能为真吗？当 i 的值为多少时这个表达式为真？为什么？

3.17　编写程序，将程序清单 3.3 中定义的类生成 5 个对象实例，看一看在第 5 个对象实例生成之后静态变量 counter 的值是多少，并解释为什么。

3.18　接口继承和实现练习：编写 3 个接口，其中各含有 3 个方法声明，并且在 3 个接口所声明的方法中有名称相同的方法出现，在这 3 个接口的基础上派生一个新的接口，在其中增加 3 个常量成员，并编写一个主类来实现这个子接口。

3.19　访问权限练习：编写一个类，其中含有 4 个访问权限分别为 private、protected、public、friendly 的变量成员和方法成员，再编写一个主类，其中生成前面的类的一个对象实例，然后使用这个对象实例分别访问 4 个变量成员和方法成员，并编译程序，看看将有哪些现象出现。

3.20　方法重载练习：编写一个类，其中含有至少两个具有相同名称但形参列表不同的方法，再编写一个主类，其中生成前面的类的一个对象实例，然后使用这个对象实例分别访问重载的方法成员。

3.21　编写一个类，其中声明了实例变量成员，也声明了类变量成员。编写两个类方法成员，分别用于访问实例变量成员和类变量成员；编写两个实例方法成员，分别用于访问实例变量成员和类变量成员。再编写一个主类，其中生成前面的类的一个对象实例，然后调用 4 个方法成员并编译程序，看看将有哪些现象出现。

3.22　模仿程序清单 3.1 中的程序代码，定义“正方形”类，定义描述正方形边长和面积的变量成员 edgelength 和 area，定义相应的方法成员 setEdgelength()、getEdgelength()、setArea()、getArea()，重新定义方法成员 toString()，并编写一个主类，在其中生成一个正方形的对象实例，然后通过这个对象实例完成对上述方法成员的调用操作。

3.23　在前一个题目的基础上定义“正方形”类的子类“正方体”，改变变量成员 area 的含义为正方体的表面积，增加描述体积的变量成员 volume，修改方法成员 setArea()、getArea()的定义为对表面积的操作，增加方法成员 setVolume()、getVolume()以完成对体积的操作，重写方法成员 toString()，并编写一个主类，在其中生成一个正方体的对象实例，然后通过这个对象实例完成对上述方法成员的调用操作。

3.24　在程序清单 3.1 中的程序代码中增加一个判定三角形是否为直角三角形的方法，并将其命名为 isRightAngled()；增加一个判定三角形是否为等腰三角形的方法，并将其命名为 isEquilateral()；增加一个判定三角形是否为等边三角形的方法，并将其命名为 isIsosceles()。

3.25　定义一个“圆”类，其中含有一个浮点类型的变量作为其半径，并定义获取直径、获取周长、获取面积等方法。在“圆”类的基础上定义“圆锥”“圆柱”两个子类，增加一个浮点类型的变量作为其高度，并定义各自的获取体积的方法。

3.26　将 3.25 题中的程序用一个主类来实现对象实例，并输出有关的计算结果。

实验 3

S3.1　编写一个有关车主信息管理的程序。首先定义一个“汽车”类，包含以下信息：汽车品牌、汽车类型（轿车、多用途汽车、载重车、客车等）、汽车型号。这几项都要求是字符串对象实例。然后在“汽车”类的基础上，定义一个“车主”类，包含以下信息：姓名、性别、出生日期、汽车、车辆颜色、底盘号码、购买日期、购买价格、联系电话。其中，姓名、性别、车辆颜色、底盘号码、联系电话等要求是字符串对象实例；汽车要求是上述定义的“汽车”类的对象实例；购买价格要求是浮点型数据；出生日期、购买日期要求是日期和时间类型，使用 GregorianCalendar 类的对象实例。最后编写一个主类，用于测试这两个类的性能。

S3.2　根据图 3.7 所定义的类的继承关系编写 Java 语言程序以实现 3 个类并测试运行，要求为每个类的每个成员都定义 setter 和 getter 方法。其中，“合同”类是父类，“采购合同”类和“销售合同”类是“合同”类派生的子类。

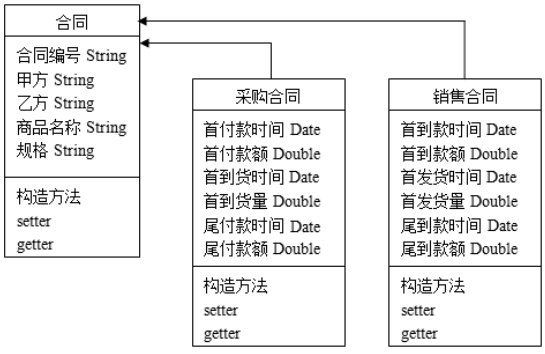

图 3.7　类的继承关系