第 5 章　深入讨论类

教学要求

- 理解引用赋值，类的静态成员，重载的概念，继承、抽象类、接口的概念，包的作用，访问控制修饰符的作用。
- 掌握对象的创建与使用，方法的定义和调用，构造方法的使用，方法重载的使用，通过继承创建新类，定义和使用接口，使用包，使用 import。

5.1　对象的创建与销毁

面向对象程序设计的核心是对象，程序是一系列对象的组合。单个对象能够实现的功能是有限的，应用程序往往包含很多的对象，这些对象相互调用彼此的方法，交互作用以实现更高级、更复杂的功能。在程序中当需要使用新对象时，应进行对象的创建，为其分配内存空间，在对象完成了自己的功能后可以销毁它以释放占用的内存。

创建对象就是指产生类的一个实例，就好比依照设计图纸实实在在地生产一辆车。要生产出产品首先要有设计图纸，创建对象也是如此：首先有类，然后才能产生这个类的实例。

在 Java 中，利用 new 关键字可以创建类的对象，这是最常用的创建对象的方式。

用 new 创建对象的语法格式如下所示：

```
SomeClass ob = new SomeClass(参数列表);
```

new 关键字后调用的是类的构造方法。前面章节已经介绍过，类中有一种特殊的方法，其名称与类名相同，无返回类型，就是类的构造方法，用于对对象进行初始化操作。

对象的创建过程主要包含了如下的步骤：

（1）给对象分配内存空间。

（2）若类的实例变量在定义时未赋初值，则将它们自动初始化为其所属类型的默认值。如 int 类型、double 类型等数值类型的默认值为 0，boolean 类型的初始值为 false 等。若实例变量在定义时就赋了值，则按给定的值进行初始化。

（3）调用类的构造方法进行对象的初始化工作，若构造方法中包含对实例变量赋值的语句，则为实例变量赋予相应的初始值。

注意：类的实例变量，指的是类中未被 static 修饰的成员变量，static 的作用后续章节会介绍。

假设有如下的学生类 Student 的定义：

```
public class Student {
    String name;
    int age;
    String major;
```

```
    public Student(String stuName, int stuAge, String stuMajor) {
        name = stuName;
        age = stuAge;
        major = stuMajor;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int newAge) {
        age = newAge;
    }
    public String getMajor() {
        return major;
    }
    public void setMajor(String newMajor) {
        major = newMajor;
    }
    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
}
```

可以使用 new 来创建具体学生对象，如：

```
Student aStudent = new Student("Tom",19,"计算机");
```

例 5-1　NewDemo1.java

```
public class NewDemo1 {
    public static void main(String[] args) {

        // 创建一个学生对象
        Student aStudent = new Student("Tom", 19, "计算机");

        // 显示这个学生的相关信息
        System.out.println("学生姓名: " + aStudent.getName());
        System.out.println("学生年龄: " + aStudent.getAge());
        System.out.println("学生专业: " + aStudent.getMajor());
    }
}
```

运行结果如图 5-1 所示。

对象创建之后，可以通过访问对象的变量或调用对象的方法来使用对象。

1. 访问对象的变量

访问对象的变量的一般格式如下：

```
对象名.变量名
```

图 5-1　例 5-1 运行结果

通过访问对象的变量可以修改对象的属性值，如可以给学生改名：

```
aStudent.name = "John";
```

类似的，还可以访问学生对象的 age、major 变量修改其年龄、专业信息：

```
aStudent.age = 20;
aStudent.major = "英语";
```

需要注意的是：像这样直接操纵对象的属性的做法是不提倡的，更好的访问方式是通过调用各个属性的 get 和 set 方法来访问。

2. 调用对象的方法

调用对象的方法的一般格式如下：

对象名.方法名(参数列表)

方法代表对象具有的行为，调用方法就相当于实施行为，能实现一定的功能。

比如，如果希望对这个学生的信息进行修改，可以调用各个属性的 set 方法来实现：

```
aStudent. setName("John");
aStudent.setAge(20);
aStudent.setMajor("英语");
```

3. 缺省构造方法

现在修改一下 Student 类，将其中的构造方法去掉，没有构造方法，还能创建学生对象吗？

答案是可以，但是在 NewDemo1 类中构造学生对象的语句要做相应的改动：

```
Student aStudent = new Student();
```

再次运行 NewDemo1 类，得到如下的结果：

图 5-2　例 5-1 运行结果 2

运行结果表明，学生对象创建成功了，但各属性值被初始化为默认值。

这是因为在 Java 中，每个类至少要有一个构造方法，如果用户在定义类时，并没有定义任何的构造方法，则 Java 将自动提供一个隐含的构造方法，叫做缺省的构造方法。

这个缺省的构造方法不带参数，用 public 修饰，而且方法体为空，形如：

```
public ClassName(){}
```

因此，修改后的 Student 类，事实上包含无参的缺省构造方法，使用 new 创建对象时，就可以调用这个构造方法来创建对象了。

由于实例变量在定义时未赋初值，缺省的构造方法的方法体也是空的，故这些实例变量均只被赋予了所属类型的默认值 null、0 和 null，即得到上面所示的运行结果。

4. 对象的销毁

对象创建之后就占用一定的内存空间，Java 程序会陆续创建许多的对象，若这些对象一直占用内存而不释放的话，内存总会被耗尽，最后引发内存空间不足的问题。因此，当对象已经不再需要的时候，应该及时地销毁它们，以释放内存空间，保证内存空间的有效利用。

在其他一些语言中，当对象不再需要时，程序可能要显式地利用一定的语句销毁对象，释放对象所占用的内存空间。这导致了一定的弊端：由于程序员的粗心，可能会忘记及时释放无用对象占用的内存，也有可能错误地释放了不该释放的内存而导致系统问题等。

而 Java 程序员则可以轻松一些，在 Java 中，销毁无用对象，回收其占用的内存资源这一工作由 Java 虚拟机来承担，这使得程序员可以从复杂的内存追踪、检测和释放中解放出来，减轻程序员进行内存管理的负担。

在 Java 的运行时环境中，Java 虚拟机提供了一个垃圾回收器线程，它负责自动周期性地检测、回收那些无用对象所占用的内存，这种内存回收的机制被称为自动垃圾回收（Garbage Collection），这也是 Java 显著的特色之一。

5.2 引用赋值

我们知道，同种基本数据类型的变量之间可以互相赋值，比如：

```
int i = 3;
int j = i;
```

与此类似，同种类型的对象之间也可以赋值，比如：

```
Student stu1 = new Student();
Student stu2 = stu1;
```

不同的是：

✧ 同种基本数据类型的变量之间的赋值是值的拷贝，内容的拷贝。

✧ 同种类型的对象之间的赋值叫做引用赋值，不是拷贝对象的内容。

对象名其实也是变量名，它所属的类型就是实例化它的类，如上面代码中的 stu1、stu2 就是 Student 类型的变量。与基本数据类型的变量不同的是，对象名 stu1、stu2 并不存储具体的学生对象的内容，而只是代表了指向学生对象存储空间的一个引用，因此，对象也被称为引用类型变量。

对象名的引用作用如图 5-3 所示。

同种类型对象之间的赋值，是将一个对象名所代表的引用赋给另一个对象名，使得两个对象名具有相同的引用，那么这两个对象名可以访问到同一个对象的存储空间，这种对象的赋值就叫做引用赋值。

假设已有学生类 Student（见 5.1 节），通过引用赋值，可以用两个不同的名称来表示同一个学生，如下例所示。

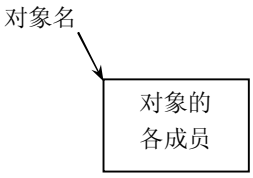

图 5-3　对象名的引用作用

例 5-2　FuZhi.java

```
public class FuZhi {

    public static void main(String[] args) {
        Student tom = new Student("Tom", 20, "计算机");
        Student tom_Smith;
        tom_Smith = tom; // 引用赋值

        // tom 的相关信息
        System.out.println("学生 tom 的基本信息如下：");
        System.out.println("  姓名：" + tom.getName());
        System.out.println("  年龄：" + tom.getAge());
        System.out.println("  专业：" + tom.getMajor());

        // tom_Smith 的相关信息
        System.out.println("学生 tom_Smith 的基本信息如下：");
        System.out.println("  姓名：" + tom_Smith.getName());
        System.out.println("  年龄：" + tom_Smith.getAge());
        System.out.println("  专业：" + tom_Smith.getMajor());
    }
}
```

运行结果如图 5-4 所示。

图 5-4　例 5-2 运行结果

引用赋值 tom_Smith = tom 的作用如图 5-5 所示。

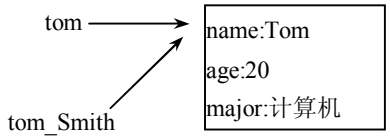

图 5-5　引用赋值的作用

若对象只声明了类型而没有创建，则对象的值为 null，代表空引用，表示没有指向任何的对象。这时候如果试图使用对象，则会出错，例如：

```
Student jane;
System.out.println("学生姓名：" + jane.getName());
```

使用空引用的对象，在运行时会产生 NullPointerException 异常，异常在后续章节中会有介绍。

5.3 深入讨论方法

5.3.1 方法的定义、调用和返回

类的组成主要包括成员变量和成员方法两部分：成员变量描述属性，成员方法描述行为。对象所具有的行为能力都是通过方法体现的。

下面我们具体讨论在 Java 中如何定义和调用方法，以及方法如何返回值。

1. 方法的定义

在 Java 中，方法是类的组成部分之一，每个方法都属于特定的类，方法的定义必须位于类体之中。

方法定义的一般格式为：

```
返回值类型  方法名（形式参数列表）{
    // 方法体
}
```

左大括号{之前的部分为方法定义的头部，主要包括返回值类型、方法名和形式参数列表。

- ✧ 方法名，必须是合法的标识符，应按照 Java 的命名规范进行命名，最好还能做到见名知意。
- ✧ 返回值，如果方法执行完后会带回一个值，这个值就叫做返回值。
- ✧ 返回值类型，表示该方法执行完毕后带回的值所属的类型，可以是基本数据类型，也可以是对象类型。但如果方法只是完成一定的操作而没有带回任何的值，则返回值类型应该声明为 void。
- ✧ 形式参数列表：使用这个方法实现一定功能时需要传递给方法的若干数据，可以没有，也可以是多个，各个参数之间要用逗号“,”分隔，每个参数都必须带上类型的声明。
- ✧ {和}之间是方法体，包含若干的可执行语句，用于实现方法的功能。
- ✧ 方法定义时的可缺省部分这里并未列出，后面的章节将讨论到其他部分。

以下就是一个合法的方法声明：

```
int add(int a,int b){
    int sum=0;
    sum = a+b;
    return sum;
}
```

add()方法实现求两个加数的和，当需要使用该功能时，应调用 add()方法，提供两个实际

的加数分别传给形式参数 a 和 b，方法体结束之前通过“return sum;”语句将计算得到的和值带回给调用者。

例 5-3　Add.java,MethodDemo1.java

```
public class Add {

    public int add(int a, int b) {// [1]
        int sum = 0;
        sum = a + b;
        return sum;// [2]
    }
}

public class MethodDemo1 {

    public static void main(String[] args) {
        Add ob = new Add();
        int sum = ob.add(13, 26);// [3]
        System.out.println("2 个数的和值为：" + sum);
    }
}
```

运行结果如图 5-6 所示。

图 5-6　例 5-3 运行结果

注释[1]处，add()方法的返回值类型为 int 型，表示该方法执行完毕会带回 1 个 int 类型的值。注释[2]处，利用“return sum;”语句将和值作为方法的返回值带回给方法的调用者。

在 MethodDemo1 类中使用该方法的功能：创建 Add 类的对象 ob，调用其 add()方法，并提供 2 个加数。这里的 main()方法就是 add()方法的调用者，当执行至注释[3]处时，首先执行方法调用 ob.add(13,26)，此时将转去执行 add()方法的方法体，并且将 13、26 这两个实际参数分别赋给 a 和 b 这两个形式参数，方法体执行完毕，求得的和值由“return sum;”语句返回到调用处，则赋值运算符“=”右边就是和值，然后该值被赋给 sum 变量。

2. *方法的调用*

方法的定义描述了方法能实现的功能，但真正要使用方法的功能，需要通过方法的调用来实现，如上例 MethodDemo1.java 的注释[3]处的“ob.add(13,26)”语句，再比如经常使用的“System.out.println(字符串)”语句，就是调用了 System 类中的成员 out 对象的 println()方法，实现将参数字符串输出到控制台显示出来。

方法调用，代表要执行方法体，实现方法的功能。在发生方法调用时，执行流程将转去执行方法体。

方法由类的对象（静态方法由类名直接调用，参见 5.5 节）用圆点运算符“.”来调用，方法调用的格式为：

对象名.方法名（实际参数列表）

方法定义时，方法名后()中的参数称为形式参数，代表该方法在执行时需要哪些数据。如上例，加法功能需要两个加数，故 add()方法定义时，用形式参数 int a,int b 指明了需要的数据，那么调用该方法实现功能时，就应给出实际要做加法运算的数据，如上例的 13 和 26，它们被称作实际参数。

在调用方法时，应注意如下几点：

- ✧ 实际参数应与形式参数的个数、类型、顺序均保持一致。
- ✧ 实际参数的值将对应地传给形式参数。
- ✧ 若方法定义时未定义任何形式参数，则调用方法时参数列表留空，但是()一定不能省略。
- ✧ 若方法定义时的返回值类型非 void，则方法调用就相当于是同类型的一个值（因为有返回值的方法必会用 return 返回一个同类型的值），这种方法调用就可以出现在赋值运算符“=”的右边，如上例的注释[3]处所示。
- ✧ 若方法定义时的返回值类型为 void，则方法调用一定不能出现在赋值运算符“=”的右边。

利用方法实现两个整数的四则运算，如下例所示。

例 5-4 Arithmetic.java，MethodDemo2.java

```
public class Arithmetic {
    //加法
    public int add(int a, int b) {
        int sum = 0;
        sum = a + b;
        return sum;
    }
    //减法
    public int sub(int a, int b) {

        int result = 0;
        result = a - b;
        return result;
    }
    //乘法
    public int mul(int a, int b) {

        int result = 0;
        result = a * b;
        return result;
    }
    //除法
    public double div(int a, int b) {
```

```
        double result = 0;
        result = a * 1.0 / b;
        return result;
    }
}

import javax.swing.JOptionPane;

public class MethodDemo2 {

    public static void main(String[] args) {
        Arithmetic ob = new Arithmetic();
        String strA = JOptionPane.showInputDialog("请输入运算数 1：");
        int a = Integer.parseInt(strA);
        String strB = JOptionPane.showInputDialog("请输入运算数 2：");
        int b = Integer.parseInt(strB);
        //两数进行四则运算的结果
        System.out.println(a + "+" + b + "=" + ob.add(a, b));
        System.out.println(a + "-" + b + "=" + ob.sub(a, b));
        System.out.println(a + "*" + b + "=" + ob.mul(a, b));
        System.out.println(a + "/" + b + "=" + ob.div(a, b));

    }
}
```

运行结果如图 5-7 所示。

图 5-7　例 5-4 运行结果

注意实现除法的方法 div()，为提高运算精确度，返回类型定义为 double 类型，则返回的除法结果 result 变量也应为 double 类型。由于运算的参数都为 int 型，2 个 int 型的数做除法得到的还是 int 型的结果，故对其中一个除数 a 做了 a*1.0 的运算将其转化为 double 型的值后，那么除法运算就会按 double 类型来进行了。

Arithmetic 类的这些方法都有返回值，方法调用就相当于是一个值，因此可以在System.out.println(字符串)方法中利用字符串连接符“+”与其他字符串或数值连接起来，一起输出到控制台。

方法的参数可以是任意类型的，包括对象类型。如果参数是对象类型的，那么方法调用时，实际参数传递给形式参数的就是对象的引用了。

例如，使用 5.1 节定义的 Student 类作为方法的参数定义一个 modify()方法，该方法实现修改学生的年龄。

例 5-5　MethodDemo3.java

```
public class MethodDemo3 {

    public static void main(String[] args) {

        MethodDemo3 ob = new MethodDemo3();     //[1]
        Student jerry = new Student("Jerry", 19, "计算机");

        ob.modify(jerry, 21);       //[2]

        System.out.println("Jerry 现在的年龄是：" + jerry.getAge());
    }

    public void modify(Student stu, int newAge) {
        stu.setAge(newAge);
    }
}
```

运行结果如图 5-8 所示。

图 5-8　例 5-5 运行结果

在 MethodDemo3 类中定义了无返回值的 modify()方法，实现的操作是修改参数指明的学生的年龄。在 main()方法中调用该方法时，仍然要先创建对象（注释[1]）才能调用该方法（注释[2]）。该方法无返回值，方法调用加上分号即成为一条语句，一定不能将其置于赋值号的右边。

modify()方法中需要一个 Student 类型的参数，则调用时的实际参数应是一个 Student 类的对象，这里为 jerry。

此时，实际参数 jerry 传值给形式参数 stu 是引用赋值，使得 stu 与实际参数 jerry 指向的是同一个学生对象，因此，modify()方法体中对 stu 调用 setAge()方法进行年龄的修改效果等同于对 jerry 进行年龄的修改。

对象类型做参数，实参对形参进行的是引用赋值，这与基本数据类型做参数是不同的。

基本数据类型做参数时只是简单地将实际参数的值拷贝一份给形式参数，之后形式参数与实际参数就毫无关联了，在方法体中无论如何修改形式参数的值都不会对实际参数造成影响。

3. *方法的返回*

发生方法调用时会转去执行方法体，方法体执行完毕则从方法返回，表示方法调用结束，将返回到被调用处，如果有必要还可以返回值给调用者。

一般情况下，方法体执行完毕遇到方法体的}就从方法返回，但也有不同的情况，比如遇到 return 语句。

一旦方法体中有 return 语句被执行，就表示要从方法返回了，将终止方法的执行，即使 return 之后还有其他语句未被执行。

return 语句的格式如下：

```
return [表达式];
```

其后的表达式是可以缺省的部分，是否缺省视方法是否返回值而定。如果方法定义时声明了返回值类型，就必须使用 return 表达式返回一个同类型的值；如果方法的返回值类型被声明为 void，则方法体中不能用 return 表达式带回值。

例 5-6　ReturnDemo1.java，ReturnTest1.java

```
public class ReturnDemo1 {

    public int absolute(int x) {

        if (x < 0) {
            return -x;  //[1]
        }
        return x;       //[2]
    }
}
import javax.swing.JOptionPane;

public class ReturnTest1 {

    public static void main(String[] args) {
        ReturnDemo1 ob = new ReturnDemo1();

        String strX = JOptionPane.showInputDialog("请输入一个整数：");

        int x = Integer.parseInt(strX);

        System.out.println(x + "的绝对值为：" + ob.absolute(x));

    }

}
```

这里的

```
if (x < 0) {
```

```
        return -x;
    }
```

表示满足条件 x<0 时，才会执行其后{}中的语句；否则不执行。if 语句是实现分支结构的流程控制语句，第 6 章将会具体介绍。

运行 ReturnTest1，若输入负数，则满足 x<0 条件，将进入 if 分支执行，注释[1]处的 return 语句被执行，方法将提前返回，之后的“return x;”语句不会被执行；如果输入的是正数或 0，则不会进入 if 分支，注释[2]处的 return 语句将被执行而从方法返回。

运行结果如图 5-9 所示。

图 5-9　例 5-6 运行结果

例 5-7　ReturnDemo2.java，ReturnTest2.java

```
public class ReturnDemo2 {
    public void div(int a, int b) {

        double result = 0;

        if (b == 0) {

            System.out.println("出错：除数为 0!!! ");
            return;
        }

        result = a * 1.0 / b;
        System.out.println(a + "/" + b + " = " + result);
    }
}

import javax.swing.JOptionPane;

public class ReturnTest2 {

    public static void main(String[] args) {
        ReturnDemo2 ob = new ReturnDemo2();
```

```
            String strA = JOptionPane.showInputDialog("请输入被除数：");

            int a = Integer.parseInt(strA);

            String strB = JOptionPane.showInputDialog("请输入除数：");

            int b = Integer.parseInt(strB);

            ob.div(a, b);
        }
    }
```

除数为 0 不能进行除法运算，div()方法中将判断除数是否为 0，如果为 0，即 b==0 条件满足，进入 if 分支执行，输出出错提示后，利用 return 语句提前从方法返回，后续的除法运算将不被执行。

这里的 div()方法返回类型声明为 void 即无返回值，故 return 后不带表达式，直接分号结束语句即可。

当输入的除数非 0 时，b==0 的条件不成立，不会进入 if 分支提前从方法返回，后续的除法运算及结果输出才能被执行。

运行结果如图 5-10 所示：

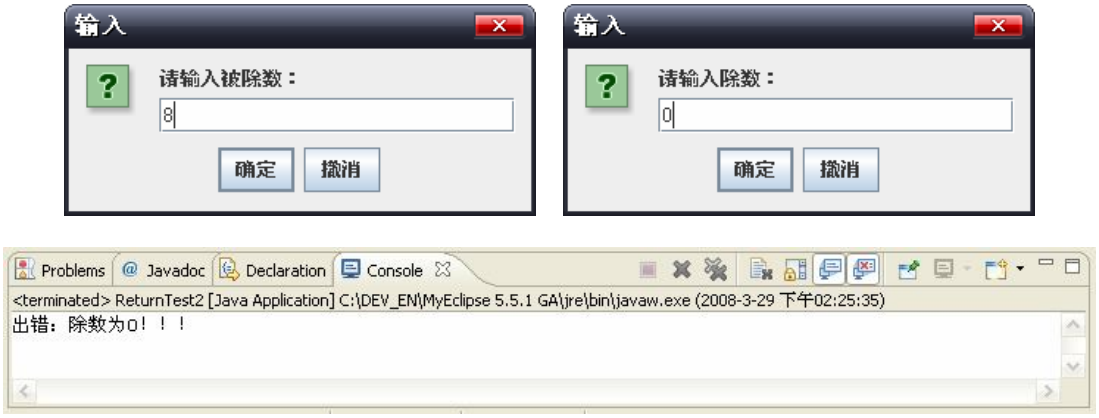

图 5-10　例 5-7 运行结果

关于方法的定义、调用和返回的补充说明：

- ✧ 调用类的成员方法，一般需要先创建类的对象（调用类的静态方法不用如此），再以对象名.方法名(实际参数列表)的格式调用。
- ✧ main()方法中可以创建自己所属的类自身的对象。

4. 方法的嵌套调用

如果将一个方法调用作为另一个方法的实际参数，就形成了方法的嵌套调用。

如果有形如：funa(funb(func(参数),其他参数),其他参数) 的多层嵌套调用，执行时将从处于最内层的方法调用开始，依次返回到上一层方法调用，即最先执行 func()方法，从 func()返回后，将返回值传给 funb()方法的形参开始执行 funb()，从 funb()返回后再将其返回

值传给 funa()的形参，开始执行 funa()。

5.3.2 从方法返回对象

方法的返回类型除了基本数据类型外，还可以是对象类型，即可以从方法返回对象。如下例所示。

例 5-8 Person.java,ReturnDemo3.java

```
public class Person {

    String name;

    public Person(String nm) {
        name = nm;
    }

    public String getName() {
        return name;
    }

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

    public Person friend(String nm){
        Person frd = new Person(nm);
        return frd;
    }

}
public class ReturnDemo3 {
    public static void main(String[] args) {
        Person tom = new Person("Tom");
        Person jerry = tom.friend("Jerry");
        System.out.println(tom.getName() +
                           "和" + jerry.getName() + "是好朋友");
    }
}
```

5.3.3 区别同名的局部变量与成员变量：this 引用

出现在类中各个不同位置的变量都是有其作用域的，在其作用范围内，这个变量才能使用。在相同的作用范围内，不能定义同名的变量。

在类体中、方法外声明的变量为类的成员变量，其作用域为整个类体。

方法的形式参数，以及方法体内部声明的变量，都属于局部变量，作用域仅为方法体，一旦方法调用结束，这些局部变量也将消失。

局部变量和成员变量定义的位置不同，如图 5-11 所示。

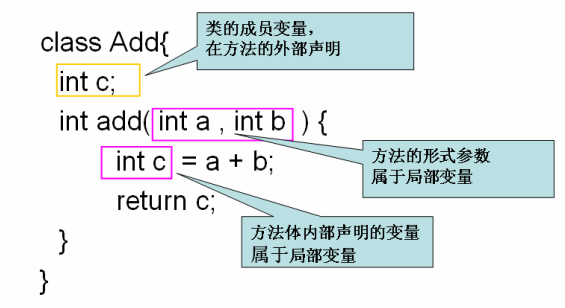

图 5-11　成员变量与局部变量

假设有如下的类定义：

```
public class SameName{
    int a;
    double a;
}
```

将不能通过编译，因为在相同的作用域内出现了重复的变量 a。

如果将上述类定义改为：

```
public class SameName {
    int a=10;
    public void method() {
        int a = 20;
        System.out.println("a=" + a);
    }
}
```

则可以通过编译，但在 method()方法中，成员变量 a 和局部变量 a 都是有效的，这种情况下，作用域范围小的局部变量 a 将屏蔽作用范围大的成员变量 a。

例 5-9　SameNameTest.java

```
public class SameNameTest {

    public static void main(String[] args) {
        SameName ob = new SameName();
        ob.method();
    }
}
```

运行结果为：

a=20

说明在 method()方法中有效的是局部变量 a。

如果希望在 method()方法中使用成员变量 a，可以使用关键字 this，该关键字代表对对象自身的引用。

在类体中，引用成员变量时可以加上 this 引用，即将 this.作为成员变量的前缀。那么在出现成员变量与局部变量同名的情况时，就可以区分了。使用了 this 引用的是成员变量，而无 this 引用的是局部变量。

例 5-10 SameName2.java，SameNameTest2.java

```
public class SameName2 {
    int a=10;
    public void method() {
        int a = 20;
        System.out.println("a=" + a);
        System.out.println("a=" + this.a);
    }
}

public class SameNameTest2 {

    public static void main(String[] args) {
        SameName2 ob = new SameName2();
        ob.method();
    }
}
```

运行结果为：

a=20

a=10

在成员变量与局部变量没有出现重名情况时，成员变量的 this 引用可以省略。this 引用还可以用在其他地方，如构造方法的重载，可以参考后续的章节。

5.4 构造方法

在第 3 章中简单介绍过，构造方法是类中的特殊方法，在创建类的对象时将会调用构造方法，构造方法一般对类的成员变量进行初始化操作。

构造方法不同于成员方法，从其定义形式上也可以体现出来：

- ✧ 构造方法名必须与所属的类名保持一致，包括大小写（Java 是大小写敏感的语言）。
- ✧ 构造方法无返回类型，连 void 都没有。
- ✧ 类中可以没有构造方法，这个时候系统会自动为类添加一个无参数且方法体为空的默认构造方法。形如：

```
public ClassName(){ }
```

前面的许多例题中都使用了系统缺省的无参构造方法来创建对象。

- ✧ 如果类中包含了构造方法，则用 new 调用构造方法创建对象时，必须按照构造方法定义的形式参数给出相应的实际参数；这时系统也不会自动添加无参数的构造方法了。

假设有如下类定义：

```
public class Abc{
    public Abc() {      //[1]
        ……
    }
```

```
    public void  Abc() {//[2]
        ……
    }
    public abc() {      //[3]
        ……
    }
}
```

只有注释[1]处的是构造方法；注释[2]处的方法虽与类名相同，但由于包含了返回类型的声明，故只是成员方法；注释[3]不能通过编译，名称与类名不同，故是成员方法，而成员方法是不能缺省返回类型的声明的。

构造方法的使用如下例所示：

例 5-11　Book.java，BookTest.java

```
public class Book {
    String bookName;
    String bookAuthor;
    String press;

    public Book(String bookName, String bookAuthor, String press) {
        this.bookName = bookName;
        this.bookAuthor = bookAuthor;
        this.press = press;
    }

    public String getBookAuthor() {
        return bookAuthor;
    }

    public void setBookAuthor(String bookAuthor) {
        this.bookAuthor = bookAuthor;
    }

    public String getBookName() {
        return bookName;
    }

    public void setBookName(String bookName) {
        this.bookName = bookName;
    }

    public String getPress() {
        return press;
    }

    public void setPress(String press) {
        this.press = press;
```

```
    }
}

public class BookTest {

    public static void main(String[] args) {
        Book one = new Book("Java2 实用教程", "耿祥义等",
                            "清华大学出版社");

        Book two = new Book("Java 面向对象编程", "孙卫琴",
                            "电子工业出版社");

        System.out.println("书目信息如下：");
        System.out.println("书名：" + one.getBookName()
                           + ",作者：" + one.getBookAuthor()
                           + ",出版社：" + one.getPress());
        System.out.println("书名：" + two.getBookName()
                           + ",作者：" + two.getBookAuthor()
                           + ",出版社：" + two.getPress());
    }
}
```

运行结果如图 5-12 所示。

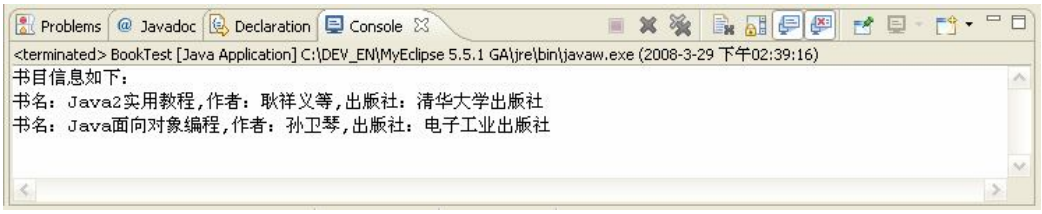

图 5-12　例 5-11 运行结果

Book 类的构造方法利用传入的实际参数给成员变量 bookName、bookAuthor 和 press 进行赋初值。创建 Book 的对象时，根据构造方法的参数要求给定书名、作者及出版社即可。

为提高可读性，构造方法的形式参数通常与成员变量同名，这时候要注意使用 this 引用表示成员变量，否则无法实现将参数值赋给成员变量而达到成员变量初始化的目的。读者可以将构造方法中各成员变量的 this 引用去掉，观察是不是如此。

5.5　类的静态成员

5.5.1　static 关键字

static 意为静态的，Java 中用 static 关键字来表示静态的成员。静态成员与非静态成员所处的存储空间不同，生命期也不一样。

类中有 static 修饰的变量和方法叫做类的静态变量、静态方法，统称为类的静态成员；而

无 static 修饰的则可相对的称为动态成员。

类的静态变量也称作类变量或域（field），无 static 修饰的成员变量也称作实例变量，无 static 修饰的成员方法也称作实例方法。

类的静态成员，不依赖于类的实例，在不创建类对象的情况下就可以直接通过类名来访问，并且这些静态成员被类的所有实例所共享。

类的静态成员的使用格式如下：

类名.静态变量名

类名.静态方法（参数列表）

1. static 变量

静态变量与实例变量的区别：

- ✧ Java 虚拟机只给静态变量分配一次内存，静态变量在内存中只有一个拷贝，任何类的实例对静态变量的修改都将有效。
- ✧ 实例变量依赖于类的实例，即具体的对象，每创建一个对象，就为该对象的实例变量分配一次内存，各个对象的实例变量占用不同的内存空间，互不干扰，对象对各自实例变量的修改不会影响到其他对象的实例变量。

在类的内部，可以在任何方法内部访问静态变量，在没有变量重名的情况下静态变量名前面可以不用带前缀。而在其他类中，可以通过类名来访问静态变量。

例 5-12　MyCircle.java，StaticTest1.java

```
public class MyCircle {
    public static double PI = 3.14;

    double radius;

    public MyCircle(double radius) {
        this.radius = radius;
    }

    public double perimeter() {

        return 2 * PI * radius;
    }

    public double area() {

        return PI * radius * radius;
    }
}

public class StaticTest1 {

    public static void main(String[] args) {

        MyCircle aCircle = new MyCircle(10);
```

```
        double perimeter1 = aCircle.perimeter();

        System.out.println("半径为 10 的圆周长为：" + perimeter1);

        double perimeter2 = 2 * MyCircle.PI * 5;    //[1]

        System.out.println("半径为 5 的圆周长为：" + perimeter2);
    }
}
```

在 StaticTest1 类中可以直接使用 MyCircle 类中定义的静态变量，使用时需要用类名来引用，如注释[1]处。

除用类名访问静态变量之外，也可以使用对象名来引用类的静态变量，就像引用实例变量那样，在类内部还可以使用 this 来引用静态变量。

类的静态变量主要有如下两个作用：

- ✧ 能被类的所有实例共享，可以作为实例之间共享的数据。
- ✧ 如果类的所有实例都需要一个相同的常量数据成员，可以把这个数据成员定义为静态的，从而节省内存空间。若要将变量的值固定成为常量，则应在变量的类型前加上 final 关键字，如上例中可以把 PI 声明为静态常量，避免错误的修改。

```
public static final double PI=3.14;
```

2. static *方法*

方法的返回类型前有 static 关键字修饰的成员方法即为类的静态方法。与静态变量类似，类的静态方法也不依赖于类的实例，不需要创建类的对象就可以通过类名来调用。

例 5-13　StaticTest2.java

```
public class StaticTest2 {

    public static void main(String[] args) {
        int sum1 = add(3, 7);   //[1]
        System.out.println("sum1 = " + sum1);

        int sum2 = StaticTest2.add(2, 4);   //[2]
        System.out.println("sum2 = " + sum2);
    }

    public static int add(int a, int b) {
        return a + b;
    }
}
```

在本类中调用静态方法可以不带前缀，如注释[1]；如果是在其他类中调用 StaticTest2 类的静态方法 add()，就必须按照注释[2]处的带类名的访问形式了。

5.5.2　成员与静态方法的关系

静态方法是随类加载的，只要类存在，静态方法就可以调用、执行。而实例变量、实例

方法这些非静态的成员都是依赖于类的实例的，必须在类对象存在的前提下才可以使用实例变量和实例方法。因此，静态方法体中不能使用无法确定是否存在的实例变量和实例方法。

例 5-14　StaticTest3.java

```
public class StaticTest3 {

    int a = 10;
    int b = 20;

    public static void main(String[] args) {
        int sum = a+b;
        System.out.println("sum = " + sum);
    }
}
```

类 StaticTest3 将会出现如图 5-13 所示的编译错误。

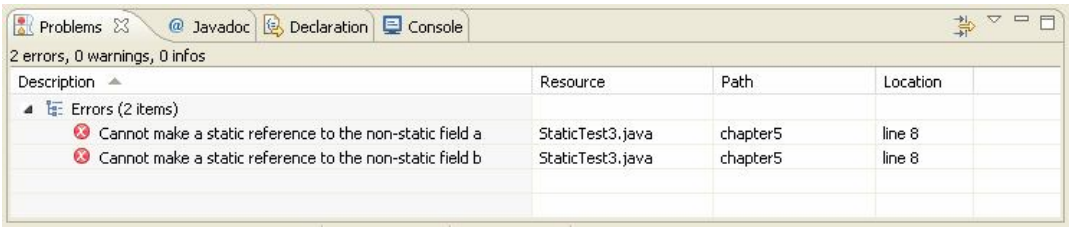

图 5-13　例 5-14 出错信息

该信息表示：不能在静态上下文中访问非静态的 a 和 b。

main()作为 Java 应用程序的执行入口，是静态方法，Java 虚拟机只要加载了 main()所属的类，就能执行 main()方法了，无须先创建类的对象，也因为这个原因，所以可以在 main()中利用 new 调用类的构造方法来创建本类的对象，事实上在其他的静态方法中也可以创建本类的对象。

在本例中，由于 a 和 b 均为非 static 的实例变量，它们在类的对象创建之后才会被分配内存而存在，在静态的 main()方法中不能对其进行引用。

解决上述访问问题的方法有两种：一种是将 a 和 b 改为 static 变量，则它们与 main()方法一样是静态成员，都随类加载，不依赖于对象的存在与否，在 main()中可以访问它们，在其他的静态方法中也可以访问。如下所示：

```
static int a = 10;
static int b = 20;
```

另一种解决方法是：静态的 main()方法中先创建类的对象，再用对象来访问这些实例变量。如下所示：

```
StaticTest3 ob = new StaticTest3();
int sum = ob.a + ob.b;
```

如果在静态方法中调用实例方法，又会出现什么情况呢，如下例所示。

例 5-15　StaticTest4.java

```
public class StaticTest4 {
    public static void main(String[] args) {
```

```
        int sub = sub(13, 5);

        System.out.println("sub = " + sub);
    }

    public int sub(int a, int b) {
        return a - b;
    }
}
```

编译信息如图 5-14 所示。

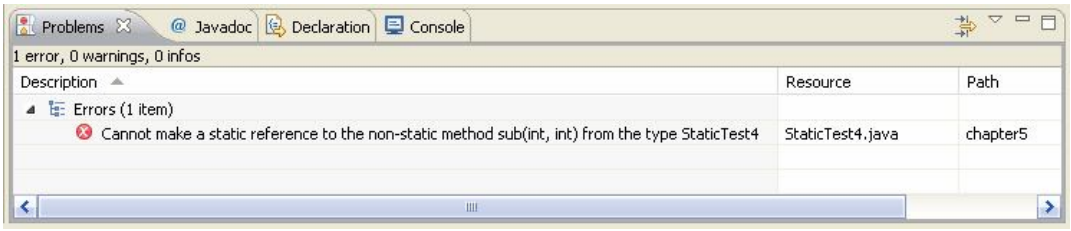

图 5-14　例 5-15 出错信息

同试图在静态方法中访问非静态的实例变量类似，此问题的出现也是因为非静态的实例方法依赖于类的实例，故在静态方法中不能调用实例方法。

解决方法有二：一种是将 sub()方法改为静态方法，如下所示：

```
public static int sub(int a, int b) {
        return a - b;
}
```

另一种解决办法是在静态的 main()中先创建类的对象，再调用实例方法，如下所示：

```
StaticTest4 ob = new StaticTest4();
int sub = ob.sub(13, 5);
```

反过来，没有 static 修饰的实例方法中没有这样的限制，实例方法可以访问类的静态变量、调用静态方法，也可以访问实例变量、调用实例方法。

5.6　表现多态：方法重载

面向对象编程具有多态特征，简单的说就是“对外一个接口，内部多种实现”。有时候，一种功能可能会有多种不同的实现方式。Java 支持方法重载（overload），可以在同一个类中定义多个名字相同但参数不同的方法。那么同一个方法名就是对外的统一接口，参数不同导致内部实现也不同，方法重载是面向对象编程多态特征的一种表现形式。

方法重载是编译时的多态，编译器在编译时刻确定具体调用哪个被重载的方法。

5.6.1　如何定义和调用重载的方法

在 Java 中，定义重载的方法必须遵循以下原则：

✧ 方法名相同，包括大小写。

✧ 方法的参数列表必须不同，也就是参数的类型、个数、顺序至少有一项不同。编译器将参数列表的不同作为重载的判定依据。

✧ 方法的返回类型、修饰符可以相同，也可以不同，它们不作为重载的判定依据。

例如，为不同的数据类型实现加法功能，可以定义重载的 add()方法，当给定不同类型的参数时，进行不同类型的运算。

例 5-16　AddOverload.java，OverloadTest.java

```
public class AddOverload {

    public int add(int a, int b) {
        return a + b;
    }

    public long add(long a, long b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }
}

public class OverloadTest {

    public static void main(String[] args) {

        AddOverload ob = new AddOverload();

        System.out.println("2 个 int 型数相加：" + ob.add(12, 34));
        System.out.println("2 个 long 型数相加：" + ob.add(123L, 456L));
        System.out.println("2 个 double 型数相加：" + ob.add(1.2, 3.4));
    }
}
```

当重载的方法被调用时，Java 编译器将根据实际参数的类型、个数和顺序来确定调用哪个重载方法的版本。如上例，假如将“System.out.println("2 个 long 型数相加：" + ob.add(123L, 456L));”中 add()方法的两个参数的 L 去掉，就不会调用 public long add(long a, long b)方法，而是调用 public int add(int a, int b)方法了。

5.6.2　构造方法的重载

构造方法也可以重载，这样在创建对象时可以调用不同版本的构造方法来进行初始化操作。在构造方法中可以使用 this 关键字调用其他版本的构造方法，减少重复编码。

用 this 关键字调用构造方法的格式如下：

```
this(参数列表)
```

例 5-17 ContactPerson.java，OverloadTest2.java

```
public class ContactPerson {
    String name;
    String selfphone;
    String email;

    // 只知联系人姓名
    public ContactPerson(String name) {
        this.name = name;
    }

    // 知道姓名和电话
    public ContactPerson(String name, String selfphone) {

        this.name = name;        //[1]
        this.selfphone = selfphone;
    }

    // 知道姓名、电话和email
    public ContactPerson(String name, String selfphone, String email) {
        this(name, selfphone);       //[2]
        this.email = email;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSelfphone() {
        return selfphone;
    }
    public void setSelfphone(String selfphone) {
        this.selfphone = selfphone;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}

public class OverloadTest2 {
```

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

        ContactPerson no1 = new ContactPerson("Tom");
        System.out.println("只知姓名的联系人：" + no1.getName());

        ContactPerson no2 = new ContactPerson("Jerry",
                              "13812345678");
        System.out.println("知道姓名、电话的联系人："
             + no2.getName() + "  " + no2.getSelfphone());

        ContactPerson no3 = new ContactPerson("Jane",
                             "13887654321", "jane@abc.com");
System.out.println("知道姓名、电话和email的联系人：" +
                   no3.getName() + "  " + no3.getSelfphone()
                   + "  " + no3.getEmail());
    }
}
```

运行结果如图 5-15 所示。

图 5-15　例 5-17 运行结果

ContactPerson 类根据联系人信息的完整程度，定义了 3 个版本的构造方法以适应需求。第 3 个版本的构造方法中，利用 this 调用了第 2 个版本的构造方法：

```
public ContactPerson(String name, String selfphone)
```

该构造方法实现了给 name 和 selfphone 初始化并直接调用，在第 3 个版本的构造方法中就不必重复书写初始化语句了。事实上，注释[1]处也可以换成 this 调用构造方法 this(name)。

在 OverloadTest2 中，根据联系人信息的完整程度不同，分别调用了不同版本的构造方法创建了联系人对象，具体调用哪个版本的构造方法由调用时给定的实际参数列表与形参列表匹配而定。

需要注意的是：

- ✧ 重载构造方法时，若要使用 this 调用其他版本的构造方法，则该 this 调用语句必须作为构造方法的方法体中的第一条语句。

5.7　继承

继承是实现代码重用的一种有力手段。如果有多个类具有一些相同的属性和方法，可以将这些相同的属性和方法抽象出来定义为一个父类，在父类中定义这些属性和方法，则具有这

些属性和方法的其他类可以从这个父类派生出来，不必再重新定义这些属性和方法，我们称这些派生而来的类为子类。

5.7.1 类的继承

类的继承，利用已有的类来创建新类，在父类中定义过的属性和方法，子类中可以不用重新定义，可实现代码重用，降低编码和维护的工作量。

在继承关系中，被继承的类称为父类 、超类或基类，其中定义了子类所共有的属性和方法。由继承方式创建的类称为子类，子类将父类的属性和方法继承过来，不必重新定义，并且可根据需要增加新的属性和方法。

Java 的所有类都直接或间接地继承自 java.lang.Object 类。类定义没有用 extends 继承某个类时，实际上隐式继承了 Object 类。

类的继承示例如图 5-16 所示。

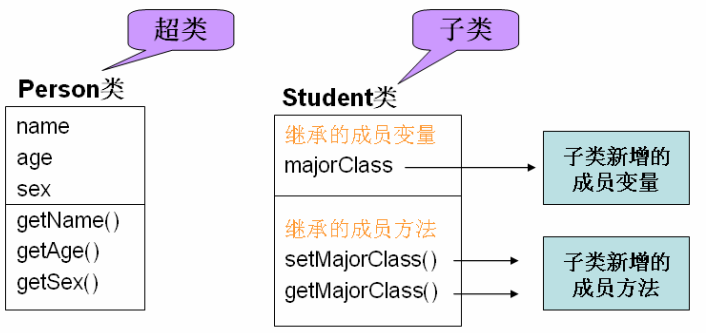

图 5-16 类的继承

Java 中类的继承用关键字 extends 实现，用继承来定义一个新类的格式如下所示：

```
class SonClass extends SuperClass {
    ……
}
```

上述代码表示 SonClass 类继承了 SuperClass 类，将继承 SuperClass 类中所有的非 private 的成员变量和成员方法。private 是访问控制符，被 private 修饰的变量和方法都只能在本类中访问（参见 5.9 节）。

与 C++不同的是，Java 是单继承的，一个子类只能有一个直接的父类，如果出现类似如下的类定义，编译时将会报错：

```
class SonClass extends SuperClass1,SuperClass2,SuperClass3,…{
    ……
}
```

下面是一个继承的示例，定义一个描述人的共有信息的父类，社会关系中各种各样不同的人就可以从这个类继承而来，再添加上能表现自己独特性的属性和方法即可。

为与前面的例子中的类区分开来，这里将描述人共有信息的父类命名为 Person2，由 Person2 派生而来的学生子类命名为 Student2。

例 5-18　Person2.java，Student2.java2

```
public class Person2 {
    String name;
    String sex;
    int age;

    public Person2(String name, String sex, int age) {
        this.name = name;
        this.sex = sex;
        this.age = age;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSex() {
        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    public void display() {
        System.out.println("This person's information : \n");

        System.out.println("Name : " + name);
        System.out.println("Sex : " + sex);
        System.out.println("Age : " + age);
    }
}

public class Student2 extends Person2{

    String majorClass;  //新增属性：专业班级

    public Student2(String name, String sex, int age,
String majorClass){
```

```
        super(name,sex,age);  //调用父类的构造方法

        this.majorClass = majorClass;
    }

    //新增方法
    public String getMajorClass(){
        return majorClass;
    }
    public void setMajorClass(String newMajorClass){
        majorClass = newMajorClass;
    }

    //重写父类的display方法
    public void display(){
        System.out.println("This Student's information : \n");

        System.out.println("Name : " + name);
        System.out.println("Sex : " + sex);
        System.out.println("Age : " + age);

        System.out.println("Major Class : " + majorClass);
    }
}
```

在子类 Student2 中，虽然只有 1 个成员变量的定义，但事实上包含 4 个成员变量，其中 3 个从父类 Person2 继承而来。构造方法实现给所有 4 个成员变量进行初始化操作，其中 **super**(name,sex,age)的作用是调用父类 Person2 的构造方法，实现对从父类继承来的 3 个变量进行初始化操作。

成员方法也是如此，除了子类特有的新增成员变量 majorClass 的 get 和 set 方法外，还有其他 3 个变量的 get 和 set 方法从父类继承而来。

父类中包含显示一个大众化的 person 信息的 display()方法，而对于学生而言，需要显示的信息还要包括代表学生特征的部分，即 majorClass，则父类的 display()方法对于子类不适用，但方法的功能是一样的，这时候子类可以定义一个跟父类同名的方法，返回类型、参数列表都相同，这样子类方法就覆盖（或叫重写）了父类的方法，子类对象调用 display()方法时调用的就是子类的版本了。

如下例所示，创建子类 Student2 的对象，并显示学生的相关信息。

例 5-19　Student2Test.java

```
public class Student2Test {
    public static void main(String[] arg) {
        Student2 jane = new Student2("Jane", "female",
                                     19, "Computer");
        jane.display();   //调用的是子类重写的方法
    }
}
```

运行结果如图 5-17 所示。

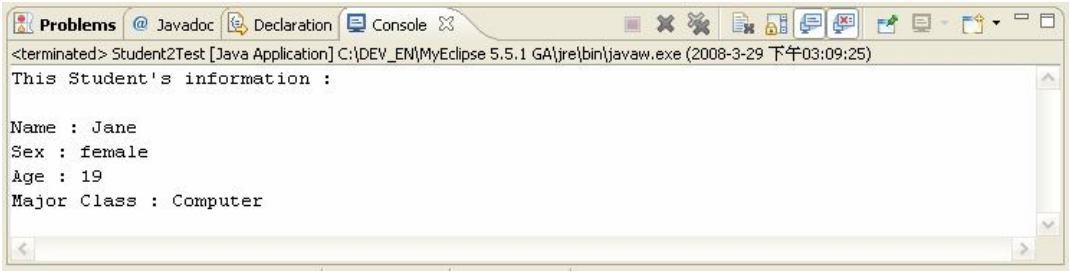

图 5-17　例 5-19 运行结果

由运行结果可知，子类对象调用的 display()方法是在子类中重写的 display()方法。

5.7.2　super 关键字

我们知道，如果类的成员变量与局部变量重名，类的成员变量将被隐藏，如果要使用类的成员变量需要使用 this 引用之。

在继承关系中，也存在类似的问题：

- ✧ 若子类中定义了与父类同名的成员变量，则父类的成员变量被隐藏。
- ✧ 若子类的方法中定义了与父类成员变量同名的局部变量，则父类的成员变量被隐藏。
- ✧ 若子类中定义了与父类相同的成员方法（同方法名，同参数列表，同返回类型），则父类方法被覆盖，在子类范围内，父类方法不可见。

解决继承关系中类变量或方法不可见的问题，需要使用关键字 super。

顾名思义，super 可以用来引用继承自父类的成员。super 的使用有如下几种形式：

- ➢ super.变量名：引用父类成员变量。
- ➢ super.方法名（参数列表）：调用父类成员方法。
- ➢ super(参数列表)：调用父类构造方法，在子类构造方法中调用父类的构造方法以实现对继承自父类的成员变量的初始化，如 Student2 类构造方法中的 super(name,sex,age)。同 this(参数列表)调用本类构造方法一样，super(参数列表)调用也应出现在构造方法体的第一条语句处。

在下例中，父类 SuperClass 和子类 SubClass 都包含名为 data 的成员变量和成员方法 method()。在子类 SubClass 中可以通过 super 关键字访问到父类 SuperClass 中的成员变量 data 和成员方法 method()。

例 5-20　SuperClass.java，SubClass.java

```
public class SuperClass {

    String data = "父类的成员变量";

    public void method() {
        System.out.println("正调用父类的方法 method()……");
    }
```

```
}

public class SubClass extends SuperClass {

    String data = "子类的同名变量"; // 隐藏了父类同名变量 data

    public void method() { // 覆盖了父类方法 method()
        System.out.println("正调用子类 SubClass 的方法 method()……");
    }

    public void method2() {

        String data = "子类的局部变量";
        // 子类局部变量隐藏了父类同名变量
        // 同时也隐藏了本子类的同名成员变量

        System.out.println("data is : " + data);

        System.out.println("this.data is : " + this.data);

        System.out.println("super.data is : " + super.data);

        System.out.print("直接调用 method() : ");
        method();

        System.out.print("this.method() : ");
        this.method();

        System.out.print("super.method() : ");
        super.method();
    }

    public static void main(String[] args) {

        SubClass ob = new SubClass();
        ob.method2();
    }
}
```

运行结果如图 5-18 所示。

图 5-18　例 5-20 运行结果

观察运行结果，体会 this 与 super 的作用。

5.7.3　继承中的 final 修饰符

继承和方法重写虽然应用广泛，但有时候也可能不希望从类派生子类，或不希望类中的方法会被重写。比如，出于安全考虑，类的实现细节不允许被改动，或者不允许子类覆盖父类的某个方法，这时候就可以对类或成员方法使用 final 修饰符。

final 有不可改变的含义，可以用于修饰类、成员方法以及成员变量。

（1）定义类时，在 class 关键字前可以加上 final 修饰符，则这个类将不能再派生子类，即不能被其他类所继承。例如：

```
public final class A{
    ……
}
```

则类 A 不可被继承。

（2）声明类的成员方法时，在返回类型前可以加上 final 修饰符，则方法所属的类被继承时，这个方法不会被重写。例如：

```
public final void method(){
    ……
}
```

（3）final 修饰成员变量与继承无关，它表示的是变量一经赋值，其值将不能改变，也就是通过 final 可以定义常量，常量名一般全部大写。例如：

```
final int MAX_NUM=100;
```

如果程序中试图修改由 final 修饰的 MAX_NUM 的值，将会产生编译错误。

5.8　抽象类与接口

5.8.1　抽象类与抽象方法

Java 中可以用 abstract 修饰符修饰类和成员方法。

- 用 abstract 修饰的类为抽象类。在类的继承体系中，抽象类常位于顶层。抽象类不能被实例化，即不能创建抽象类的对象。
- 用 abstract 修饰的方法为抽象方法，抽象方法没有方法体，一般用来描述具有什么功能，而不提供具体的实现。

例如以下代码中 AbstractClass 为抽象类，它包含一个抽象方法 method1()和一个具体方法（非抽象方法）method2()。

```
public abstract class AbstractClass{
    public abstract void method(); //抽象方法，无方法体

    public void method2(){      //具体方法，即使方法体为空，{}也不能省略
        ……
    }
}
```

抽象类和抽象方法有一些注意事项：

✧ 抽象类中可以包含构造方法、成员变量、具体方法、甚至可以没有抽象方法，但包含了抽象方法的类必须定义为抽象类，否则编译出错。例如下述代码中，Son 类继承 Father 类，重写了抽象方法 m1()，为其提供了具体实现，而没有提供抽象方法 m2() 的实现，则 Son 类中包含抽象方法 m2()，因此 Son 类必须被声明为 abstract 的，否则编译报错。

```
abstract class Father{
    abstract void m1();
    abstract void m2();
}
class Son extends Father{   //包含 abstract 的 m2()，编译出错
    void m1(){  //即使方法体为空，也是具体方法
    }
}
```

✧ 没有抽象的构造方法，也没有静态的抽象方法，即 static 和 abstract 不能一起修饰方法。

✧ 抽象类与抽象方法不能被 final 修饰。抽象类存在的意义就是为了被继承，抽象方法存在的意义也是为了被重写（或称作被实现：抽象方法无方法体，重写抽象方法是对抽象方法的具体实现），而 final 所代表的含义正与其相反，final 修饰类和成员方法分别代表类不可被继承和方法不可被重写，abstract 与 final 是自相矛盾的，故不可一起使用。

✧ 抽象类不可实例化。例如，老虎、猴子、狮子都是具体类，自然界有它们的实例，动物类是它们的父类，是抽象类，自然界并不存在动物类本身的实例。

5.8.2 接口

Java 中的接口（interface）使抽象类的概念更深入一层。外界观察一个对象，主要关注对象提供了什么服务，至于服务在对象内部是如何具体实现的，外部并不关心。对象所提供的服务由方法实现，因此，对象中所有外界能使用的方法的集合，就构成了对象与外界进行交互的“界面”，即接口。

在语法上，接口类似于抽象类，但比抽象类更抽象，接口中只声明方法，但不定义方法体，不能包含具体方法。接口只声明能做什么，但不声明怎么做，怎么做将由实现接口的类来确定。可以认为接口就是一个行为的协议或规范，实现一个接口的类将具有接口规定的行为，并提供具体实现。

接口的定义格式：

```
[public] interface InterfaceName [extends SuperInterface]{
    //接口体
}
```

其中，接口体与类体类似，也包含成员变量和成员方法，但有一些限制：

✧ 接口中的成员变量默认都是 public、static、final 类型的，即都是静态常量，必须显式地进行初始化。

✧　接口中的方法默认都是 public、abstract 类型的，即都是抽象方法，无方法体，不提供具体实现。

以下代码就是一个合法的接口定义：

```
public interface  Computable {
    public static final double PI = 3.1415926;
    public abstract double sum( double  x, double y);
    public double sub(double x, double y);
    //即使缺省 abstract，系统会自动添加
}
```

✧　接口没有构造方法，不能创建接口的对象。

类声明时可以使用 implements 子句来表示实现某个或某些接口。接口声明了一些行为规范，实现接口的类将具有这些行为规范，但必须提供所有行为的实现细节，即实现接口的类必须重写接口中的所有抽象方法。

如果一个类实现多个接口，则各个接口之间需用逗号“,”分隔。

类实现接口的格式如下：

```
class  类名  implements  接口名 [, 接口 2, ……] {
    // 各个方法的具体实现
}
```

下面的代码是一个接口及类实现接口的简单例子：

```
public interface AnimalCry{
   public void cry();
}
class Dog implements AnimalCry {
   public void cry(){
         System.out.println(“汪汪汪…”);
   }
}
class Cat implements AnimalCry {
   public void cry(){
         System.out.println(“喵喵喵…”);
   }
}
```

动物都能发出叫声，在 AnimalCry 接口中定义该行为规范 cry()，但不同的动物叫声不同，Dog 和 Cat 对这个行为规范有不同的具体实现。

下面是一个接口及其实现类的示例。类实现接口时，需要实现接口的所有抽象方法，将行为具体化，类可以使用接口中定义的常量，事实上类实现接口也是一种继承，故类继承了接口中的常量。

例 5-21　MyInterface1.java，MyCrl.java，MyRect.java，InterfaceTest.java

```
public interface MyInterface1 {
    public static final double PI = 3.14;

    public double area();
```

```
    public double length();
}

public class MyCrl implements MyInterface1 {

    double radius;

    public MyCrl(double radius) {
        this.radius = radius;
    }
    public double area() {
        return PI * radius * radius;
    }
    public double length() {
        return 2 * PI * radius;
    }
}

public class MyRect implements MyInterface1 {

    double width;
    double height;

    public MyRect(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public double area() {
        return width * height;
    }
    public double length() {
        return 2 * (width + height);
    }
}

public class InterfaceTest1 {

    public static void main(String[] args) {

        MyCrl circle = new MyCrl(5);
        MyRect rect = new MyRect(10, 6);

        System.out.println("圆面积：" + circle.area());
        System.out.println("圆周长：" + circle.length());
        System.out.println("矩形面积：" + rect.area());
        System.out.println("矩形周长：" + rect.length());
    }
}
```

运行结果如图 5-19 所示。

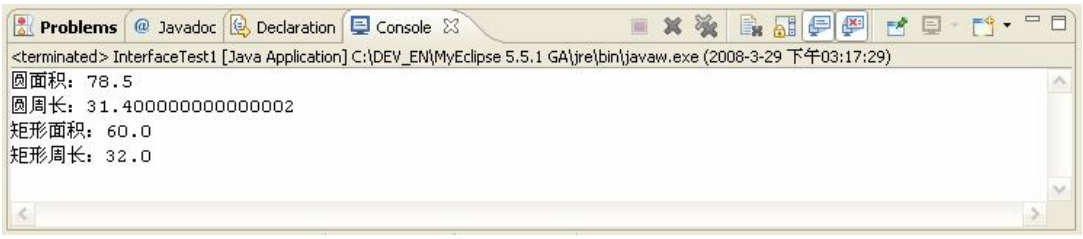

图 5-19　例 5-21 运行结果

5.8.3　接口实现多继承效果

Java 是单继承的，一个类只能有一个直接父类，但利用接口，可以达到多继承的效果。因为一个类可以实现多个接口，就继承了所实现的所有接口中的静态常量，以及所有的抽象方法，若实现这些接口的类不是抽象类，则这个类要提供所有这些方法的实现。

设有如下的类定义：

```
class A implements B,C,D {
    ……
    //必须重写接口 B、C、D 中的抽象方法
}
```

则类 A 继承了接口 B、C、D 的所有静态常量和所有方法，但在类 A 中必须重写这些方法，具体实现接口的行为。

下面是一个类实现多个接口从而实现多继承效果的示例。

例 5-22　Inter_Area_Volumn.java，Inter_Color.java，MyCrl2.java，InterfaceTest2.java

```
public interface Inter_Area_Volumn {
    public static final double PI = 3.14159;

    public abstract double area();
    public abstract double volume();

}

public interface Inter_Color {
    public abstract void setColor(String color);
}

class MyCrl2 implements Inter_Area_Volumn, Inter_Color {

    double radius;
    String color;

    public MyCrl2(double radius) {
        this.radius = radius;
    }
```

```
    // 实现接口 Inter_Area_Volumn 的方法
    public double area() {
        return PI * radius * radius;
    }
    public double volume() {
        return 4 * PI * radius * radius * radius / 3;
    }

    // 实现接口 Inter_Color 的方法
    public void setColor(String color) {
        this.color = color;
    }

    String getColor() {
        return color;
    }
}

public class InterfaceTest2 {
    public static void main(String args[]) {

        MyCrl2 c = new MyCrl2(4);

        System.out.println("Circle area = " + c.area());
        System.out.println("Circle volumn = " + c.volume());

        c.setColor("RED");
        System.out.println("Circle colore = " + c.getColor());
    }
}
```

运行结果如图 5-20 所示。

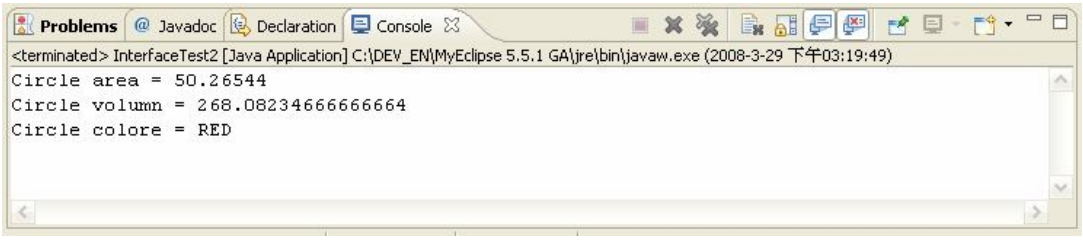

图 5-20　例 5-22 运行结果

创建类时，还可以让这个类既继承某个类，又实现某个或某些接口，让这个类具备更多的属性和行为，达到多继承的效果。

设有如下的类定义：

```
class A extends B implements C,D{
```

```
    ……
    //必须重写接口 C、D 中的方法
}
```

则类 A 将继承类 B 的非 private 的成员变量、成员方法，继承接口 C、D 的静态常量，并要实现接口 C、D 的所有方法。

如下示例仍使用上例中的两个接口 Inter_Area_Volumn 和 Inter_Color，新增类 Display，MyCrl3 类将继承类 Display，并实现两个接口。

例 5-23　Display.java，MyCrl3.java，InterfaceTest3.java

```
public class Display {
    public void heading() {
        System.out.println("以下是这个图形的基本信息： ");
    }
}

class MyCrl3 extends Display implements Inter_Area_Volumn,Inter_Color {

    double radius;
    String color;

    public MyCrl3(double radius){
      this.radius = radius;
    }
    //实现接口 Area_Volumn 的方法
    public double area(){
       return PI*radius*radius;
    }
    public double volume(){
       return 4*PI*radius*radius*radius/3;
    }
    //实现接口 Colour 的方法
    public void setColor(String color){
        this.color = color;
    }
    String getColor(){
        return color;
    }
}

public class InterfaceTest3 {

    public static void main(String args[]) {

        MyCrl3 c = new MyCrl3(2);

        c.heading(); // 继承自类 Display 的方法
```

```
        System.out.println("Circle area = " + c.area());
        System.out.println("Circle vloume = " + c.volume());

        c.setColor("Red");
        System.out.println("Circle colore = " + c.getColor());
    }
}
```

接口还可以继承接口，而且与类继承类不同的是，一个接口可以利用 extends 继承多个接口，也是多继承效果的一种体现。

设有如下的接口定义：

```
interface  A  extends  B,C{
    //接口 A 的静态常量和抽象方法
}
```

则 B、C 必须都是接口，接口 A 继承了接口 B、C 的所有静态常量和抽象方法，但并不实现任何的方法。

已有如上所示的接口 A，那么如下的类定义：

```
class ClassA implements A{
    ……
    //实现 A 的所有抽象方法
 }
```

类 ClassA 实现接口 A 就要实现其所有的抽象方法，这些方法除了接口 A 自身包含的抽象方法之外，还包括接口 A 从其父接口 B、C 继承而来的抽象方法。

5.9 包与访问控制修饰符

5.9.1 包的概念与作用

Java 中，包（package）是相关类与接口的一个集合，是一种管理和组织类（包括类和接口）的机制。

简单来看，包就相当于是一个目录，其中可以包含类、接口、子包（相当于子目录）。其作用包括：一是能减少类的名称的冲突问题，二是能分门别类地组织各种类，三是有助于实施访问权限控制，当位于不同包的类相互访问时，会受到访问权限的约束。

典型的，Java 的类库就是按有层次的包的方式组织的，如图 5-21 所示。

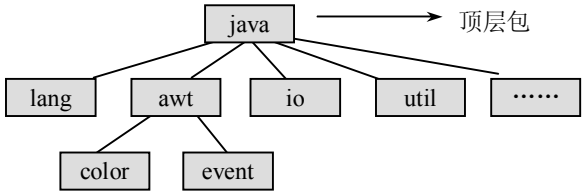

图 5-21 包层次图

其中，Java 的一些基本包如下所示：

- java.lang 包：核心语言包，包含 System 类（系统类）、String 类（字符串类）、Exception 类（异常类）等，这些类是编写 Java 程序经常要使用的。这个包由 JVM 自动引入，在编写程序时可以直接使用这个包中的类，而不必用 import 语句引入。
- java.awt 包：抽象窗口工具集包，包含了用于构建图形用户界面（GUI）程序的基本类和绘图类。
- java.io 包：输入/输出包，包含各种输入流类和输出流类，用于实现程序与外界的数据交换。
- java.util 包：使用工具包，提供一些实用类，如 Date 类（日期类）、Random 类（随机数类）、Collection 类（集合类）等。

包的表示法以圆点“.”作为分隔符，如 System 类的包表示法为 java.lang.System。包的结构映射到操作系统中时就是目录结构，System 类对应的.class 文件就在 java\lang\的目录结构下。

5.9.2　使用包

到目前为止，所有的示例程序都没有使用包，而事实上，当编写比较大的 Java 项目时，类的数量不断增加，会生成许多的.class 文件，为了有效地对这些类及其.class 文件进行组织和管理，应该使用包。

1. 包声明语句 package

包声明语句用于将 Java 的类放到特定的包中，对应地，类的.class 文件被组织到包结构映射而来的目录结构中。

Java 利用 package 关键字声明包，格式如下：

```
package packageName;
```

例如：

```
package cn.whvcse;
public class School{
    ……
}
```

则 School 类被置于名为 cn.whvcse 的包中，编译产生的类文件 School.class 将被置于 cn\whvcse\目录下，类的完整名称为 cn.whvcse.School。

使用包声明语句时，需要注意如下事项：

- ✧ 包声明语句（package 语句）必须出现在 Java 源文件的第一行（忽略注释行）。
- ✧ 如果有包声明语句的 Java 源文件中包含了多个类或接口的定义，则这些类和接口都将位于声明的这个包中。
- ✧ 一个 Java 源文件只能包含一个 package 语句。
- ✧ 如果 Java 文件中没有 package 语句，则这个文件中的类位于默认包，默认包没有名字。

例 5-24　PackageDemo.java

```
package cn.whvcse;
public class PackageDemo {
    public void hi() {
```

```
        System.out.println("Hi~~~~");
    }
}
```

在名为 chapter5 的项目中创建该类，编译后产生的.class 文件被组织到项目根目录的 cn\whvcse\目录下，如图 5-22 所示。

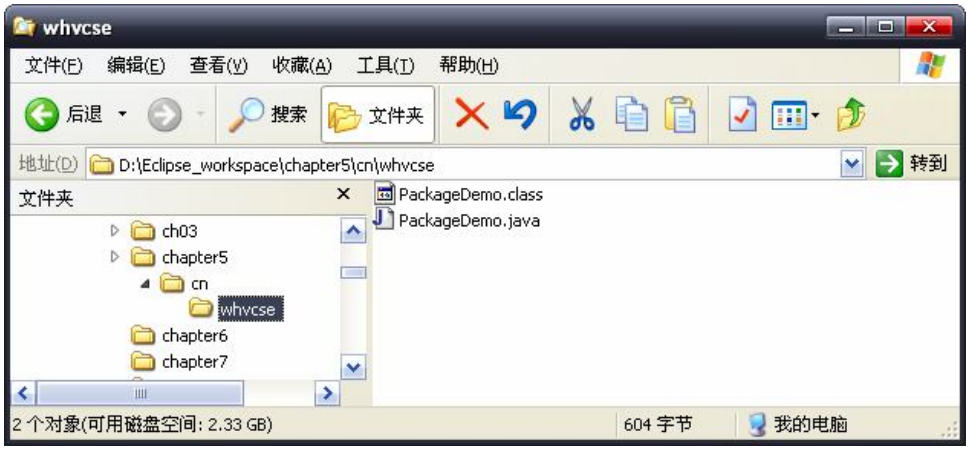

图 5-22　例 5-24 的效果图

2. 包引入语句 import

位于同一个包（即同一个目录）中的类可以直接相互访问，不需要做额外操作。但如果一个类要访问来自于另外一个包中的类，则需要通过 import 语句将其需要访问的类引入（java.lang 包中的类除外），否则无法使用其他包中的类，编译时会报错。

比如，前面介绍过的实现简单输入/输出对话框的 JOptionPane 类，在使用该类时，我们总是在类定义的前面写上这样一条语句：

```
import javax.swing.JOptionPane;
```

Java 使用 import 关键字来引入类，格式如下：

```
import 完整类名;
```

注意：

✧ 此处的类名要使用包表示法。

✧ import 语句要位于 package 语句之后，类或接口定义之前（忽略注释）。

例 5-25　PackageDemo2.java

```
package cn.xyy;

import cn.whvcse.PackageDemo;

public class PackageDemo2 {
    public static void main(String[] args) {
        PackageDemo ob = new PackageDemo();
        ob.hi();
    }
}
```

运行结果如图 5-23 所示。

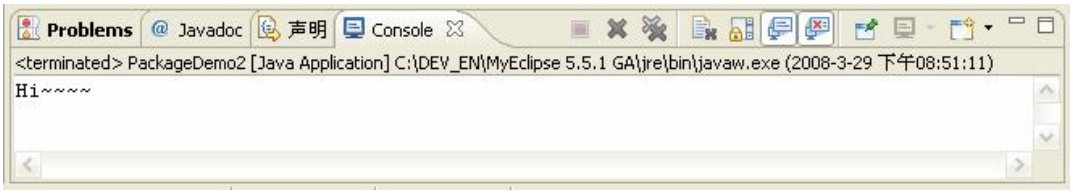

图 5-23　例 5-25 运行结果

PackageDemo2 类位于 cn.xyy 包，若希望访问 cn.whvcse 包中的 PackageDemo 类，必须在类定义前引入这个类。

尝试将上述示例中的 import 语句去掉，将出现如图 5-24 所示的编译错误。

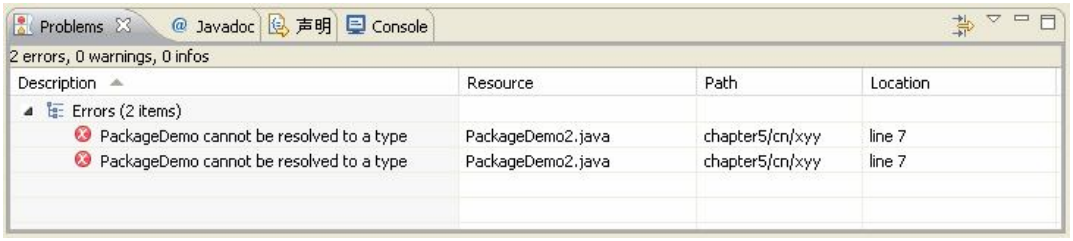

图 5-24　例 5-25 出错信息

此错误信息表示在 PackageDemo2 类中无法解析 PackageDemo 类，这是没有引入类造成的。

有时候，可能需要访问某个包中的多个类，如果不想一个一个引入的话，可以引入包中所有的类，用*替换处在包名最末尾的类名即可，如：

```
import cn.whvcse.*;
```

就表示引入 cn.whvcse 包中所有的类。

5.9.3　访问控制符

访问控制符可以对被其修饰的元素进行访问权限控制，这种控制也与包相关。类中成员的可见性（或叫可访问性）取决于它的访问控制符和它所在的包和类的性质。

Java 的访问控制符有 4 种：public 、protected、default（默认）和 private。其中，default 默认访问控制符，指的是不添加任何访问控制的关键字。访问控制符可以修饰类，也可以修饰类的成员，访问控制符应放在类、变量或方法声明的最前面。

图 5-25 和表 5-1 总结了访问控制符对类及类的成员的可访问性的影响。

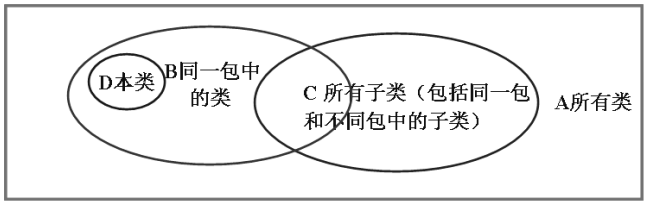

图 5-25　可访问范围示意图

表 5-1 访问控制符的作用

访问控制符	修饰的元素	可访问范围
public	类、变量、方法	A
Protected	变量、方法	B + C
default（默认）	类、变量、方法	B
Private	变量、方法	D

1. private

private 意为私有的，只能修饰类的成员，不能修饰类，由 private 修饰的成员只能在类的内部访问。一般可以对希望隐藏的内部数据或内部方法添加 private 关键字，以保护敏感信息或隐藏功能的实现细节。

例 5-26 PriavateDemo1.java，PrivateTest1.java

```
package cn.whvcse;
public class PrivateDemo1 {

    private int price = 30;
    private int num = 5;

    private int total() {
        return price * num;
    }
    int getNum() {
        return num;
    }
    void setNum(int num) {
        this.num = num;
    }
    int getTotal() {
        return total();
    }
}

package cn.whvcse;
public class PrivateTest1 {

    public static void main(String[] args) {
        PrivateDemo1 ob = new PrivateDemo1();

        // ob.price = 40; //无法通过编译
        // ob.num = 10;     //无法通过编译
        // int total = ob.total(); //无法通过编译

        int total = ob.getTotal();
```

```
        System.out.println("总价为：" + total);

        ob.setNum(10);
        //PrivateDemo1 提供了 default 的 setNum()方法
        //与 PrivateDemo1 在同一个包中的 PrivateTest1 中可以访问到该方法

        System.out.println("修改数量后的总价为：" + ob.getTotal());
    }
}
```

运行结果如图 5-26 所示。

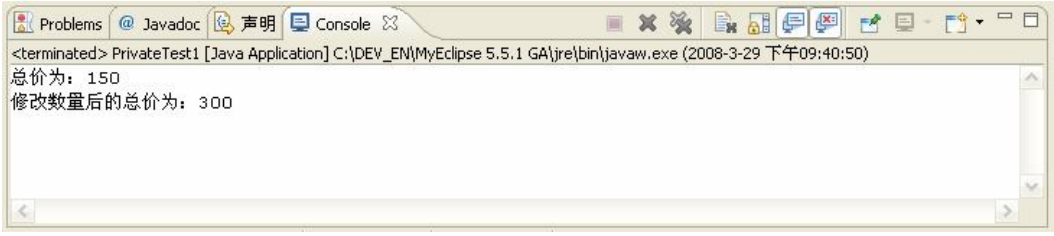

图 5-26　例 5-26 运行结果

PrivateDemo1 类的成员 price、num 和 total()方法都被 private 修饰，因此它们只在本类中可见，也就是只能在本类的方法中访问，在 PrivateTest1 类中是无法访问的。

假如将 PrivateTest1 类中用注释隐藏起来的 3 条可执行语句恢复出来，会得到如图 5-27 所示的编译出错信息。

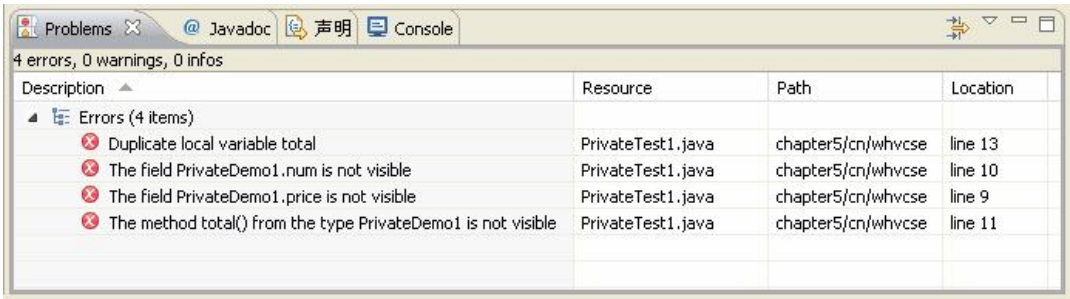

图 5-27　例 5-26 出错信息

观察第 2、3、4 个错误，num、price 和 total()都是无法访问的，因为它们是 private 成员。第 1 个错误是“int total = ob.total();”恢复出来后，有两个重复的 total 变量造成的。

2. default

default 是默认访问控制符，既可以修饰类的成员，也可以修饰类，由 default 修饰的元素都是同一个包中的类可以访问。

由 default 修饰类的成员，见上述的例 5-26 中的 getNum()、setNum()和 getTotal()。

对于 num，虽然自身是 private 的，但提供了一个 default 的修改方法 setNum()，PrivateDemo1.java 与 PrivateTest1.java 又是位于同一个包 cn.whvcse 中的，因此在 PrivateTest1 中可以调用 setNum()方法，从而实现修改 num 的值。

默认访问控制符的类的可访问性见下面的示例。

例 5-27 DefaultClassDemo.java，DefaultClassTest.java

```
package cn.whvcse;

class DefaultClassDemo {

    public void hi() {

        System.out.println("Hi~~~,in DefaultClassDemo");
    }
}

package cn.xyy;

import cn.whvcse.*;

public class DefaultClassTest {

    public static void main(String[] args) {

        DefaultClassDemo ob = new DefaultClassDemo();

    }
}
```

在 DefaultClassTest 类中试图访问 DefaultClassDemo 类，即使用 import 语句将 DefaultClassDemo 类所在的包 cn.whvcse 中的所有类都引入了，仍然出现如图 5-28 所示的错误信息。

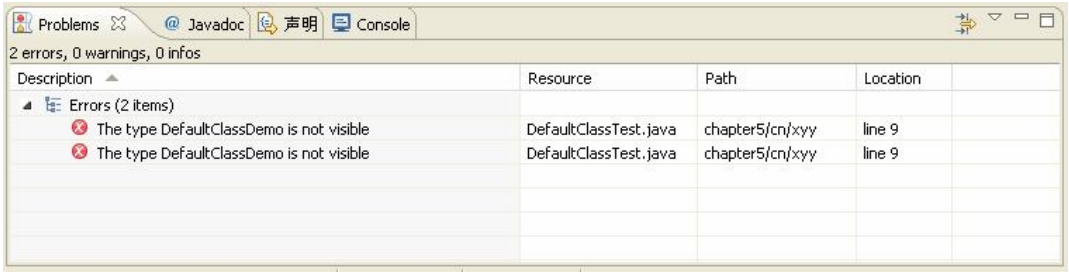

图 5-28 例 5-27 错误信息

错误信息表示 DefaultClassDemo 类是不可见的，这是因为位于 cn.whvcse 包的 DefaultClassDemo 类是 default 的，只能被 cn.whvcse 包中的其他类访问，任何其他包的类都无法访问它。

3. protected

protected 是受保护的，只能修饰类的成员，代表的可访问范围比 default 大，除了本包中可见外，其他包中的子类也可以访问到 protected 修饰的成员，但其他包中的非子类就不能访问 protected 修饰的成员了。

例 5-28　ProtectedDemo.java，ProtectedTest.java，ProtectedTest2.java

```
package cn.whvcse;

public class ProtectedDemo {
    protected int a = 10;

    protected void hi() {

        System.out.println("Hi~~~,in ProtectedDemo");

        System.out.println("a = " + a);
    }
}

package cn.xyy;

import cn.whvcse.ProtectedDemo;

public class ProtectedTest extends ProtectedDemo {

    public void method() {
        super.hi();
        super.a = 20;
        System.out.println("now,in ProtectedTest , a = " + a);
    }

    public static void main(String[] args) {
        ProtectedTest ob = new ProtectedTest();
        ob.method();
    }
}
```

运行结果如图 5-29 所示。

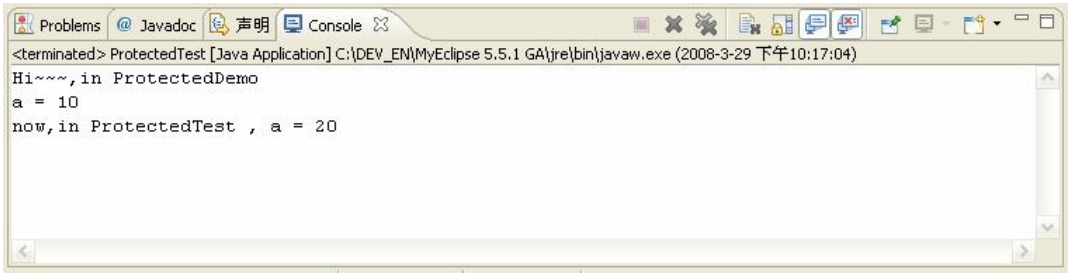

图 5-29　例 5-28 运行结果

若试图在其他包的非子类中访问 protected 修饰的成员，如 ProtectedTest2 类所示。

```
package cn.xyy;
```

```
import cn.whvcse.ProtectedDemo;

public class ProtectedTest2 {

    public static void main(String[] args) {
        ProtectedDemo ob = new ProtectedDemo();
        ob.hi();
    }
}
```

则会出现如图 5-30 所示出错信息。

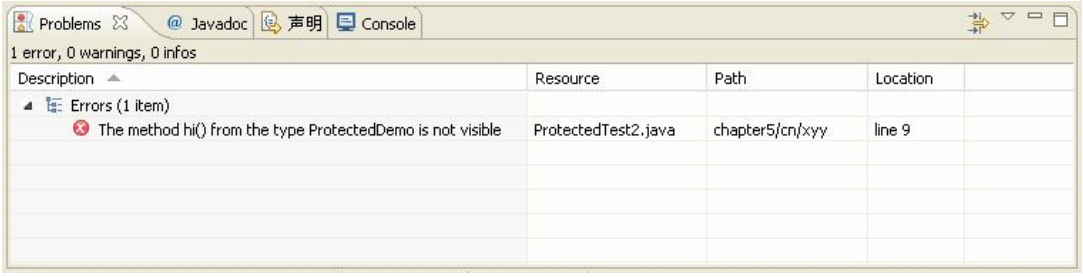

图 5-30　例 5-28 出错信息

protected 修饰的 hi()方法在 ProtectedTest2 类中不可见。

4. public

public 意为公共的，所代表的可访问范围是最广的，可以修饰类，也可以修饰类的成员，被 public 修饰的元素在本包和其他包中都是可见的。

被 public 修饰的类在其他包中可见，可以创建其对象，或是派生其子类。

被 public 修饰的成员在其他包中也可见，可以访问 public 的变量或是调用 public 的方法。通常将类提供的一些对外访问的接口方法声明为 public 的。

本章小结

- ✧ 用 new 创建类的对象，开辟对象的存储空间。Java 有垃圾回收机制能自动清理无用的对象。
- ✧ 若类中未定义构造方法，则会自动添加一个缺省的无参构造方法。
- ✧ 同种类型的对象可以相互赋值，这种赋值是引用赋值，使得两个对象名可以指向同一个对象。
- ✧ 若对象作为方法参数，则实参传给形参的是对象的引用，使得形参与实参指向同一个对象，则利用形参对对象进行修改能确实生效。
- ✧ 若有类的成员变量与方法的局部变量同名，则局部变量会将成员变量屏蔽掉，可以使用 this 引用来显式引用类的成员变量，解决名称冲突问题。
- ✧ 类的静态成员由 static 修饰，随类加载，不依赖于类对象的存在与否，静态方法中不能使用实例变量或实例方法。类的静态成员利用类名进行引用。

- ✧ 类中可以定义多个方法名相同但参数列表不同的方法，称为方法重载。方法重载是多态性的一种体现。
- ✧ 继承机制允许从现有的类中派生新类，Java 中继承利用关键字 extends 实现，Java 是单继承的，一个子类只能有一个直接父类。Java 的所有类都是直接或间接地从 java.lang.Object 派生而来的。
- ✧ 可以使用 final 关键字修饰类，防止类被继承；可以使用 final 修饰方法，防止方法被重写；可以使用 final 修饰变量，则该变量只能赋值 1 次，成为常量。
- ✧ 接口定义一些行为规范，实现接口的类具有这些行为规范，但要提供具体的实现，即要重写接口中的所有方法。
- ✧ 一个类可以实现多个接口，可以既继承类又实现接口，接口可以继承多个接口，这些都可以达到多继承的效果。
- ✧ Java 利用包分类管理和组织大量的类，利用 package 语句声明包，则类的.class 文件将被组织到包结构映射而来的目录结构中。包声明语句必须位于 Java 源文件的第一行。
- ✧ 使用 import 可以引入其他包中的类，除 java.lang 包不需要引入外，在使用系统提供的其他包中的类时，需要引入。
- ✧ Java 提供了 4 种访问控制符：private、default、protected 和 public，它们可以修饰类或类的成员，同包相结合，可以实现对类或类的成员的访问权限的控制。

编程练习题

1．创建一个名为 Fan 的类来模拟风扇，该类包含属性 speed、on 和 radius，此外还要求提供一个方法，用来设置风扇的速度和开关状态等信息。另外编写一个类来测试 Fan 的使用。

2．创建一个描述学生的类 Student，其中包括属性：学号、姓名、年龄和语文、数学、外语 3 门课程成绩，要求包含 1 个可以给所有属性赋初值的构造方法，要求给每一个属性定义一个设置（setXXX()）方法和一个获取方法（getXXX()），要求定义一个用于计算 3 门课程平均成绩的方法 average()。

编写一个包含 main()方法的测试类，测试 Student 类的使用，显示学生的基本信息和平均成绩，利用 setXXX()方法修改学生的各科成绩，再输出新的平均成绩。

3．创建名为 Point 的类描述点。创建名为 Shape 的类描述图形，Shape 类要求包含属性：代表图形左上角坐标的 location，Point 类型，包含方法 area()计算图形的面积。继承 Shape 创建圆类，增加必要的属性和方法；继承 Shape 创建矩形类，增加必要的属性和方法，圆类和矩形类都要求含有构造方法。

编写包含 main()的类测试圆类、矩形类的使用，能显示它们的左上角坐标，计算其面积等。

4．创建一个名为 Charge（收费）的接口，其中包含一个收取费用的方法声明：public void getCharge()；创建类 BusCharge 实现接口 Charge，实现显示公共汽车的收费标准，如“公汽票价：1.2 元/张，不计算公里数”；创建类 TaxiCharge 实现接口 Charge，实现显示出租车的收费标准，如“出租车：1.4 元/公里，起价 3 元”；创建类 CinemaCharge 实现接口 Charge，实现显示电影院的收费标准，如“电影票价：20 元/张”。

创建包含 main()方法的测试类，测试 BusCharge、TaxiCharge、CinemaCharge 类的使用。