第3章 面向对象程序设计基础

本章导读

面向对象程序设计(Object Oriented Programming,OOP)是一种程序开发模式,与面向过程的程序设计相比较,更接近于人的自然思维。面向对象程序设计模拟客观世界中事物的特点和规律,采用抽象的方法将软件系统中待解决的问题抽象为人们熟知的对象模型,然后通过模型创建实体(对象),再通过对这些对象状态和行为的描述,来实现软件系统的整体功能。采用面向对象程序设计方法开发的软件系统具有更稳定的软件结构,易于维护、修改、扩充,并可较好地支持代码复用。本章将详细介绍面向对象程序设计基本思想和方法。

本章要点

- 面向对象程序设计的特点。
- 面向对象程序设计基本概念及使用方法,包括类、对象、属性、方法等。
- 继承的作用、原则、使用方法。
- 多态的含义及实现方法。
- 包的概念、作用及使用方法。
- 接口的含义、作用及使用方法。
- Java 常用类的作用及使用方法。
- Java API 使用方法。

3.1 面向对象程序设计特点

面向对象程序设计方法以客观世界存在的事物为基础,采用人类自然思维模式,对问题(软件系统)进行抽象,从而构造软件系统。例如,要设计计算器,首先要确定计算器的设计图,包括计算器面板样式,也就是按键数量、各个按键的位置、大小、颜色等,另外还包括计算器功能设计。然后,根据设计图制作出实际的计算器。现在要使用面向对象程序设计语言 Java 实现计算器软件,将计算器抽象为一个类,声明如下:

```
class Calculator
{
    private JFrame f;                          //计算器面板窗口
    private double result;                     //存储计算器运算结果的变量
    private JButton[ ] btn;                    //按键
    private HelpWindow helpWindow;             //帮助窗口
    private MemWindow memWindow;               //记忆窗口
```

```
    private JPanel[] p;                          //子面板
    private JTextField displayText;              //显示运算结果的文本框
…
}
```

类相当于计算器的设计图，有了设计图后，根据设计图制作实际的计算器，抽象出计算器类后，使用 new 关键字创建计算器对象，创建计算器对象时，如果采用相同的属性值，则产生相同的计算器；如果使用不同的属性值，例如不同的按键大小、颜色、排列等，则可以制作出不同的计算器。

面向对象程序设计的主要特点包括封装、继承和多态。

1. 封装

封装是指将数据和对数据的操作处理放在一起，是面向对象程序设计的核心思想之一。其优势在于：一是信息的隐藏。将数据和对数据的操作封装在一起，也就是将描述对象的属性和行为封装在一起，封装之后外部只能通过对象提供的接口访问对象内部数据，从而保护内部数据不受外部干扰，达到隐藏对象实现细节的目的；二是提高程序的可维护性，当一个对象的源代码独立编写，其内部结构发生改变时，只要对象的访问接口不变，其余代码就可以不变，从而提高程序的可维护性。

2. 继承

继承是根据已有类(对象模型)创建新类的一种方法，也是面向对象程序设计体现代码复用的关键特性。继承时已有类称为父类(或超类)，新创建的类称为子类(或派生类)，子类可以继承父类的属性和行为，子类也可以在继承后添加自己特有的属性和行为。继承避免了具有相同属性和行为的类的重复定义。

3. 多态性

多态性是指使用相同接口完成不同操作或操作不同类型数据，它是面向对象程序设计的另一个重要特征。Java 有两种多态，一是重载，即同一接口可接收不同信息完成不同操作；二是重写和覆盖，也就是继承于同一父类的不同子类，改写了父类中的同一属性或行为，从而体现出各自的特征或表现出不同的行为。

注意：本章前面的叙述中提到的接口和方法为广义的概念，与后面将要讲述的 Java 中的接口和方法略有不同。

3.2　类与对象

类和对象是面向对象程序设计最重要的两个概念。类是一种数据类型，每个类都有一个特定的数据结构，用于描述一类事物(对象)共有的属性和行为。面向对象程序是由类构成的，面向对象编程的实质就是类设计的过程。对象是类的一个特定实例，类是创建对象的模型。对象的属性通过类中的成员变量来描述，对象的行为通过类中的成员方法来描述，通过成员方法对变量的操作实现软件系统功能，接下来详细介绍类和对象。

3.2.1 类的结构

在Java语言中,类由声明和类体两部分构成。

1. 类的声明部分

类声明的格式如下:

```
[修饰符] class <类名>[extends 父类名][implements 接口列表]
```

参数说明如下:

(1) 修饰符用于说明类的访问权限或类型。可选参数为public、abstract、final。其中,public描述类的访问权限,为公共类,具有公共访问权限,没有public修饰符则仅允许在同一个包中的类访问;abstract表示该类为抽象类;final表示该类为最终类,不允许被继承。

(2) 类名:指定类的名称,其命名必须满足Java标识符命名规范。

(3) extends父类名:表示该类继承于extends后面指定的父类,Java为单继承,因此extends后面有且仅有一个父类名。

(4) implements接口列表:表明该类实现了哪些接口,一个类可以实现多个接口。

例如:

```
public class Hello                          //声明一个公共类 Hello
class MyFrame extends JFrame                //声明类 MyFrame 继承 JFrame 类
class MyThread implements Runnable          //声明类 MyThread 实现 Runnable 接口
```

2. 类体

在类声明之后,一对大括号括起来的部分称为类体。类体中通常包括两部分内容,一是变量的声明;二是方法的定义。其中,类中声明的变量又称为成员变量或域变量,用于描述该类对象的属性;类中定义的方法称为成员方法,成员方法用于描述该类对象的行为。

例如,例1-2声明了Count类,类体中声明了a和b两个成员变量,并定义了additive和subtraction等方法。其中,变量用于存储参与运算的值,方法则用于完成运算。

另外,还可以在类中定义内部类,详见3.5节。

3.2.2 成员变量

类中声明的变量称为成员变量,成员变量声明的一般语法格式如下:

```
[修饰符] <变量类型> <变量名>;
```

参数说明如下:

(1) 修饰符用于定义成员变量的访问权限或类型。可选参数有:public、protected、private、static、final。其中,public、protected和private用于描述成员变量的访问权限。public表示公共成员变量,protected表示受保护的成员变量,private表示私有的成员变量。static表示该成员变量为静态变量,也称为类变量,否则称为实例变量。静态变量可以通过类名访问,也可以通过对象名访问,实例变量只能通过对象名访问;final用于声明常量。访

问权限修饰符与 static 和 final 可联合使用。

(2) 变量类型可以为 Java 任意数据类型。

(3) 变量名需满足 Java 标识符命名规范。

例 3-1　使用成员变量。

源文件为 Sample 3_1.java,代码如下。

```
public class Sample 3_1
{
    public int age;                                    //声明了公共整型变量
    protected String name;                             //声明了受保护的字符串变量
    float weight;                                      //声明了友好的浮点类型变量
    private boolean marry;                             //声明了私有的布尔类型变量
    public static String address;                      //声明了公共的静态字符串变量
    protected final char SEX = 'w';                    //声明了受保护的字符常量
    private static final String STATE = "中国";        //声明了私有的静态字符串常量
}
```

成员变量通常声明在类的开始部分,也可以在类的其余部分声明,成员变量在整个类内有效,并且与声明位置无关。

3.2.3　成员方法

成员方法是类体的另一个重要组成部分。简单地讲,方法是能够完成一定功能的程序片段。Java 中成员方法包括方法声明和方法体两部分。

1. 方法的声明

成员方法声明的一般格式如下:

```
[修饰符]<方法返回值类型><方法名>([参数列表])
```

参数说明如下:

(1) 修饰符用于定义成员方法的访问权限和类型,可选参数有:public、protected、private、static、final、abstract。其中,public、protected、private 这三个参数用于定义访问权限。static 表示成员方法为静态方法,即类方法,与静态成员变量类似,静态方法也可以通过类名引用;没有 static 修饰的成员方法称为实例方法,实例方法只能用对象名引用;final 用于声明最终方法,最终方法可以被子类继承但不允许被覆盖;abstract 用于声明抽象方法,抽象方法没有方法体,必须在子类中实现后才可以使用。

(2) 方法返回值类型可以是任何 Java 数据类型,用于指定方法返回值类型。如果方法没有返回值,则采用 void 进行说明。

(3) 方法名需满足 Java 标识符命名规范。

(4) 参数列表用于指明该方法所需参数。可以有多个参数,参数中间用逗号分隔,可以使用任何 Java 数据类型。方法也可以没有参数,没有参数时,“()”不可以省略。

例如:

```
public static int count()                    //声明了公共的类方法,返回值为整型,没有参数
```

```
void maxValue((int numberOne,int numberTwo) //声明了友好的无返回值的方法,有两个整型参数
```

2. 方法体

方法声明之后,用一对大括号“{}”括起来的部分是方法体,用于完成指定的功能。方法体中包括若干合法的 Java 语句,可以是常量或变量声明的语句,也可以是对常量或变量进行操作处理的语句。在方法体内既可以对方法体内声明的常量或变量进行处理,也可以对方法体外在类中声明的成员变量进行处理。另外,在方法体内还可以声明内部类。

需要读者注意的问题是,在方法体内声明的变量称为局部变量,其声明和使用与成员变量有一定的区别。

1) 局部变量的声明

```
[final]<变量类型><变量名>;
```

其中,final 用于声明常量。

例如:

```
final float PI = 3.14f;                    //声明一个局部常量 PI
int partOne;                               //声明一个局部变量 partOne
```

2) 使用范围

局部变量可以声明在方法体内的任何位置,其有效范围与声明位置相关,从声明之后到该方法结束范围内有效。如果局部变量声明在方法体内的复合语句中,则其有效范围为该复合语句块;如果局部变量声明在循环语句内,则其有效范围为该循环语句(包括循环体)。除了在方法内声明的变量,方法的参数也是局部变量,其有效范围为整个方法。

由于方法既可以操作成员变量也可以操作局部变量,在同一范围内,当局部变量与成员变量同名时,成员变量被隐藏(屏蔽),如果想操作成员变量必须使用 this 关键字。

例 3-2 使用局部变量。

源文件为 Sample 3_2.java,代码如下。

```
public class Sample3_2
{
    public int vOne = 100;          //vOne 成员变量在整个类有效
    public void test (int pOne)     //参数 pOne 在整个方法内有效
   {
        int vOne  = 200;            //局部变量 vOne 在整个方法内有效,成员变量 vOne 被隐藏
        pOne = 300;                 //参数赋值
        System.out.println("vOne = " + vOne);  //vOne = 200
        for(int i = 0;i < 10;i++)               //复合语句块中定义的局部变量 i
       {
            System.out.print(i + " ");
        }
        //System.out.print(i); 局部变量 i 仅在 for 语句块内有效
        if (pOne > 0)
       {
          //int vOne = 500; vOne 已经在方法 test 中声明,此处不可再声明
           System.out.println("vOne = " + vOne);    //vOne = 200,成员变量被屏蔽
```

```
        }
    }
}
```

3.2.4　构造方法

构造方法是一种特殊的方法，通常用于成员变量初始化，在创建对象时被调用执行。构造方法的名称必须与其所在类的类名称相同，没有返回值，也不使用 void 关键字。在一个类中允许定义多个构造方法，多个构造方法名称相同，但参数个数或类型不同，如果没有定义构造方法，系统会创建一个默认的构造方法，默认的构造方法没有参数，也不包括任何语句。如果用户定义了构造方法，则系统不再创建默认的构造方法。另外，构造方法前不可以使用 static、final、abstract 修饰符。

例 3-3　使用构造方法。

源文件为 Sample3_3.java，代码如下。

```
public class Sample3_3
{
    float price;
    int numOne;
    public Sample3_3()                  //无参数且不进行任何操作的构造方法，通常在继承中使用
   {
   }
    public Sample3_3(float p, int n)  //有参数的构造方法
   {
         price = p;
         numOne = n;
    }
}
class Area                              //无构造方法
{
     int countArea (int length )
   {
          return length * length;
   }
}
```

Sample3_3 类中声明了两个构造方法，调用时通过参数确定调用哪个构造方法。

3.2.5　对象

类声明之后，就可以使用类声明对象，然后调用构造方法创建对象，也称实例化对象，由类创建的对象也称实例。对象创建后才能使用，通过对象完成各种功能。对象的声明、创建、使用方法如下。

1. 声明对象

声明对象的一般格式如下：

```
类名  对象名;
```

其中,类名必须是已经定义的类,可以是系统标准 API 中的类,也可以是用户定义的类;对象名要符合标识符命名规则。

例如:

```
Sample3_3 sOne;
Area  aOne;
```

2. 创建对象

类是一种引用类型,采用类声明的对象则为引用类型变量,在 Java 语言中采用 new 关键字创建(实例化)对象。

创建对象的一般格式如下:

```
对象名 = new 构造方法名([参数列表]);
```

实例化对象时需调用类中编写的构造方法,完成初始化工作。如果类中没有构造方法,则调用默认的无参构造方法完成实例化对象工作。

例 3-4 创建对象。

源文件为 Sample3_4.java,代码如下。

```
class SampleOne
{
    int var;
    SampleOne(int pVar)
   {
        var = pVar;
   }
    void printSample()
   {
        System.out.println("测试成员方法的调用情况!");
    }
}
class SampleTwo
{
     int var = 5;
}
public class Sample3_4
{
     public static void main(String args[])
   {
          SampleOne  sOne = new SampleOne(5);              //声明并实例化对象
          SampleOne  sTwo = new SampleOne(10);
          SampleTwo  s1 = new SampleTwo();
          SampleTwo  s2 = new SampleTwo();
          System.out.println("sOne.var = " + sOne.var);    //输出成员变量 var 的值
          sOne.printSample();                              //调用成员方法
          System.out.println("sTwo.var = " + sTwo.var);
```

```
        System.out.println("s1.var = " + s1.var);
        System.out.println("s2.var = " + s2.var);
    }
}
```

运行结果如图 3.1 所示。

图 3.1　例 3-4 运行结果

3. 使用对象

对象创建后，可以使用“.”运算符调用成员变量和成员方法，例如：sOne. var、sOne. printSample()。通过调用成员变量获得对象属性特征，调用成员方法完成一定功能。

3.2.6　static 成员

在类中使用关键字 static 声明的变量称为类变量或者静态变量，使用 static 关键字声明的方法称为类方法或静态方法；没有 static 声明的变量则称为实例变量，没有 static 声明的方法则称为实例方法。static 成员属于类，而非 static 成员属于对象。Java 语言中，static 成员与非 static 成员的使用方法不同。

1. 类变量与实例变量

类变量为该类所有对象共享，而每个对象独自拥有各自的实例变量。类变量可以通过类名访问，也可以通过对象名访问；而实例变量只能通过对象名引用；类变量在加载字节码文件时分配内存，而实例变量在创建对象时分配内存。

例 3-5　使用类变量与实例变量。

源文件为 Sample3_5.java，代码如下。

```
public class Sample3_5
{
   static final int vOne = 11;                          //类常量
   static int vTwo;                                     //类变量
   final int vThree;                                    //实例常量
   int vFour;                                           //实例变量
   Sample3_5(int parameter)
  {
     vThree = parameter;
```

```
  }
   public static void main (String args[])
  {
     Sample3_5 sOne = new Sample3_5(31);
     Sample3_5 sTwo = new Sample3_5(32);
     System.out.println(Sample3_5.vOne);                    //类常量可以直接用类名访问
     Sample3_5.vTwo = 21;                                   //类变量可以直接用类名访问
     sOne.vFour = 41;                                       //实例变量只能用对象名访问
     sTwo.vFour = 42;
     System.out.println(sOne.vOne + " " + sOne.vTwo + " " + sOne.vThree + " " + sOne.vFour);
     System.out.println(sTwo.vOne + " " + sTwo.vTwo + " " + sTwo.vThree + " " + sTwo.vFour);
     sOne.vTwo = 201;                                       //可以用对象名访问类变量
     sTwo.vTwo = 202;
     System.out.println(Sample3_5.vTwo + " " + sOne.vTwo + " " + sTwo.vTwo);
     //三个值相同,类变量为该对象共享
     }
}
```

运行结果如图 3.2 所示。

图 3.2 例 3-5 运行结果

类中为各对象共享的常量或变量可声明为 static 类型,能够节省存储空间,并且可以在不创建对象的情况下访问。

2. 类方法与实例方法

类方法可以用类名和对象名两种方法调用,而实例方法只能用对象名调用。实例方法可以调用实例方法和变量,也可以调用类方法和类变量;而类方法中不可以使用非 static 成员。

例 3-6 使用类方法与实例方法。

源文件为 Sample3_6.java,代码如下。

```
public class Sample3_6
{
     static int a;
     int b;
     static int cuntone()                          //类方法
    {
          return a * a;                            //类方法只能访问类变量
```

```
    }
     int countTwo()                          //实例方法
   {
          b = a * a;                         //可以访问实例变量,也可以访问类变量
          return  b;
    }
}
```

3. static 代码块

在类中除声明成员变量、成员方法、内部类，还可以声明静态代码块，静态代码块在加载字节码文件时执行一次，静态代码块可以有多个，按声明先后执行。

例如，加载 JDBC 驱动程序代码，可形成一个静态块在加载字节码文件时执行。

```
public class JDBC
{
    …
    static
    {
        try
        {
            class.forName("com.in.crosoft.jdbc.sqlserver.SQLServer");
        }
        catch(classNotFoundException e)
        {
             e.printstackTrace();
        }
    }
    …
}
```

另外，static 还可以用于声明内部类。

3.2.7　this 关键字

this 关键字表示当前创建的对象，可以在实例方法中使用，但不能在类方法中使用。this 关键字可用于访问被同名局部变量隐藏的成员变量，也可以在构造方法中通过 this 关键字调用其余的构造方法，此时 this 关键字必须是构造方法的第一条语句。

例 3-7　使用 this 关键字。

源文件为 Sample3_7.java，代码如下。

```
public class Sample3_7
{
    int a;
    int b;
    Sample3_7 (int pa)
   {
        this();
        a = pa;
```

```
    }
     Sample3_7()
    {
          b = 20;
    }
     void test()
    {
        int a = 1,b = 2;
        System.out.println("test a = " + a + ",test b = " + b);            //调用局部变量
        System.out.println("this.a = " + this.a + ",this.b = " + this.b);
                                                                          //调用当前对象的成员变量
    }
     public static void main(String args[])
    {
        Sample3_7 sOne = new Sample3_7(10);
        System.out.println("sOne.a = " + sOne.a + ",sOne.b = " + sOne.b); //调用 sOne 成员变量
        sOne.test();                                                      //调用 sOne 成员方法
    }
}
```

运行结果如图 3.3 所示。

图 3.3 例 3-7 运行结果

3.2.8 参数传递

调用成员方法时,可以传递参数,参数可以是基本数据类型也可以是引用数据类型。与 C 略有不同,Java 参数传递是值传递。

例 3-8 参数传递。

源文件为 Sample3_8.java,代码如下。

```
public class Sample3_8
{
    int a;
    Sample3_8(int pa)
    {
        a = pa;
    }
}
class TestClass
{
```

```
    public static void testOne (int tOne)
    {
        System.out.println("形参 tOne 被修改前的值: " + tOne);
        tOne = 1;
        System.out.println("形参 tOne 被修改后的值: " + tOne);
    }
    public static void testTwo(Sample3_8 sOne)
    {
        System.out.println("形参 sOne.a 被修改前的值: " +  sOne.a);
        sOne.a = 2;
        System.out.println("形参 sOne.a 被修改后的值: " +  sOne.a);
    }
    public static void testThree (Sample3_8 sTwo)
    {
        System.out.println("形参 sTwo 被修改前的值: " +  sTwo.a);
        sTwo = new Sample3_8(3);
        System.out.println("形参 sTwo 被修改后的值: " +  sTwo.a);
    }
    public static void main(String args[])
    {
        int t1 = 10;
        Sample3_8 s1 = new Sample3_8(100);
        Sample3_8 s2 = new Sample3_8(200);
        System.out.println("实参 t1 被调用前的值: " + t1);
        TestClass .testOne(t1);
        System.out.println("实参 t1 被调用后的值: " + t1);
        System.out.println("实参 s1.a 被调用前的值: " + s1.a);
        TestClass .testTwo(s1);
        System.out.println("实参 s1.a 被调用后的值: " + s1.a);
        System.out.println("实参 s2.a 被调用前的值: " + s2.a);
        TestClass .testThree(s2);
        System.out.println("实参 s2.a 被调用后的值: " + s2.a);
    }
}
```

运行结果如图 3.4 所示。

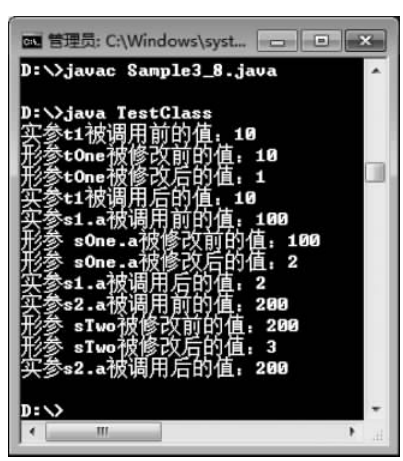

图 3.4　例 3-8 运行结果

参数传递内存模型示例如图 3.5 所示。

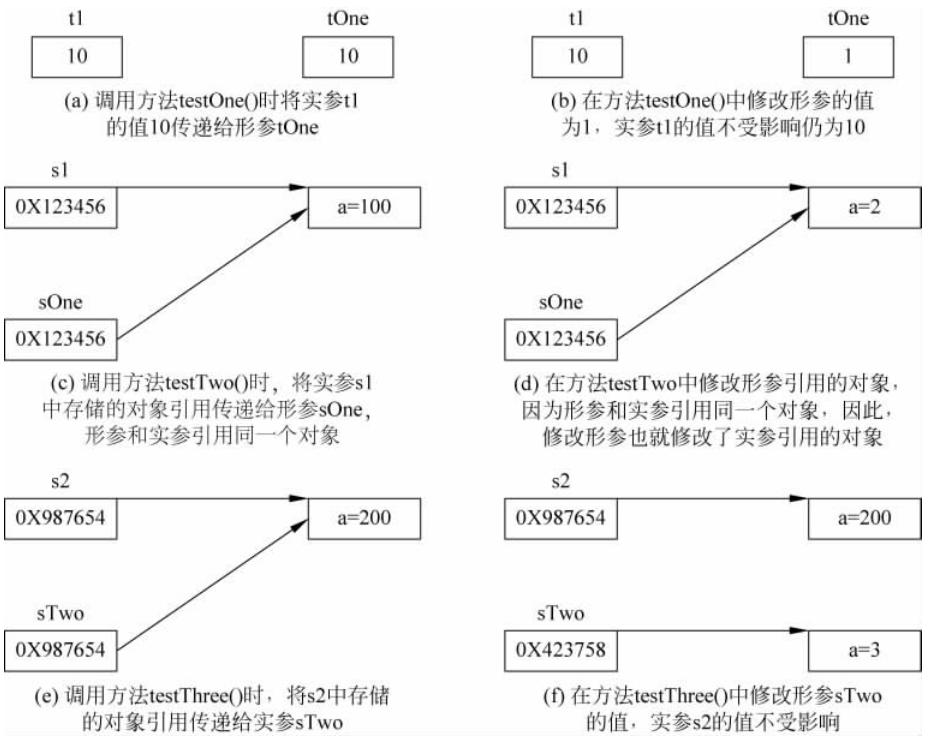

图 3.5　参数传递内存模型变化示例

3.2.9　重载

重载是指在同一个类中出现各个方法名相同，但参数不同的方法。参数不同可以是参数个数不同，也可以是参数类型不同。调用重载方法时，根据传递参数的不同决定具体执行哪个方法。

例 3-9　重载。

源文件为 Sample3_9.java，代码如下。

```
public class Sample3_9
{
    int count(String str)
    {
        return str.length();
    }
    int count(int a)
    {
        return a * a;
    }
    int count (int a,int b)
    {
        return a * b;
```

```
    }
}
class Test
{
    public static void main (String args[])
    {
        Sample3_9 s = new Sample3_9();
        String str = "abcdef";
        System.out.println("调用 s.count(str)的输出结果: " + s.count(str));
        System.out.println("调用 s.count(3,5)的输出结果: " + s.count(3,5));
        System.out.println("调用 s.count(5)的输出结果: " + s.count(5));
    }
}
```

运行结果如图 3.6 所示。

图 3.6　例 3-9 运行结果

注意：重载方法的返回值类型和参数的名称不参与比较，也就是说方法的返回值类型及参数名称是否相同不能作为判断方法是否重载的条件。

3.3　包

包是 Java 中提供的一种管理类的机制。使用包可以将相关的类放在同一包中，方便使用；另外，使用包还可以解决类名冲突问题，可以将相同名称的类分别存放在不同的包中；最后，使用包可以控制类及成员的访问范围。下面介绍包的使用方法。

3.3.1　包的声明

使用 package 语句声明包。package 语句必须作为 Java 源文件的第一条语句出现，用于指定该源文件中类和接口所在的包。如果 Java 源文件中没有 package 语句，则该源文件中的类和接口默认存储在无名包中，这样的类和接口如果在同一文件夹下，则认为在同一包中。

1. package 语句格式

package 语句的一般格式如下：

```
package 包名;
```

例如：

```
package sun;
```

包的功能与操作系统中文件夹(目录)的功能类似。包可以嵌套声明，包中还可以内嵌包，包与其内嵌的包用“.”运算符连接，例如：

```
package sun.com.sample;
```

表示 sun 包中内嵌 com 包，com 包中又内嵌着 sample 包。

包的结构与目录结构相对应，上述声明表示当前操作系统中存在这样的目录结构：\sun\com\sample\。

如果一个 Java 源文件中声明了包，则该源文件中类和接口的字节码文件中一定保存在指定的包中，否则 JVM 无法加载这样的字节码文件。Java 源文件的存储位置没有特殊要求。

2. Java 程序的编译与运行

1) 编译

如果包路径已经创建，Java 源文件保存在包语句所指定的路径中，编译时可以在源文件所在的路径中编译，也可以在包的上层路径中进行编译。如果 Java 源文件保存在包的上层路径中，或包路径未创建，则需在源文件所在路径中进行编译，且需正确指定字节码文件存储的路径。

例 3-10 使用包。

源文件为 Sample3_10.java，代码如下。

```
package sun.com;
public class Sample3_10
{
    public static void main (String args[])
    {
        System.out.println("测试包语句的使用情况");
    }
}
```

如果保存 Sample3_10.java 在 d:\sun.com 中，则可以在 D:\sun\com 路经下对类进行编译，命令如下：

```
D:\sun\com > javac Sample3_10.java
```

也可以在包上层路径中进行编译，即在 D:\>路径中完成编译，命令如下：

```
D:\> javac sum\com\ Sample3_10.java
```

此时必须指明源文件所在的路径。

如果 Sample3_10.java 保存在 d:\中，则需要在 d:\中进行编译。命令如下：

```
D:\> javac -d .  Sample3_10.java
```

其中,“-d .”参数表示创建包路径并将字节码文件生成在包路径中;“.”表示当前路径;“-d .”表示在当前路径下创建包路径,也可以换成实际存在的路径名。例如:

```
D:\> javac -d d:\ Sample3_10.java
```

表示在 d:\下创建包路径。

注意:上述命令不可以写成如下形式:

```
D:\> javac Sample3_10.java
```

这样写编译不会提示错误,但字节码文件生成在当前路径(d:\)下,运行时 JVM 无法加载 Java 字节码文件。

2) 运行

运行有包语句的 Java 程序,需在包的上层路径中进行,例如,运行 sun.com 包中的 Sample3_10,使用如下命令:

```
D:\> java sun.com. Sample3_10
```

另外,Java 规定不允许使用 java 作为包名。

3.3.2　类的引入

采用 Java 进行程序开发时,可以使用已经存在的类,这样可以避免一切从头做起。Java 规定,在当前类中可以直接使用与该类在同一包中的类。如果不在同一包中,则需通过 import 语句将要使用的类引入当前源文件才可以使用。import 语句需要放在 Java 源文件 package 语句之后,类声明之前的位置;并且一个源文件可以使用多个 import 语句。import 语句不仅可以引入系统提供的类,也可以引用用户自己编写的类。如果希望引入某个包中全部的类,可以使用“*”。

例如:

```
import java.net.*;                    //引入 java.net 包中的全部类
import java.util.Scanner;             //引入 java.util 包中的 Scanner 类
import sun.com.Sample;                //引入用户自己定义的类 Sample
```

另外,如果需要使用的类在当前源文件中使用次数很少,也可以使用长名引用包中的类,而不用 import 语句引入类。

例如:

```
Sun.com.Sample s = new sun.com.Sample();
```

3.4　继承

继承是采用已有类创建新类的一种方法,是实现代码复用的关键技术。由已有类创建新类,新类称为已有类的子类或派生类,而已有类称为父类、基类或超类。子类通过继承具

备了父类的特征，并可以对其进行修改和扩充，使子类具备独有的属性和行为。Java 仅支持单重继承，在类的声明语句中，使用 extends 关键字声明继承关系。

例如：

```
class Rose extends Flower
{
…
}
```

其中，Rose 为 Flower 的子类，Flower 是 Rose 的父类。

Java 语言规定，所有类都有父类，如果声明类时没有使用 extends 关键字指明继承关系，则系统默认该类的父类为 Object 类。

例如：

```
public class Sample
{
…
}
```

等价于

```
public class Sample extends Object
{
…
}
```

3.4.1 继承的原则

继承时，原则上除父类的构造方法外，子类可以继承父类的全部成员，但由于父类成员方法和成员变量访问权限的限制，造成父类的部分成员在子类中无法继承。具体继承原则如下。

(1) 子类可以继承父类中声明为 public 和 protected 的成员。

(2) 如果子类和父类在同一包中，则子类可以继承父类中由默认修饰符修饰的成员。

(3) 子类不可以继承父类中声明为 private 的成员。

(4) 子类不仅继承父类中直接声明的成员，父类继承于上一级父类的成员也可以被子类继承。

注意：在父类中子类不能继承的成员变量和方法，子类可以通过父类中公有的成员方法或内部类间接调用。

例 3-11 继承的原则。

源文件为 Sample3_11.java、Sample3_11_1.java 和 Sample3_11_2.java。

Sample3_11.java 代码如下。

```
package sun.com;
public class Sample3_11
{
    public int t1 = 1;
```

```
    protected int t2 = 2;
    int t3 = 3;
    private  int t4 = 4;
    public void pOne()
    {
        System.out.println("father public method");
    }
    public void pTwo()
    {
        System.out.println("t4 = " + t4);
        pThree();
    }
    private void pThree()
    {
        System.out.println("father private method");
    }
}
```

Sample3_11_1.java 代码如下。

```
package sun.com;
public class Sample3_11_1 extends Sample3_11
{
    public static void main(String args[])
    {
        Sample3_11_1 s = new Sample3_11_1();
        System.out.println(s.t1);     //父类 public 成员
        System.out.println(s.t2);     //父类 protected 成员
        System.out.println(s.t3);
/*由默认修饰符修饰的成员,如果子类和父类在同一包中,则子类可继承,如果不在同一包中,则不
可以继承。*/
        //System.out.println(s.t4);   private 成员不可以继承
        s.pOne();
        //s.pThree();          private 成员不可以继承
        s.pTwo();             //可以通过父类的公有成员方法间接调用父类私有成员变量和方法
    }
}
```

运行结果如图 3.7 所示。

图 3.7　例 3-11 运行结果 1

Sample3_11_2.java 代码如下。

```
public class Sample3_11_2 extends Father
{
    int p3 = 1;
    public static void main(String args[])
    {
        Sample3_11_2 son = new Sample3_11_2();
        System.out.println(son.p1);
        System.out.println(son.p2);
        System.out.println(son.p3);
    }
}
class Grandpa
{
    public int p1 = 10;
}
class Father extends Grandpa
{
    public int p2 = 20;
}
```

运行结果如图 3.8 所示。

图 3.8 例 3-11 运行结果 2

父类继承于上一级父类的成员也可以被子类继承。

3.4.2 隐藏与覆盖

子类不仅可以继承父类的属性和行为,子类还可以进行改写和扩充,使其具有独自的特征和行为。

例 3-12 隐藏与覆盖。

源文件为 Sample3_12.java,代码如下。

```
package sun;
import sun.com.Sample3_11;    //不在同一个包中,Sample3_11 必须为 public 才可以使用
public class Sample3_12 extends Sample3_11
{
    public int t1 = 10;
    int s1 = 100;
```

```
    public void pOne()          //重写时访问权限不可降级
    {
        System.out.println("Son change father public method.");
    }
    public void sOne()
    {
        System.out.println("son add method");
    }
    public static void main(String args[])
    {
        Sample3_12 s = new Sample3_12();
        System.out.println(s.t1 + " " + s.s1);
        s.pOne();
        s.sOne();
    }
}
```

运行结果如图 3.9 所示。

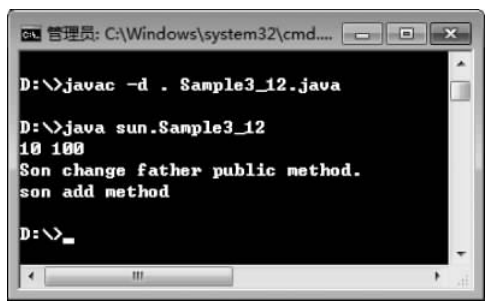

图 3.9　例 3-12 运行结果

具体原则如下。

(1) 如果子类声明了与父类成员变量同名的成员变量,则父类的成员变量被隐藏。子类成员方法中使用的是子类中声明的成员变量而不是继承于父类的同名成员变量。

(2) 如果子类声明了与父类成员方法同名的成员方法(返回值、方法名称、参数个数及类型都相同),则父类的成员方法被覆盖;子类中调用的是在子类中声明的方法而非继承于父类的同名成员方法,覆盖通常称为重写,重写父类的方法时不允许降低父类方法的访问权限。

(3) 父类中的 final 方法不允许被重写(覆盖)。

(4) 父类中的 static(静态)方法只能被子类中同名的 static 方法覆盖;非 static 方法也只能被非 static 方法覆盖。

注意:

① 如果子类中声明的方法与父类方法比较,返回值类型不同,方法名及参数相同,则子类方法无法覆盖父类的同名方法,编译时会出错。

父类中方法:

```
public int testOne(int a)
{…}
```

子类中方法：

```
public string testOne(int a)
{…}
```

上述方法不能覆盖，编译时出错。

② 如果子类中声明的方法与父类方法比较，名称相同，参数不同（参数个数或类型不同），则子类方法重载了父类中的同名方法。

例如：

父类中方法：

```
public int testTwo(int a)
{…}
```

子类中的方法：

```
public int testTwo(int a,int b)
{…}
```

子类重载父类方法 testTwo。

(5) 子类只能隐藏或覆盖能够从父类继承的成员。

3.4.3 super 关键字

super 关键字主要有两种用途，一是操作被隐藏或覆盖的成员；二是调用父类的构造方法。具体使用方法如下。

1. 使用 super 关键字操作被隐藏的成员变量或被覆盖的成员方法

语法格式为：

```
Super.成员变量名
Super.成员方法名([参数列表])
```

例 3-13 使用 super 关键字操作被隐藏的成员变量或被覆盖的成员方法。

源文件为 Sample3_13.java，代码如下。

```
public class Sample3_13
{
    int p = 10;
    void pMethod()
   {
        System.out.println("father");
   }
}
class Test extends Sample3_13
{
    int p = 20;
    void pMethod()
   {
```

```
        System.out.println("son");
    }
    void print()
    {
        super.pMethod();                    //父类
        System.out.println(super.p);        //父类
    }
    public static void main(String args[])
    {
        Test t = new Test();
        t.pMethod();                        //子类
        System.out.println(t.p);            //子类
        t.print();
    }
}
```

运行结果如图 3.10 示。

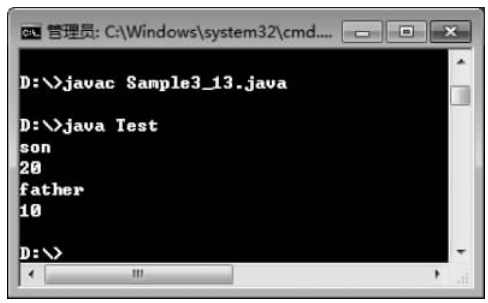

图 3.10　例 3-13 运行结果

2. 使用 super 关键字调用父类构造方法

父类的构造方法不能被子类继承，可以在子类构造方法中使用 super 关键字调用父类的构造方法；如果没有在子类构造方法中使用 super 关键字显式调用父类构造方法，系统将在子类构造方法被执行时默认首先调用父类的无参构造方法，此时如果父类中没有无参构造方法则运行时会出错。

使用 super 关键字调用父类构造方法的语法格式如下：

```
super([参数列表]);
```

例 3-14　使用 super 关键字调用父类构造方法。

源文件为 Sample3_14.java，代码如下。

```
class Sample3_14
{
    Sample3_14()
    {
        System.out.println("父类无参构造方法");
    }
    Sample3_14(int fp)
```

```
    {
        System.out.println("父类有参构造方法 fp = " + fp);
    }
}
class Test extends Sample3_14
{
    int son;
    Test()
    {
        super(5);
    }
    Test (int sp)
    {
        son = sp;
    }
    public static void main (String args[])
    {
        Test t1 = new Test();
        System.out.println(t1.son);
        Test t2 = new Test(10);
        System.out.println(t2.son);
    }
}
```

运行结果如图 3.11 所示。

图 3.11　例 3-14 运行结果

3.4.4　final 关键字

final 关键字可用于修饰变量、方法或类。

1. final 类

final 类不允许继承，也就是 final 类没有子类。

例如：

```
final class sample                          //final 类没有子类。
{
…
}
```

2. final 方法

final 方法不允许被重写。也就是说子类不能覆盖父类中可继承的 final 方法，但可以使用。

3. 常量

由 final 修饰的变量称为常量。常量值在程序运行过程中不允许改变。常量通常在声明时赋值，但 Java 也允许对常量延迟赋值，但不同常量规定不同，其中 static 类型常量只能在声明时赋值或在静态块中赋值；非 static 类型的常量可以在声明时赋值也可以在构造方法中赋值；局部常量则在方法中赋值。在类中声明的常量与成员变量相似，static 类型常量为类所有，非 static 类型常量为对象所有，每个对象的常量值可以不同。

例 3-15　使用 final 关键字。

源文件为 Sample3_15.java，代码如下。

```
public class Sample3_15
{
    static final int a;
    final int b;
    final String s = "常量";
    static
    {
        a = 10;
    }
    Sample3_15(int p)
    {
        b = p;
    }
    static int returna()
    {
        return a;
    }
     int returnb()
     {
        return  b;
     }
     String returns()
     {
        return  s;
     }
  public static void main(String args[])
  {
        Sample3_15 s1 = new Sample3_15(1);
        System.out.println(s1.returna());
        System.out.println(s1.returnb());
        System.out.println(s1.returns());
        Sample3_15 s2 = new Sample3_15(2);
        System.out.println(s2.returna());
```

```
        System.out.println(s2.returnb());
        System.out.println(s2.returns());
    }
}
```

运行结果如图 3.12 所示。

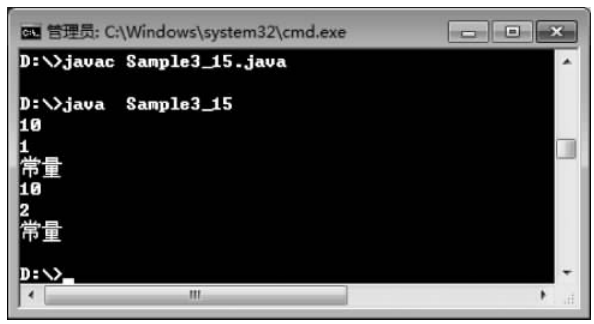

图 3.12 例 3-15 运行结果

3.4.5 abstract 关键字

abstract 关键字可用于修饰方法和类，分别称为抽象方法和抽象类。

1. 抽象方法

采用 abstract 关键字修饰的方法称为抽象方法，这种方法只有方法声明，而没有方法体，也就是没有方法的具体实现细节。

例如：

```
public abstract double getArea();          //声明一个抽象方法 getArea();
```

抽象方法只能在抽象类里定义，并且抽象方法前不可以使用 static、final、private 修饰符，也就是说 abstract 关键字不可以与 static、final 和 private 联合使用；另外，构造方法不可以声明为 abstract 方法。抽象方法必须在子类中实现(重写)后才可以使用。

2. 抽象类

采用 abstract 修饰的类称为抽象类。抽象类不允许实例化对象，但可以声明对象。

例如：

```
public abstract class countArea
{
    public final double PI = 3.14;
    public  abstract double getArea();     //抽象类中包括一个抽象方法
}
```

抽象类必须在其子类中将抽象方法全部实现后才可以用子类实例化对象，通过子类对象调用已经被实现的抽象方法；抽象类不允许声明为 final 类，因为抽象类必须被继承；抽象类中可以包括抽象方法，也可以不包括抽象方法，但如果类中包括抽象方法则该类必须声

明为抽象类，并且继承于某一抽象类的子类，如果没有实现抽象类中的全部抽象方法，则该子类也必须声明为抽象类。

例 3-16　使用 abstract 关键字。

源文件为 Sample3_16.java，代码如下。

```
abstract class countArea                    //抽象类
{
    final double PI = 3.14;
    abstract double getArea();              //抽象方法
}
class Circle extends countArea              //继承于抽象类
{
    double r;
    Circle (double r)
    {
        this.r = r;
    }
    double getArea()                        //实现抽象方法，否则 Circle 必须声明为抽象类
    {
        return PI * r * r;
    }
}
class Triangle  extends countArea           //继承于抽象类
{
    int bottom,height;
    Triangle(int bottom,int height)
    {
        this.bottom = bottom;
        this.height = height;
    }
    double getArea()                        //实现抽象方法，但具体实现功能与 Circle 中的不同
    {
        return 0.5 * bottom * height;
    }
}
public class Sample3_16
{
    public static void main(String args[])
    {
        Circle  c = new Circle(5);
        Triangle t = new Triangle(5,10);
        System.out.println("圆的面积为：" + c.getArea());   //调用类中已经实现的方法
        System.out.println("三角形面积为：" + t.getArea()); //调用对象所在类实现的方法
    }
}
```

运行结果如图 3.13 所示。

抽象类一般在面向对象程序开发过程中设计阶段使用，使设计者可以集中精力于全局结构的设计而不必费心于具体的实现细节，从而设计出优化的软件结构。

图 3.13 例 3-16 运行结果

3.4.6 上转型对象

将子类创建对象的引用赋值给父类声明的对象，则称此父类对象是该子类对象的上转型对象。

例如，假设类 A 是类 B 的父类：

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

则称 a 是 b 的上转型对象。上转型对象的使用受到一定的限制，具体使用规则如下。

(1) 上转型对象不能调用子类新增的成员变量，也不能调用新增的方法。

(2) 上转型对象可以访问子类中继承或隐藏的成员变量，也可以调用子类继承或覆盖的方法。如果子类重写的是父类的实例方法，则调用的是子类中重写的方法，否则调用的是父类中的方法。

另外，在使用上转型对象时，不要将上转型对象和父类创建的对象混淆。因为上转型对象的实体是子类创建的，而父类对象的实体是父类创建的，两种实体的结构不同；其次，由于上转型的规定，上转型对象失去了原子类对象的部分属性和功能(子类中新增的)，但如果将上转型对象强制转换为子类类型，并将其引用赋值给一个子类对象，则该子类对象可以访问或调用子类中的全部成员；最后，不可将父类创建对象的引用赋值给子类对象，即不可向下转型。

例 3-17 使用上转型对象。

源文件为 Sample3_17.java，代码如下。

```
abstract class Fly
{
    abstract void flyAction();
}
class WildGoose extends Fly
{
    void flyAction()
    {
        System.out.println("The wild goose in rows……");
    }
```

```
}
class Sparrow extends Fly
{
    void flyAction()
    {
        System.out.println("The sparrow flocks……");
    }
}
class Eagle extends Fly
{
    void flyAction()
    {
        System.out.println("The eagle circled alone……");
    }
}
class Bird
{
    void birdFly(Fly birdfly)
    {
        birdfly.flyAction();
    }
}
public class Sample3_17
{
    public static void main(String args[])
    {
        Bird bird = new Bird();
        WildGoose  d = new WildGoose();
        Sparrow  m = new Sparrow();
        Eagle  e = new Eagle();
        bird.birdFly(d);
        bird.birdFly(m);
        bird.birdFly(e);
    }
}
```

运行结果如图 3.14 所示。

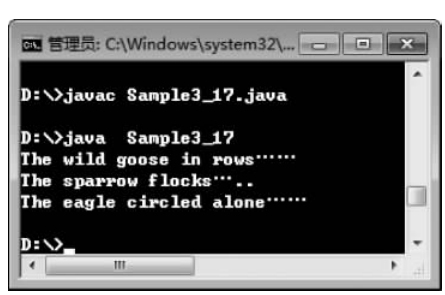

图 3.14　例 3-17 运行结果

3.5 内部类

类中可以声明成员变量、成员方法、static 块以及内部类，前三种成员前面已经介绍，本节讲述内部类。内部类是指在一个类中声明另一个类，这样的类称为内部类，而包含内部类的类相应地称为外部类。Java 规定内部类中可以使用外部类的成员，使用内部类可以方便类之间信息的交互。

根据内部类声明的位置可以将内部类分为成员内部类和局部内部类，另外还有一种特殊的内部类——匿名类。下面分别加以介绍。

3.5.1 成员内部类

声明在类中(成员方法外)的类，称为成员内部类。成员内部类与成员变量和成员方法的规定类似，也可以采用 public、protected、private、static、final、abstract 等修饰符。并且外部类可以使用成员内部类声明对象，作为外部类成员。

例 3-18 使用成员内部类。

源文件为 Sample3_18.java，代码如下。

```
class Outer
{
    innerOne in1 =  new innerOne();                    //使用成员内部类声明对象,作为外部类的成员
    innerTwo in2 = new innerTwo();
    String OutStrOne = "非静态外部类成员变量";
    static String outStrTwo = "静态外部类成员变量";
    class innerOne
    {
        String inStrOne = "非静态内部类成员变量";
        //static String  inStrTwo;不允许在非静态成员内部类中声明静态成员
        void print()
        {
            System.out.println(OutStrOne);
            System.out.println(outStrTwo);
            System.out.println(in1.inStrOne);
            System.out.println(in2.sinStrOne);
            System.out.println(in2.sinStrTwo);
        }
    }
    static class innerTwo
    {
        String sinStrOne = "静态内部类非静态成员变量";
        static String sinStrTwo = "静态内部类静态成员变量";
        void print()
        {
            System.out.println(outStrTwo);
            //System.out.println(OutStrOne); 不允许在静态成员类中使用非静态成员
            //System.out.println(in1.inStrOne);
```

```
            //System.out.println(in2.sinStrOne);
            //System.out.println(in2.sinStrTwo);
        }
    }
    void print()
    {
        System.out.println(OutStrOne);
        System.out.println(outStrTwo);
        System.out.println(in1.inStrOne);
        System.out.println(in2.sinStrOne);
        System.out.println(in2.sinStrTwo);
    }
}
public class Sample3_18
{
    public static void main(String args[])
    {
        Outer o = new Outer();
        Outer.innerOne s1 = o.new innerOne();        //使用非静态内部类声明并实例化对象
        Outer.innerTwo s2 = new  Outer.innerTwo();  //使用静态内部类声明并实例化对象
        o.print();
        s1.print();
        s2.print();
    }
}
```

3.5.2　局部内部类

在成员方法中声明的类称为局部内部类，局部内部类与局部变量类似，只在方法内有效。

例 3-19　使用局部内部类。

源文件为 Sample3_19.java，代码如下。

```
class Outer
{
    String outStrOne = "外部类成员变量";
    void outerPrint()
    {
        class Inner
        {
            String inStrOne = "局部内部类变量";
            void innerPrint()
            {
                System.out.println(outStrOne);
                System.out.println( inStrOne);
            }
        }
        Inner in1 = new Inner();
```

```
            in1.innerPrint();
        }
}
public class Sample3_19
{
    public static void main(String args[])
    {
        Outer o = new Outer();
        o.outerPrint();
    }
}
```

3.5.3 匿名类

匿名类是指没有名称的内部类,常用于事件处理程序。

例 3-20 使用匿名类。

源文件为 Sample3_20.java,代码如下。

```
class Father
{
    void print()
    {
        System.out.println("fahter");
    }
}
public class Sample3_20
{
    public static void main(String args[])
    {
        Father s = new Father()
        {
            public void print()
            {
                System.out.println("son");
            }
        };
        s.print();                                    //运行结果为: son
    }
}
```

需要注意的是:在 Java 中,每个 class 和 interface 关键字定义的类和接口,编译后都会产生一个字节码文件.class。因此,每个内部类编译同样会产生一个字节码文件,但内部类产生的字节码文件的名字与通常的类略有不同。成员内部类的字节码文件名为“外部类名$内部类名.class”,局部内部类的字节码文件名为“外部类名$编号内部类名.class”。其中,编号表示该内部类是所在方法中定义的第几个内部类。另外,匿名类产生的字节码文件名为“外部类名$编号.class”。

3.6　接口

接口与类相似，也是一种重要的引用数据类型，其作用与抽象类相似但又不同，接口主要用于描述不同类的共有行为，但不包括行为的具体实现细节。当某个类要展现这些行为只要实现该接口，并具体设计行为细节。另外，Java 语言仅支持单继承，虽然单继承简化了管理，但无法解决实际应用中必须由多继承解决的问题。而接口弥补了这一缺陷，也就是可以通过接口技术间接实现多继承。

3.6.1　定义接口

Java 语言中使用 interface 定义接口，与类相似，接口也包括接口声明和接口体两部分。并且接口体中仅包括公共静态常量和公共抽象方法。

定义接口的一般语法格式如下：

```
[修饰符] interface 接口名 [extends 父接口列表]      //接口声明
{
    [public] [static][final]常量;                   //接口体
    [public][abstract]方法;
}
```

参数说明：

(1) 接口声明中的修饰符可以为 public，表示公共接口，用于控制该接口的访问权限范围。

(2) 接口体中使用的修饰符如 public、static、final、abstract 都可以省略，但表示的含义相同。

例如：

```
interface Example
{
    public PI = 3.14;
    void print();
    int max(int a,int b);
}
```

等价于：

```
interface Example
{
    public static final double PI = 3.14;
    public abstract void print();
    public abstract int max(int a,int b);
}
```

(3) 接口允许继承，并且可以是多继承。

例 3-21　接口定义。

源文件为 Sample3_21.java，代码如下。

```
interface Action
{
    void act();
}
interface Breathing
{
    void breathe();
}
interface Speak
{
    void language();
}
interface Play
{
    void performance();
}
interface Fish extends Action,Breathing
{
    String name = "鱼";
    void live();
}
```

注意：接口与抽象类不同，接口中只有常量和抽象方法；抽象类中可以有常量、变量、抽象方法和非抽象方法。接口采用 interface 定义，支持多继承；抽象类采用 abstract 定义，支持单继承。

3.6.2 实现接口

Java 语言中由类实现接口。所谓实现接口就是具体实现接口中的方法。Java 规定，非抽象类实现接口时必须实现该接口中的全部方法，包括该接口继承于父接口的全部方法，否则该类必须声明为抽象类。在类中采用 implements 关键字实现接口，一个类可以同时实现多个接口，如果多个接口有同名的常量则可以用“接口名.常量名”的方式引用；如果有同名的方法（参数也相同）则只实现一个方法即可。

例 3-22 实现接口。

源文件为 Sample3_22.java，代码如下。

```
class GoldFish implements Fish
{
    public void live()
    {
        System.out.println("live in water!");
    }
    public void act()
    {
        System.out.println(name + " can swim!");
    }
    public void breathe()
```

```
    {
        System.out.println("I breathe with gills!");
    }
}
class Human implements Action,Breathing,Speak,Play
{
    String name;
    Human (String Str)
    {
        name = Str;
    }
    public void act()
    {
        System.out.println("I can walk!");
    }
    public void breathe()
    {
        System.out.println("I breathe with lung!");
    }
    public void language()
    {
        System.out.println("I speak Chinese!");
    }
    public void  performance()
    {
        System.out.println("I can sing!");
    }
}
public class Sample3_22
{
    public static void main(String args[])
    {
        GoldFish fish = new GoldFish();
        System.out.println(fish.name);
        fish.live();
        fish.act();
        fish.breathe();
        Human people = new Human("I am Chinese!");
        System.out.println(people.name);
        people.act();
        people.breathe();
        people.language();
        people.performance();
    }
}
```

运行结果如图 3.15 所示。

图 3.15 例 3-22 运行结果

另外,在事件处理程序中,实现接口可以采用匿名内部类。

例如:

```
item1_2.addActionListener(new ActionListener() //实现接口 ActionListener 的匿名内部类,实现
//了该接口中的 actionPerformed(ActionEvent Event)方法
{
     public void actionPerformed(ActionEvent Event)
     {
        int i = JOptionPane.showConfirmDialog(null,"是否真的需要退出系统","退出确认对话
框", JOptionPane.YES_NO_CANCEL_OPTION);
        if(i == 0)
        {
               System.exit(0);
        }
       }
   });
```

3.6.3 接口回调

接口是一种引用数据类型,可以存储对象引用。接口本身只能用于声明接口变量,不能实例化。但可以把实现了该接口的类创建的对象引用赋值给接口变量,之后接口变量就可以调用该类实现的接口方法,这就是接口回调。

例 3-23 接口回调。

源文件为 Sample3_23.java,代码如下。

```
interface Action
{
    void act();
}
interface Breathing
{
    void breathe();
}
```

```
class Bird implements Action,Breathing
{
    String skin = "feather";
    public void act()
    {
        System.out.println("I can fly");
    }
    public void breathe()
    {
        System.out.println("I breathe with lung!");
    }
    public void print()
    {
        System.out.println("I have colorful feathers!");
    }
}
public class Sample3_23
{
    public static void main(String args[])
    {
        Action a;
        Breathing b;
        Bird bird = new Bird();
        a = bird;
        b = bird;
        a.act();                                    //a.breathe(),a.print()不可以
        b.breathe();                                //b.act(),b.print() 不可以
        bird.print();                               //bird.act.bird.breathe()可以
        System.out.println(bird.skin);              //b.skin,a.skin 不可以
    }
}
```

运行结果如图 3.16 所示。

图 3.16　例 3-23 运行结果

上述程序内存模型及可调用成员关系如图 3.17 所示。

说明：接口回调时，接口变量只能调用类实现的该接口中的成员，而不能调用类中其他成员。

接口回调技术常用于接口变量用作参数的情况，此时可以将任何实现该接口的类的实

例的引用传递给该接口参数，那么接口参数就可以回调类实现的接口方法。而实现该接口的类可能具有不同的实现方式，那么接口回调接口方法时就产生多种形态（多态性）。

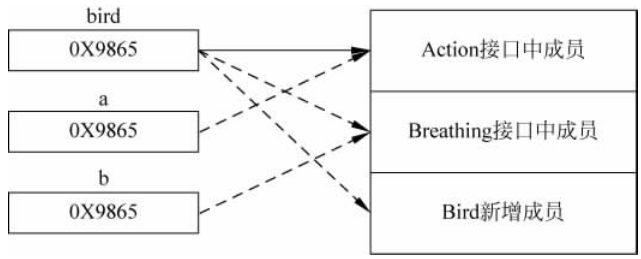

图 3.17　接口回调内存模型

3.7　API 查询方法

Java 提供了丰富的 API 文件（类库），例如，前面使用的 System 类、Scanner 类以及 String 类等。目的是利于应用程序开发人员开发 Java 程序。那么 Java API 文件中包括哪些类和接口呢？如何使用呢？下面介绍 Java API 的查询方法。

具体查询步骤如下。

（1）打开 Java 官方网站 http://www.oracle.com/technetwork/java/javase/downloads/index.html，在 Java SE 下载界面右侧栏中找到 Java APIs，如图 3.18 所示。

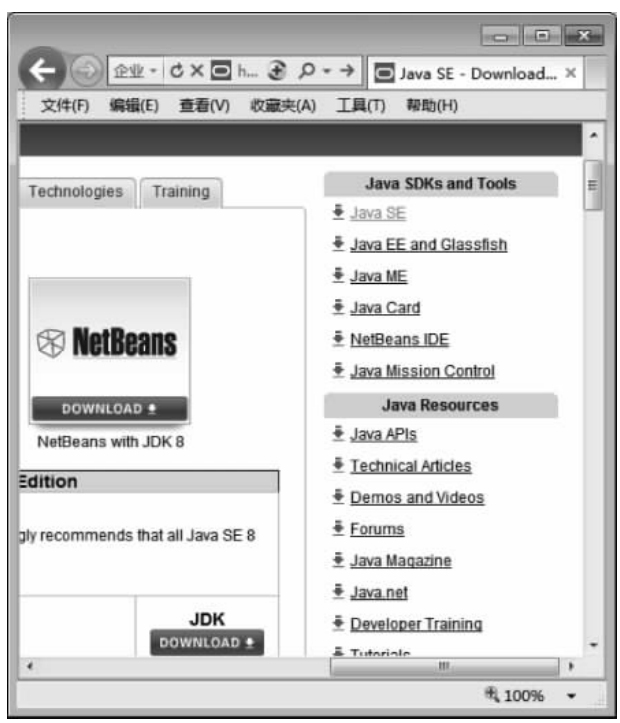

图 3.18　Java API 查询界面 1

(2) 单击图 3.18 中的 Java APIs 链接，打开 Java API 规范界面，如图 3.19 所示，从中选择所需版本。

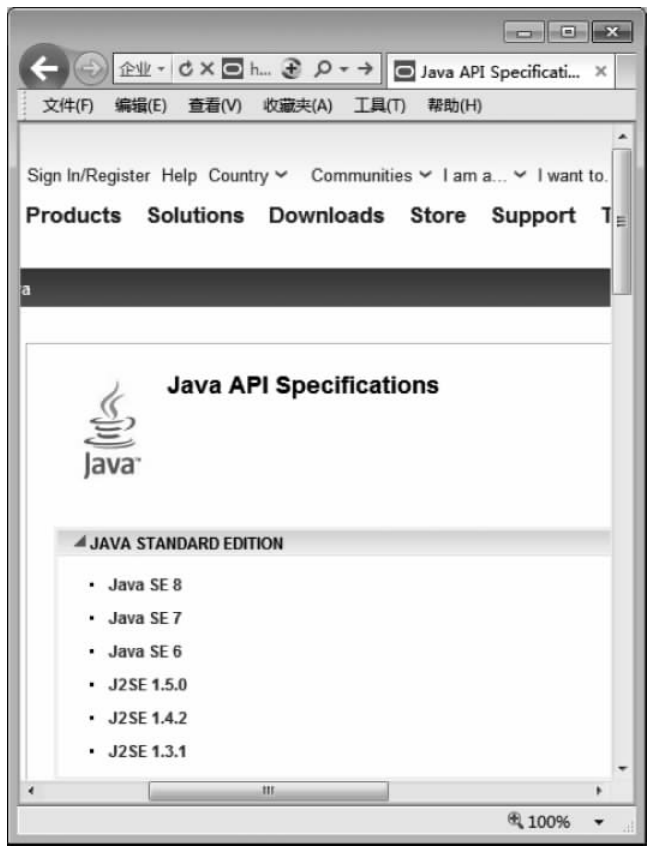

图 3.19　Java API 查询界面 2

(3) 单击图 3.19 中的 Java SE 8，打开 Java SE 8 API 规范界面，如图 3.20 所示。

(4) 图 3.20 左上窗口是包分类，单击需要查看的包，例如 java.io，左下窗口将列出该包中的接口和类。单击需要查看的类，例如 File，右侧窗口就会显示该类的说明，如图 3.21 所示。

(5) 一个类中通常包括以下几部分信息。

① Field Summary：列出类中成员变量，包括名称、类型及含义，如图 3.22 所示。

② Constructor Summary：列出类的构造方法信息，如图 3.23 所示。

③ Method Summary：列出该类中的方法，如图 3.24 所示。

④ Field Detail：成员变量详细信息。

⑤ Constructor Detail：构造方法详细信息。

⑥ Method Detail：成员方法详细信息。

(6) 单击图 3.22～图 3.24 中的变量、方法或者向下滑动右侧窗口的滚动条，可以查看类的详细信息。如图 3.25 所示为 File 类 canRead()方法的详细介绍。

除了链接 Java 官方网站查看 API，也可以将 Java API 文档下载到本地机器，解压后进行查看，方法与前面介绍的基本相同，如图 3.26 所示。

图 3.20 Java API 查询界面 3

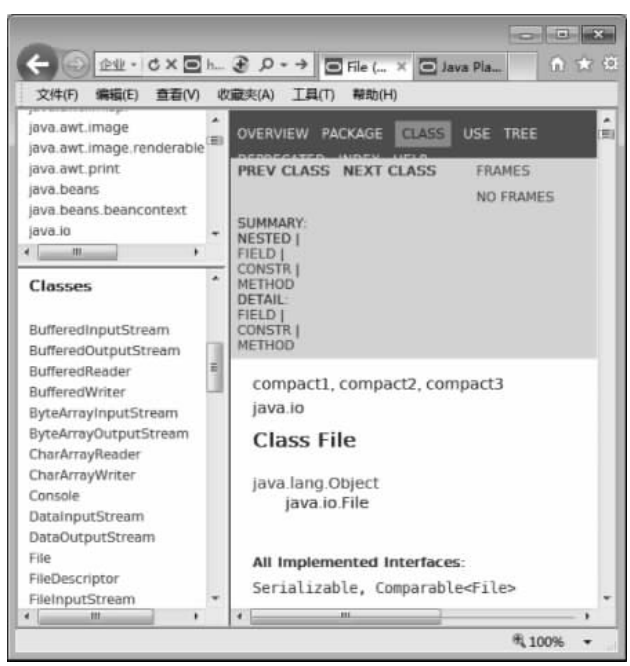

图 3.21 Java API 查询界面 4

图 3.22　Java API 查询 Field Summary 界面

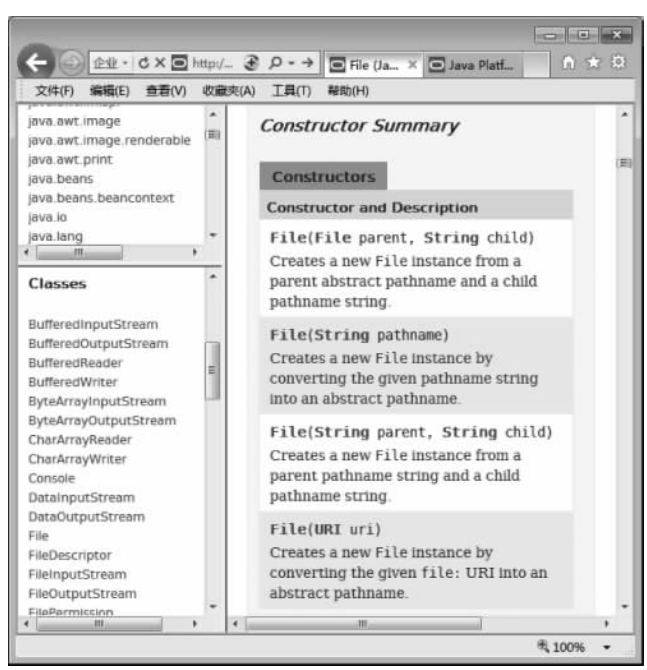

图 3.23　Java API 查询 Constructor Summary 界面

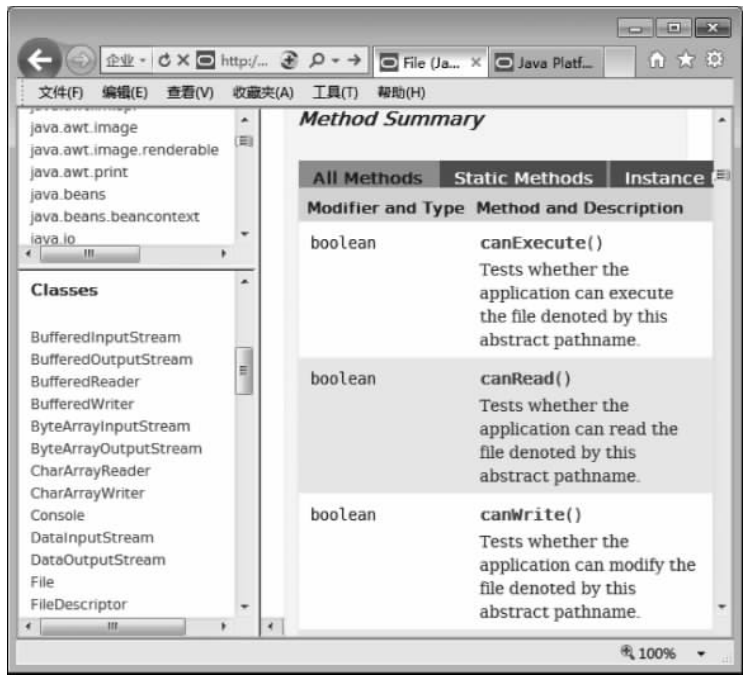

图 3.24 Java API 查询 Method Summary 界面

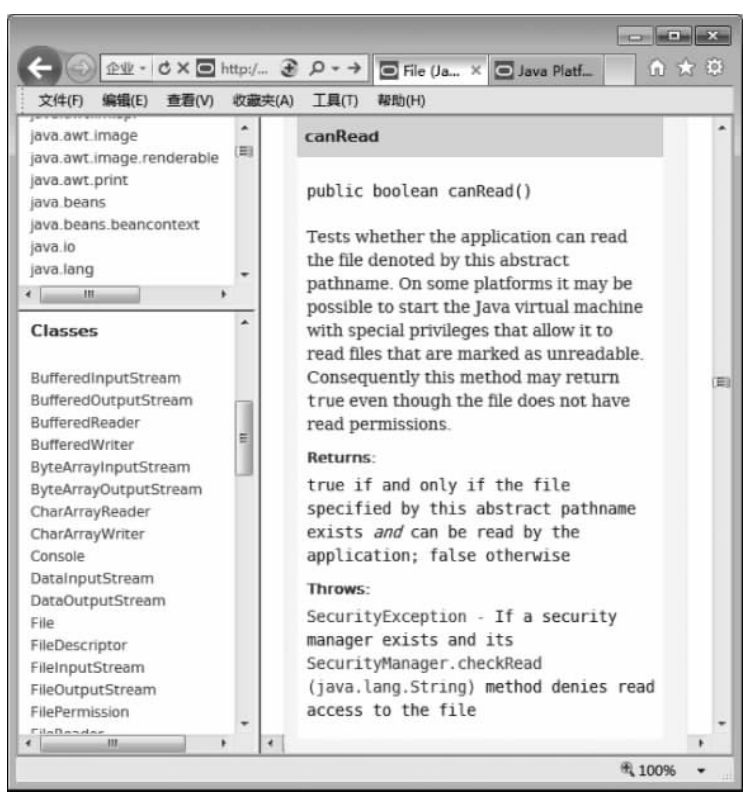

图 3.25 Java API 查询 Method Detail 界面

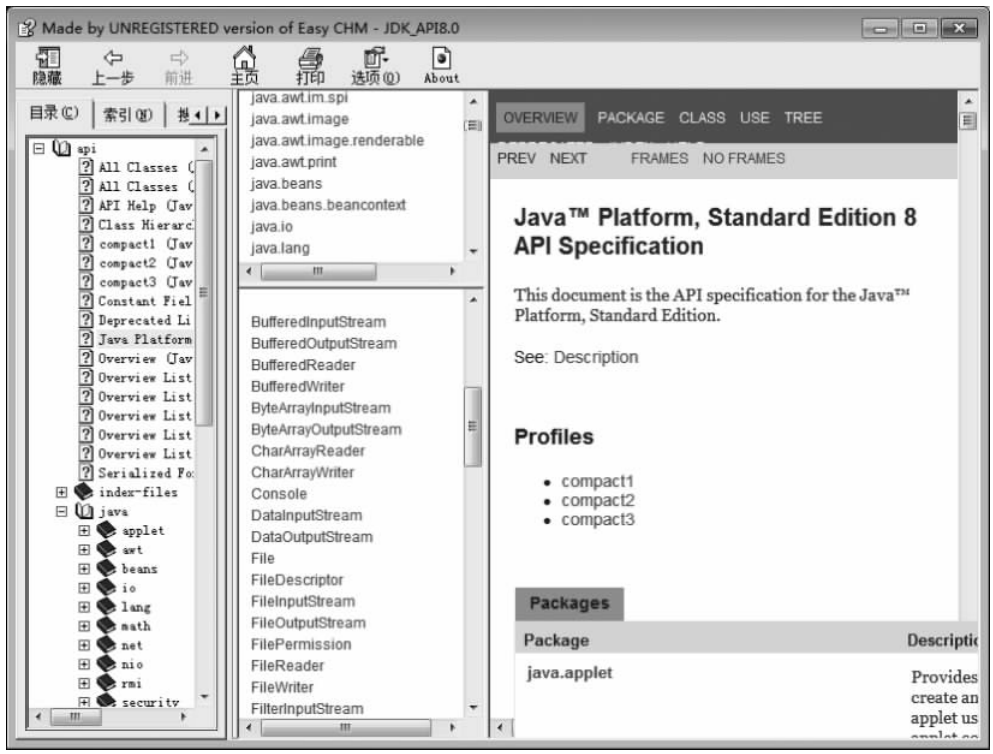

图 3.26　本地 Java API 查看界面

另外，也可以利用搜索引擎查看所需类，如图 3.27 所示。

图 3.27　搜索引擎中查看 Java API

Java API 非常丰富，熟练查询 Java API 文档可以让开发人员更快更好地使用 Java。

小结

本章主要讲述了面向对象的基础知识，包括面向对象编程的基本思想、方法、特点；类与对象的概念、含义以及具体使用方法；继承的含义、作用、原则；多态的概念及具体实现方法，例如重载、重写、接口回调等；包的概念、作用及使用方法；接口的概念、作用及定义和实现方法；最后介绍了 Java API 的查询方法。

在讲述面向对象基础知识的过程中涉及众多关键字，总结如下。

(1) 与类和接口相关的访问权限控制修饰符。

① public：用于修饰类和接口时，表示公共类和公共接口，这样的类和接口不仅可以被同一包中的其他类和接口调用，也可以被引入到其他包的源文件中供调用。

② 默认修饰符：当声明类和接口时如果没有采用 public 修饰符，也就是采用默认修饰符时，这样的类和接口只能被同一包中的其他类和接口调用，不在同一个包中的类和接口不可以使用。

(2) 与成员变量和成员方法相关的访问控制权限修饰符。

① public：用于修饰成员变量和成员方法时，表示公共成员变量和成员方法，这样的成员变量和成员方法不仅可以在本类中使用，也可以被同一包或不同包中的其他类使用。

② protected：由 protected 修饰的成员变量和成员方法表示受保护的，这样的成员变量和成员方法可以在本类中使用，也可以被同一包中的其他类使用，还可以被不在同一包中该类的子类使用。但不能被不在同一包中又没有继承关系的类中使用。

③ 默认修饰符：当成员变量和成员方法前面无访问权限修饰符时，表示采用的是默认修饰符，这样的成员变量和成员方法只能在本类或同一包中的其他类中使用，不在同一包中的其他类不可以使用。

④ private：由 private 修饰的成员变量和成员方法是私有的，只能在本类中使用，其他类中不可使用。

(3) 与类声明相关的其他修饰符和参数。

① final：表示最终类，这样的类不可以被继承，没有子类。

② abstract：表示抽象类，可声明对象变量，但不能用于实例化(创建对象)。

③ extends：用于声明类的继承关系。Java 规定，类仅支持单继承。

④ implements：用于声明类实现的接口，一个类可以实现多个接口。

⑤ class：用于声明类。

(4) 与接口声明相关的其他参数。

① interface：用于声明接口。

② extends 用于声明接口的继承关系，接口支持多继承。

(5) 与成员变量声明相关的其他修饰符。

① final：表示常量。

② static：表示类变量或静态变量，该类各实例对象共有；没有 static 修饰的成员变量称为实例变量，该类各实例各自私有。类变量可以采用类名直接调用，而实例变量只能采用对象名调用。staitc 不能用于修饰局部变量。

(6) 与成员方法相关的其他修饰符。

① final：表示最终方法，不允许被重写。

② abstract：表示抽象方法，这样的方法只有方法声明而没有方法体，必须被重写后才能使用。

③ static：表示静态方法或类方法，与类变量相似，类方法可以使用类名直接调用，也可以采用对象名调用，而非静态方法只能使用对象名调用。

另外，接口中的成员变量必须修饰为 public static final，成员方法必须修饰为 public abstract。声明时修饰符可以省略不写。

(7) 在类中采用的几个关键字。

① this：表示当前对象，用于调用该类其他的构造方法或被隐藏的成员变量，不能用在静态方法中。

② super：用于调用父类的构造方法，此时必须写在当前类构造方法的开始位置，另外，super 也用于调用被子类屏蔽、隐藏或覆盖的父类的成员变量或成员方法。

③ new：用于创建对象(实例化对象)。

(8) 与包相关的关键字。

① package：用于声明包，必须写在 Java 的源文件第一条语句。

② import：引入类，写在包声明语句后。类和接口声明之前。一个源文件可以使用多个 import 语句。

思考练习

1. 思考题

(1) 比较面向对象与面向过程程序设计的不同。

(2) 什么是封装、继承与多态？

(3) 什么是重载？什么是重写？二者有何不同？

(4) 简述类与对象的关系。

(5) 构造方法可以重载吗？可以重写吗？

(6) 简述类变量与实例变量的区别。

(7) 简述类方法与实例方法的区别。

(8) 子类可以继承父类的哪些成员？

(9) 简述 final 的含义。

(10) 如何调用被局部变量屏蔽的成员变量？

(11) 如何调用父类的构造方法和被子类隐藏或覆盖的成员？

(12) 什么叫抽象类？有何作用？

(13) 简述类的结构及类声明中各个参数的含义。

(14) 简述包的作用及声明的方法。

(15) 简述有包声明语句的 Java 程序编译及运行的方法。

(16) 简述 Java 访问控制权限的种类及含义。

(17) new 关键字有什么作用?

(18) 简述局部变量与成员变量的区别。

(19) 如何在类的外部访问类中的私有成员?

(20) 构造方法与普通方法有何不同?

(21) 如何在 Java 源文件中引入需要的类?

(22) 什么是上转型对象? 叙述上转型对象的使用原则。

(23) 接口有何用途? 如何声明接口? 如何实现接口?

(24) 如何通过接口实现多继承?

(25) 什么是匿名类?

2. 拓展训练题

(1) 编程测试有包声明语句的 Java 程序编译及运行的方法。

(2) 编程测试重载方法的声明及使用的方法。

(3) 编程测试各访问控制权限修饰符的作用。

(4) 编程测试继承的原则。

(5) 编程测试隐藏与覆盖的规则。

(6) 编程测试构造方法的重载与使用方法。

(7) 编程测试成员变量与局部变量的区别。

(8) 编程测试 Java 中方法调用时参数传递的特点。

(9) 编程测试类成员与实例成员的不同。

(10) 编程测试抽象类及抽象方法的使用原则。

(11) 编程测试 final 类、方法及变量的使用原则。

(12) 编程测试父类中构造方法编写的限制。

(13) 编程测试各修饰符的组合使用情况,包括: public, protected, private, final, abstract, static。

(14) 编程测试内部类的使用方法。

(15) 编程测试接口的声明、实现及继承的原则。