第5章 类和对象(下)


在线答题


观看视频

5.1 类的静态成员

在声明类的成员时,如果使用了关键字 static,那么这个成员就是类的静态成员。没有用关键字 static 声明的成员是类的实例成员。

实例成员属于类的某一个具体实例(对象),而静态成员属于类。

类的静态成员包括静态数据成员和静态成员方法两部分。

5.1.1 静态数据成员

在类的声明中,如果定义数据成员时使用了关键字 static,则该数据成员就成为静态的数据成员。而没有使用 static 修饰的数据成员,称为类的实例数据成员。例如:

```
class A {
    …
    static int i;
    int j;
}
```

i 是类 A 的静态数据成员,而 j 是类 A 的实例数据成员。

顾名思义,类的实例数据成员属于该类的某个具体实例(对象)。它随着实例(对象)的产生而产生,随着实例(对象)的消亡而消亡。当程序中没有为类创建任何实例(对象)时,相应的实例数据成员也不存在。而属于同一个类的不同实例(对象)的同名的实例数据成员是相互独立的变量,占据不同的存储空间,互不影响。例如:

```
A a1 = new A();
A a2 = new A();
a1.j = 5;
a2.j = 10;
```

a1 和 a2 是类 A 的两个实例,a1.j 是对象 a1 的实例数据成员,a2.j 是对象 a2 的实例数据成员,它们属于各自的对象,占据不同的存储空间。修改 a1.j 的值,不会影响 a2.j。

和实例数据成员相反,类的静态数据成员属于类,而不属于类的某个具体实例(对象),在程序的运行过程中,类的静态数据成员只存在一份复本,占据一块特定的内存空间,并被该类的所有实例共享。例如,由于 i 是类 A 的静态数据成员,因此它被 A 的对象 a1 和 a2 共享。

在程序运行过程中,只要类被加载到内存中,即使还没有创建类的实例(对象),类的静态数据成员就已经存在了,它已经被分配了内存空间并被初始化。

通常在声明类的静态数据成员时就对其进行初始化，例如：

```
public static double PI = 3.14; //初始化静态数据成员
```

可以直接使用类名加点操作符访问公有的静态数据成员，当定义了类的实例（对象）后，也可以使用对象名加点操作符访问它。

不应在类的构造方法中初始化类的静态数据成员，因为静态数据成员不属于某个具体的实例，而类的构造函数是在创建类的实例（对象）时被调用，用于初始化对象的实例数据成员的。

例 5.1 演示了静态数据成员和实例数据成员的区别和访问公有的静态数据成员的方法。

【例 5.1】 类的静态数据成员。

程序代码如下：

```
class A{
    public static int i;
    public int j;
    public A(){
        j = 0;
    }
    public A(int jj){
        j = jj;
    }
}
public class TestStaticDataMember {
    public static void main(String[] args) {
        A.i = 10;
        System.out.println("类 A 的静态数据成员 i 等于" + A.i);
        A a1 = new A();
        A a2 = new A(1);
        a1.i = 100;
        a1.j = 2;
        System.out.println("类 A 的静态数据成员 i 等于" + a2.i);
        System.out.println("对象 a1 的实例数据成员 j 等于" + a1.j);
        System.out.println("对象 a2 的实例数据成员 j 等于" + a2.j);
    }
}
```

```
类A的静态数据成员i等于10
类A的静态数据成员i等于100
对象a1的实例数据成员j等于2
对象a2的实例数据成员j等于1
```

图 5.1 例 5.1 程序的运行结果

例 5.1 程序的运行结果如图 5.1 所示。

5.1.2 静态成员方法

在声明类的成员方法时，如果使用了关键字 static，则该成员方法为静态成员方法。如果声明成员方法时没有使用关键字 static，则称该成员方法为实例成员方法。和静态数据成员相同，类的静态成员方法也是属于类，而不属于类的具体实例（对象）。所以，既可以使用类名加点操作符调用静态成员方法，也可以使用对象名像调用实例成员方法一样调用静态成员方法。

在类的静态成员方法中只能访问类的静态数据成员，而不能访问类的实例数据成员。因为在调用静态成员方法时，可能还没有创建类的具体实例。而在类的实例成员方法中，既可以访问类的静态数据成员，也可以访问实例数据成员。

注意每个程序的主方法 main()就是一个静态方法,所以运行程序时不用先创建主类的实例(对象)。

【例 5.2】 类的静态成员方法。

程序代码如下:

```
class A{
    private static int i;
    private int j;
    public A(){
        j = 0;
    }
    public A(int jj){
        j = jj;
    }
    public static void setI(int ii){
        i = ii;
    }
    public static int getI(){
        return i;
    }
    public void setJ(int jj){
        j = i + jj;      //实例方法可以访问静态数据成员
    }
    public int getJ(){
        return j;
    }
}
public class TestStaticDataMember {
    public static void main(String[] args) {
        A.setI(10);
        System.out.println("类 A 的静态数据成员 i 等于" + A.getI());
        A a1 = new A();
        a1.setI(100);
        System.out.println("类 A 的静态数据成员 i 等于" + a1.getI());
        a1.setJ(10);
        System.out.println("对象 a1 的实例数据成员 j 等于" + a1.getJ());
    }
}
```

例 5.2 程序的运行结果如图 5.2 所示。

```
类A的静态数据成员i等于10
类A的静态数据成员i等于100
对象a1的实例数据成员j等于110
```

图 5.2　例 5.2 程序的运行结果

【例 5.3】 使用静态数据成员统计程序运行过程中创建 Circle 类的实例(对象)的个数。

问题分析:由于类的静态数据成员被所有对象共享,因此可以用来统计程序中创建对象的数目。可以为 Circle 类添加一个静态整型数据成员 numOfCircle,用来统计 Circle 对象的数目,程序中创建一个 Circle 对象,就把 numOfCircle 的值加 1。那么静态成员 numOfCircle 加 1 的操作应放在哪里呢?我们知道,只要创建类的对象,系统就会自动调用类的构造方法来初始化对象中的实例数据成员,所以可以把为 numOfCircle 加 1 的操作放在类的构造方法中。在 Circle 类中再添加一个公有的成员方法 getnumOfCircle,用来读取私有静态数据成员 numOfCircle 的值。

在 main()函数中，先后创建 3 个 Circle 对象，并输出对象的数目。

程序代码如下：

```
//Circle.java
public class Circle {
    private static int numOfCircle = 0; //定义静态数据成员，并初始化为 0
    private double radius;
    public Circle(){
        radius = 1.0;
        numOfCircle++;
    }
    public Circle(double r){
        radius = r;
        numOfCircle++;
    }
    public static int getNumOfCircle(){
        return numOfCircle;
    }
    public void setRadius(double r){
        radius = r;
    }
    public double getRadius(){
        return radius;
    }
    public double getArea(){
        double area;
        area = 3.14 * radius * radius;
        return area;
    }
}
//Test.java
public class Test {
    public static void main(String[] args){
        Circle c1 = new Circle();
        Circle c2 = new Circle(10.0);
        System.out.println("目前创建了 Circle 类的" + c1.getNumOfCircle() + "个对象");
        Circle c3 = new Circle(5.0);
        System.out.println("目前创建了 Circle 类的" + Circle.getNumOfCircle() + "个对象");
    }
}
```

例 5.3 程序运行的结果如图 5.3 所示。

```
目前创建了Circle类的2个对象
目前创建了Circle类的3个对象
```

图 5.3　例 5.3 程序的运行结果

5.2　类的 final 成员

在类声明中可以用关键字 final 把类成员声明为 final 型成员。final 型数据成员称为常量成员，在程序运行过程中，它的值是不能被修改的。可以在声明时就给它赋初值，或是在构造方法中对它进行初始化。

有些常量对于一个类的不同对象而言，它的值都是一样的。例如，圆周率对于 Circle 类的所有对象而言，它的值都是 3.141 592 6。在声明这种常量时，除了关键字 final，通常还会使用另外一个关键字 static。这样它就会成为一个静态常量，被 Circle 类的所有对象共享。否则，它将被存储在每一个对象中，这显然是不必要的，而且浪费了内存空间。

【例 5.4】 为 Circle 类添加一个表示圆周率的常量。

程序代码如下：

```
public class Circle {
    private static final double PI = 3.14;          //声明并初始化常量 PI
    private double radius;
    public Circle(){
        radius = 1.0;
    }
    public Circle(double r){
        radius = r;
    }
    public void setRadius(double r){
        radius = r;
    }
    public double getRadius(){
        return radius;
    }
    public static double getPI(){
        return PI;
    }
    public double getArea(){
        double area;
        area = PI * radius * radius;                //使用常量 PI
        return area;
    }
}
public class Test {
    public static void main(String[] args){
        System.out.println("圆周率的值为" + Circle.getPI());
    }
}
```

例 5.4 程序的运行结果如图 5.4 所示。

```
圆周率的值为3.14
```

图 5.4　例 5.4 程序的运行结果

关键字 final 修饰的方法的作用是使该方法不能被子类重写。

5.3　关键字 this

在介绍关键字 this 之前，有必要了解对象内存空间的结构。

5.3.1　对象的内存空间

当程序中使用关键字 new 创建一个类的对象时，编译器会在“堆”中为对象分配内存空间。这块属于对象的内存空间中只保存这个对象的所有的实例数据成员，而不包含类的静态成员和实例成员方法。实例成员方法虽然也属于对象，但对于这个类的不同对象而言，实

例方法内容都完全相同，所以没必要把它们保存在每个对象中。类的实例方法单独存放，并被这个类的所有对象共享。

由于实例方法被这个类的所有对象共享，当方法被调用时，在方法内部需要知道是哪个对象调用了它，实例方法中有可能会访问对象的实例数据成员，而不同对象的实例数据成员的值是不同的。例如：

```
Circle c1 = new Circle();
Circle c2 = new Circle();
c1.setRadius(10.0);
c2.setRadius(20.0);
double area = c1.getArea();
```

如上程序片段先创建了两个 Circle 类对象：c1 和 c2。再把 c1 对象的实例变量 radius 的值设置为 10.0，把 c2 对象的实例变量 radius 的值设置为 20.0。然后由对象 c1 调用 Circle 类的实例方法 getArea()求圆的面积。实例方法 getArea()是被对象 c1 和 c2 共享的，如何让方法 getArea()知道调用它的当前对象是 c1 而不是 c2 呢？

Java 解决此问题的方法是使用关键字 this。

5.3.2 关键字 this 引用调用实例方法的当前对象

关键字 this 本质上是类的引用变量，它被保存在每个对象的内存空间中，this 变量中保存本对象的地址。

当实例方法被某个对象调用时，编译器会把引用这个对象的 this 传递给被调用的实例方法，这样实例方法内部就可以使用 this 访问调用它的当前对象了。

例如，当编译器编译“double area = c1.getArea();”这条语句时，会把 c1 对象的 this 传递给实例方法 getArea()，getArea()内部可以使用 this 引用调用方法的当前对象 c1，访问 c1 的实例成员 radius 求出对象 c1 的面积。

在编写类的实例成员方法时，可以显式地使用关键字 this 访问当前对象的实例成员。例如，Circle 类的实例成员方法 getArea()可以写成如下形式：

```
public double getArea(){
    double area;
    area = PI * this.radius * this.radius;
    return area;
}
```

这样的写法指明了 radius 是由 this 引用的调用方法的当前对象的实例数据成员，使程序的逻辑更加清晰，容易理解。

除此之外，关键字 this 还有其他的作用。

5.3.3 在构造方法中使用 this 调用其他构造方法

如果一个类声明了多个构造方法，那么可以在一个构造方法中使用关键字 this 调用其他的构造方法。

【例 5.5】 在类的构造方法中使用关键字 this 调用其他构造方法。

程序代码如下：

```
public class Circle {
    private static final double PI = 3.14;
    private double radius;
    public Circle(){
        this(1.0);
    }
    public Circle(double r){
        radius = r;
    }
    …
}
```

Circle 类声明了两个构造方法。默认构造方法中的语句“this(1.0);”调用带参数的构造方法，并把 1.0 传递给它。

5.3.4　使用 this 访问被局部变量屏蔽的数据成员

有些成员方法里有可能定义了和对象的数据成员同名的局部变量，或定义了和数据成员同名的形式参数，此时，局部变量或形式参数屏蔽了同名的数据成员，即在方法内部使用该变量名访问的是方法内定义的局部变量，而不是类的数据成员。这时可以借助关键字 this 访问被屏蔽的数据成员。例如：

```
public Circle(double radius){          //形式参数和数据成员同名
    this.radius = radius;              //使用 this 可以访问数据成员
}
```

以上是 Circle 类的构造方法，由于其形参被命名为 radius，则在方法内部必须使用关键字 this 访问对象的实例数据成员 radius。在语句“this.radius＝radius;”中，赋值号左侧的 this.radius 代表对象的数据成员——圆的半径；赋值号右侧的 radius 是方法的形式参数。

5.3.5　从实例方法返回调用方法的当前对象

使用关键字 this 可以从实例方法返回调用该方法的当前对象。

【例 5.6】 为 Circle 类添加一个实例方法，用来比较两个 Circle 类对象的大小，并返回其中较大的一个对象。

程序代码如下：

```
//Circle.java
public class Circle {
    private static final double PI = 3.14;
    private double radius;
    public Circle(){
        this(1.0);
    }
    public Circle(double radius){
        this.radius = radius;
    }
    public void setRadius(double radius){
        this.radius = radius;
    }
    public double getRadius(){
        return radius;
```

```
    }
    public static double getPI(){
        return PI;
    }
    public double getArea(){
        double area;
        area = PI * radius * radius;
        return area;
    }
    public Circle compare(Circle c){      //此方法比较当前对象和参数对象的大小,返回较大的
        if(this.radius > c.radius)
            return this;                  //使用关键字 this 返回当前对象
        else
            return c;
    }
}
//Test.java
public class Test {

    public static void main(String[] args) {
        Circle c1 = new Circle(10.0);
        Circle c2 = new Circle(20.0);
        Circle c3 = c1.compare(c2);
        System.out.println("面积较大的圆的半径为" + c3.getRadius());

    }
}
```

如上程序中的黑体字部分是为 Circle 添加的实例方法 compare()，该方法比较调用方法的当前对象和方法参数对象的大小，并返回较大的对象。如果当前对象的半径比参数对象的半径大，则使用关键字 this 返回当前对象。

```
面积较大的圆的半径为20.0
```

图 5.5　例 5.6 程序的运行结果

例 5.6 程序的运行结果如图 5.5 所示。

5.4　类组合

类的数据成员既可以是 Java 数据基本类型的变量，也可以是类的对象。如果数据成员是一个对象，则这种结构称为类组合。类组合通常用于描述对象之间"has-a"的关系。例如，如果类 A 对象的数据成员是类 B 的对象，则可以说“类 A 对象中包含(有)一个类 B 对象”。

【例 5.7】 创建一个 Point 类表示平面上的一个点，包含两个整型数据成员(x 和 y)，存储点的坐标。创建一个包含圆心坐标的 Circle 类，其圆心坐标保存在一个 Point 类的对象里。

程序代码如下：

```
//Point.java
public class Point {
    private int x;
    private int y;
    public Point(){
    }
    public Point(int x,int y){
```

```
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }
}
//Circle.java
public class Circle {
    private double radius;
    private Point centerOfCircle; //圆心坐标
    public static final double PI = 3.14;
    public Circle(){
        radius = 1.0;
        centerOfCircle = new Point();
    }
    public Circle(double radius,Point centerOfCircle){
        this.radius = radius;
        this.centerOfCircle = centerOfCircle;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public Point getCenterOfCircle() {
        return centerOfCircle;
    }
    public void setCenterOfCircle(Point centerOfCircle) {
        this.centerOfCircle = centerOfCircle;
    }
    public double getArea(){
        return radius * radius * PI;
    }
}
//Test.java
public class Test {
    public static void main(String[] args) {
        Point center = new Point(10,10);
        Circle c1 = new Circle();
        Circle c2 = new Circle(10.0,center);
        System.out.println("第一个圆的圆心坐标是(" + c1.getCenterOfCircle().getX() + "," +
        c1.getCenterOfCircle().getY() + "),半径是" + c1.getRadius());
        System.out.println("第二个圆的圆心坐标是(" + c2.getCenterOfCircle().getX() + "," +
```

```
            c2.getCenterOfCircle().getY() + "),半径是" + c2.getRadius());
        }
    }
```

程序中的 Circle 类的数据成员 centerOfCircle 是 Point 类的对象，这种结构就是类组合。例 5.7 程序的运行结果如图 5.6 所示。

```
第一个圆的圆心坐标是(0,0),半径是1.0
第二个圆的圆心坐标是(10,10),半径是10.0
```

图 5.6　例 5.7 程序的运行结果

类组合的本质是：一个对象中含有一个指向另一个对象的引用变量，它是一个联系不同对象的纽带，使程序中的一个对象可以感知到其他对象的存在，并互发消息，协同工作。换句话说，两个对象之间即使没有逻辑上的 has-a 关系，也可以通过类组合在它们之间建立联系。在 Java 程序的实践应用中，类组合技术常用于实现各种对象型设计模式。

5.5　数组

数组是一系列相同类型对象的集合，组成数组的对象叫数组的元素。数组在存储器中是连续存放的。数组可以是一维的，也可以是多维的；一维数组的元素只有一个下标，n 维数组元素有 n 个下标。数组可以由任何类型的元素构成（包括基本数据类型和类的对象）。在 Java 语言中，数组是一个对象。

5.5.1　一维数组

创建一维数组的语法格式如下：

```
数据类型[] 引用变量 = new 数据类型[元素个数];
```

或

```
数据类型 引用变量[] = new 数据类型[元素个数];
```

其中的数据类型是数组元素的数据类型，引用变量用来引用这个数组对象。例如：

```
int array[] = new int[10];
```

如上语句定义了一个包含 10 个元素的一维整型数组。

创建了数组对象之后，数组里的每个元素都被赋予默认的值。整型、浮点型等数值型数组元素的默认值为 0；字符数组元素的默认值为 Unicode 码值为 0 的空字符；布尔型数组元素的默认值为 false。

可以在创建数组对象的同时对数组进行初始化，即给数组中的每个元素赋初值。此时不能声明数组元素的个数。例如：

```
int array[] = new int[]{6,5,4,3,2,1};
```

还可以省略关键字 new，使用更简单的写法创建并初始化数组。例如：

```
int array[] = {6,5,4,3,2,1};
```

如上语句创建了一个包含 6 个元素的整型数组，并为其中的每个元素赋了初值。

可以使用下标直接访问数组中的某个元素。例如：

```
array[2] = 100; //把下标为 2 的数组元素赋值为 100
```

注意：Java 数组的下标是从 0 开始的，上面定义的包含 10 个元素的数组的第一个元素的下标为 0，即数组的第一个元素是 array[0]，最后一个元素是 array[9]。

可以通过数组对象的数据成员 length 来获取数组的容量(数组中元素的个数)。

【例 5.8】 编写一个方法 sort()对整型数组按升序进行排序。方法的参数是待排序的数组。

程序代码如下：

```
public class TestOneDemArray {
    public static void sort(int array[]){
        int min,minIndex,i,j;
        for(i = 0;i < array.length - 1;i++){
            min = array[i];
            minIndex = i;
            for(j = i + 1;j < array.length;j++){
                if(array[j]< min){
                    min = array[j];
                    minIndex = j;
                }
            }
            if(i!= minIndex){
                array[minIndex] = array[i];
                array[i] = min;
            }
        }
    }
    public static void main(String[] args){
        int[] a = {12,3,34,64,9,56,21,76,5,1};
        System.out.print("排序前: ");
        for(int i = 0;i < a.length;i++){
            System.out.print(a[i] + " ");
        }
        sort(a);
        System.out.println(); //换行
        System.out.print("排序后: ");
        for(int i = 0;i < a.length;i++){
            System.out.print(a[i] + " ");
        }
    }
}
```

sort()方法使用选择排序法对参数数组进行排序。选择排序法的算法是：在包含 n 个元素的无序序列中，选择一个最小的元素，放到无序序列的最前面。这个过程称为 1 次选择。然后再在后面包含 n－1 个元素的无序序列中重复上面的选择过程。经 n－1 次选择后，整个序列就被以升序排序了。

sort()方法使用两重循环实现选择排序，外层循环控制选择的次数，内层循环实现 1 次选择。

在 main()方法中创建了一个无序的整型数组，并把它作为参数传递给 sort()方法。由于数组是一个对象，根据 4.7.3 节的学习可知，当方法参数是对象时，实际传递的是引用对

象的地址，而不是对象本身。也就是说，sort()方法中的形参数组 array 和 main()方法中的实参数组 a 实际上是 1 个数组。所以在 sort()方法里对形参数组 array 进行排序，就是对 main()方法里的数组 a 进行排序。

```
排序前: 12 3 34 64 9 56 21 76 5 1
排序后: 1 3 5 9 12 21 34 56 64 76
```

图 5.7　例 5.8 程序的运行结果

例 5.8 程序的运行结果如图 5.7 所示。

5.5.2　二维数组

创建二维数组的语法格式如下：

```
数据类型[][] 引用变量 = new 数据类型[元素个数][元素个数];
```

或

```
数据类型 引用变量[][] = new 数据类型[元素个数][元素个数];
```

例如：

```
int[][] a1 = new int[3][3];
double a2 = new double[3][2];
```

上面第 1 条语句创建了一个 3 行、3 列的二维整型数组，第 2 条语句创建了一个 3 行、2 列的二维双精度浮点型数组。

可以在创建二维数组的同时对其进行初始化，例如：

```
int[][] a1 = new int[][]{{1,2,3},{4,5,6,},{7,8,9}};
```

也可以省略关键字 new，使用更简洁的写法创建并初始化二维数组。例如，如上语句可以写成：

```
int[][] a1 = {{1,2,3},{4,5,6},{7,8,9}};
```

通过对一维数组的学习可知，数组对象的数据成员 length 中保存数组元素的个数。那么对于如上语句中定义的二维数组 a1 对象来说，其数据成员 length 中保存的值是多少呢？是 9 吗？在此不妨验证一下，可用如下语句输出其值：

```
System.out.print(a1.length); //输出二维数组对象 a1 中的元素个数
```

如上语句的输出结果是 3，不是 9。

这是因为 Java 编译器认为，二维数组本质上是一个由一维数组构成的一维数组。也就是说，二维数组的每个元素是一个一维数组。所以，二维数组对象的数据成员 length 中保存的是二维数组中一维数组的个数，也就是二维数组的行数。

【例 5.9】 从键盘输入一个 3 行×3 列的矩阵，输出该矩阵，将该矩阵转置之后再次输出。

程序代码如下：

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args){
        int[][] a = new int[3][3];
        int temp;
        System.out.println("请输入 9 个 100 之内的整数");
        Scanner in = new Scanner(System.in);
        for(int i = 0;i < a.length;i++){
            for(int j = 0;j < a[i].length ;j++){
```

```
                a[i][j] = in.nextInt();
            }
        }
        System.out.println("------");
        for(int i = 0;i < a.length;i++){
            for(int j = 0;j < a[i].length ;j++){
                System.out.printf("%3d",a[i][j]);;
            }
            System.out.println();
        }
        for(int i = 0;i < a.length;i++)
            for(int j = 0;j < a[i].length ;j++)
                if(i < j) {
                    temp = a[i][j];
                    a[i][j] = a[j][i];
                    a[j][i] = temp;
                }
        System.out.println("------");
        for(int i = 0;i < a.length;i++){
            for(int j = 0;j < a[i].length ;j++){
                System.out.printf("%3d",a[i][j]);;
            }
            System.out.println();
        }
    }
}
```

如上程序中创建了一个 3×3 的二维数组用来存放 3 行×3 列的矩阵。矩阵转置指把矩阵上三角阵中的元素和下三角阵中的对应元素交换位置。对于二维数组元素 a[i][j]，如果 i<j，则它位于上三角阵中，和它相对应的下三角阵元素是 a[j][i]。

例 5.9 程序的运行结果如图 5.8 所示。

```
请输入9个100之内的整数
12 43 25 67 17 8 54 87 71
------
 12 43 25
 67 17  8
 54 87 71
------
 12 67 54
 43 17 87
 25  8 71
```

图 5.8　例 5.9 程序的运行结果

5.5.3　foreach 循环语句

foreach 语句是 JDK 1.5 引入的一种专门用于操作数组、集合的循环控制语句。使用 foreach 循环语句可以更加方便地遍历数组和集合中的元素。

foreach 循环语句的语法结构如下：

```
for(数据类型 标识符: 数组或集合对象){
    //循环体语句;
}
```

其中的“数据类型”指数组或集合元素的数据类型，标识符代表每次循环遍历的数组或集合中的元素，冒号后面是被遍历的数组对象或集合对象。

【例 5.10】 使用 foreach 循环语句遍历并输出一个 3×3 的二维数组。

程序代码如下：

```
public class TestForeach {
    public static void main(String[ ] args){
        int array[ ][ ] = {{1,2,3},{4,5,6},{7,8,9}};
        for(int[ ] v1:array){
```

```
            for(int v2:v1)
                System.out.printf(" % 2d", v2);
            System.out.println();
        }
    }
}
```

```
1 2 3
4 5 6
7 8 9
```

图 5.9 例 5.10 程序的运行结果

如上程序使用两重 foreach 循环结构变量二维数组 array，外层 foreach 循环遍历数组 array 中的每个一维数组对象，内层循环遍历一维数组中的每个整型元素。

例 5.10 程序的运行结果如图 5.9 所示。

5.5.4 对象数组

如果数组中的每个元素是某个类的对象，那么这个数组就是一个对象数组。创建一维对象数组的语法格式如下：

类名[] 数组名 = new 类名[元素个数];

或

类名 数组名[] = new 类名[元素个数];

例如：

```
String[] strs = new String[5];
Circle circles[] = new Circle[3];
```

使用类似上面的语句创建了对象数组之后，数组中的每个元素是引用对象的引用变量，因为还没有为它们创建引用的对象实体，所以数组元素的默认值为 null。也就是说，此时的对象数组还不完整，还要进一步为每个数组元素创建它们引用的对象实体。例如，针对上面创建的字符串数组 strs 和 Circle 类的对象数组 circles，可以使用如下两条循环语句为数组中的每个元素创建它引用的对象实体。

```
for(int i = 0; i < strs.length; i++) {
    strs[i] = new String("Hello Java!");
}
for(int i = 0; i < circles.length; i++){
    circles[i] = new Circle();
}
```

可以在创建对象数组的同时对其进行初始化。例如：

```
String[] st = new String[]{"hello","Welocome to", new String("Java")};
Circle c[] = new Circle[]{new Circle(),new Circle(10.0),new Circle(5.0)};
```

这种做法将创建对象数组和创建数组中的每个对象合并为一步完成。上面的语句还可以进一步简化如下：

```
String[] st = {"hello","Welocome to", new String("Java")};
Circle c[] = {new Circle(),new Circle(10.0),new Circle(5.0)};
```

【例 5.11】 创建一个包含 3 个元素的 Circle 类对象数组，并输出每个圆对象的面积。程序代码如下：

```
//Circle.java
```

```
public class Circle {
    private static final double PI = 3.14;
    private double radius;
    public Circle(){
        this(1.0);
    }
    public Circle(double radius){
        this.radius = radius;
    }
    …
    public double getArea(){
        double area;
        area = PI * radius * radius;
        return area;
    }
}
//TestObjectArray.java
public class TestObjectArray {
    public static void main(String[] args){
        Circle[] arrayOfCircle = {new Circle(),new Circle(10.0),new Circle(5.0)};
        for(Circle circle:arrayOfCircle){
            System.out.println("半径为" + circle.getRadius() + "的圆的面积为"
        + circle.getArea());
        }
    }
}
```

例 5.11 程序的运行结果如图 5.10 所示。

```
半径为1.0的圆的面积为3.14
半径为10.0的圆的面积为314.0
半径为5.0的圆的面积为78.5
```

图 5.10　例 5.11 程序的运行结果

5.6　递归方法

递归是一种解决问题的方法。例如,可以用递归方法求正整数 n 的阶乘 n!。n!的递归定义如下:

n!=n×(n−1)!(当 n>0 时)　　①

n!=1　　(当 n=0 时)　　②

以上两式给出了 n!的递归定义。当 n>0 时,n!=n×(n−1)!;同理(n−1)!=(n−1)×(n−2)!;以此类推。这是一个**回溯**的过程,目的是把规模较大的问题逐步化简为**相同类型的**规模较小的问题。当 n=0 时,n!=0,这时就回到了源头,这是**回溯终止的条件**,也就是说,当问题足够小时,可以容易地得到问题的解,从而终止回溯的过程。根据公式①可以从 0!递推出 1!=1×0!=1,再从 1!推出 2!,……,最后求出 n!。这是一个**递推**的过程,从小问题的解逐步推出规模较大的问题的解,最后得到原始问题的解。可以看到,用递归的方法解决问题主要包含回溯和递推两个过程。

Java 使用递归方法解决递归问题。**如果一个方法直接或间接地调用了自己,则这个方法就是递归方法**。例如,可以定义一个求 n!的静态递归方法 fac()。

```
public static long fac(int n){
    if(n == 0) return 1;
    else return n * fac(n - 1);      //对当前方法进行递归调用
}
```

图 5.11 中以 n＝4 为例，模拟 fac()方法的调用过程，揭示了递归方法的执行原理。

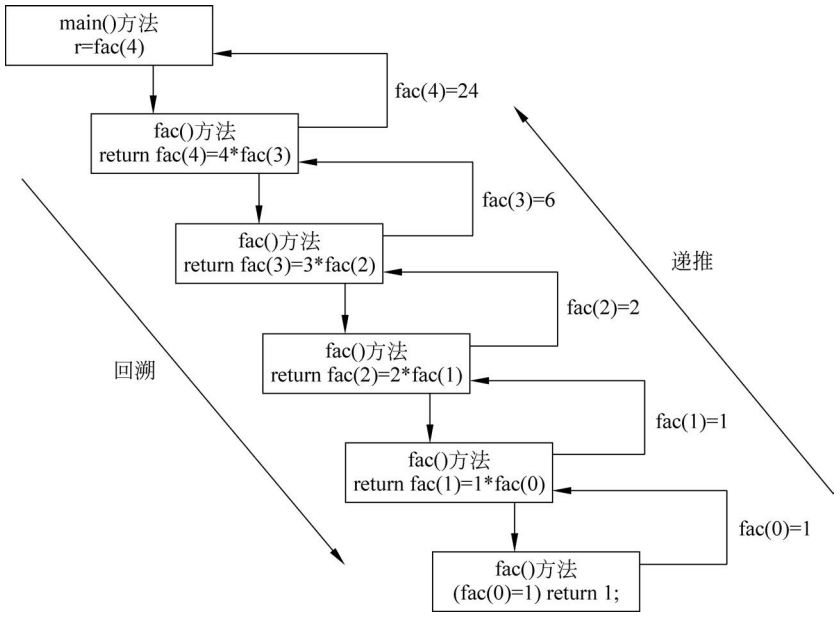


图 5.11　递归方法 fac()的调用过程

图 5.11 中的矩形表示一次方法调用，矩形间向下的箭头从主调方法指向被调方法，矩形右侧水平方向的箭头表示一次方法调用结束后返回到主调方法。从图 5.11 中可以看出，每一次递归方法调用执行结束后，总是返回到主调方法中调用它的地方继续向后执行。

递归方法的逐级调用就是从繁到简的回溯过程；而到达回溯终止条件后，方法开始逐级返回，这是由简到繁的递推过程。

【例 5.12】 编写一个递归方法，在一个数组中使用二分查找法查找一个数值在数组中的下标。

算法思想：二分查找法是一种高效的查找算法，这种算法的前提是待查数组必须是有序的。所以算法的第一步是将数组排序。

假设数组元素是按升序排列的，在数组中二分查找的步骤如下。

(1) 如果查找范围中(第 1 次查找的查找范围是整个数组)已经没有元素，则输出“查无此数”，查找过程结束。否则，用待查数值和位于查找范围中间位置的元素值相比较，如果相等，则输出找到元素的下标值，查找过程结束。

(2) 如果待查数据的值小于查找范围中间位置的元素，由于数组中的元素是按升序排列的，因此可以修改查找范围，把新的查找范围缩减到原查找范围的前一半。转第(1)步。

(3) 如果待查数据的值大于查找范围中间位置的元素，则修改查找范围，把新的查找范围缩减到原查找范围的后一半。转第(1)步。

程序完整代码如下：

```
import java.util.Arrays;
import java.util.Scanner;
public class BinarySe {
    /*binarySe()方法实现二分查找算法,参数 key 是待查数据,array 是待查数组,整型参数 low 是待查范围的下限,初始值为 0; 整型参数 high 是待查范围的上限,初始值为 array.length-1; 如果在数组 array 中找到待查值,则返回它在数组中的下标; 否则返回 -1*/
    public static int binarySe(int key,int[] array,int low,int high){
        if(low>high)                    //待查范围的下限大于上限,说明范围中已经没有元素
            return -1;                  //没找到,返回 -1
        else{
            int mid=(low+high)/2;       //获取位于待查范围中间位置的数组元素下标
            if(key==array[mid]){        //如果待查值和待查范围中间位置的数组元素相等
                return mid;             //则返回找到元素的下标
            }
            else if(key<array[mid]){  //如果待查值小于待查范围中间位置的数组元素
              return binarySe(key,array,low,mid-1);
              //则上面的语句递归调用 binarySe()方法,在范围[low,mid-1]中二分查找
            }
            else{                       //如果待查值大于待查范围中间位置的数组元素
              return binarySe(key,array,mid+1,high);
              //则上面的语句递归调用 binarySe()方法,在范围[mid+1,high]中二分查找
            }
        }
    }
//sort()方法把待查数组按升序排序
    public static void sort(int[] array){
        int min=-1;
        int indexMin=-1;
        for(int i=0;i<array.length-1;i++){
            min=array[i];
            indexMin=i;
            for(int j=i+1;j<array.length;j++){
                if(array[j]<min){
                    min=array[j];
                    indexMin=j;
                }
            }
            if(indexMin!=i){
                array[indexMin]=array[i];
                array[i]=min;
            }
        }
    }
    public static void main(String[] args){
        Scanner input=new Scanner(System.in);
        int num;
        System.out.println("请输入整数的个数:");
        num=input.nextInt();
        int[] array=new int[num];       //创建包含 num 个元素的整型数组 array
        for(int i=0;i<array.length;i++){
            array[i]=(int)(Math.random()*100)+1;   //使用 1～100 的随机数初始化数组
        }
        System.out.println("排序之前:");
        for(int var:array){
```

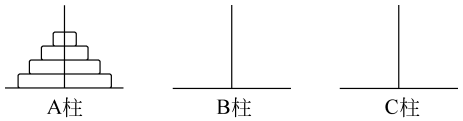
```
            System.out.print(var + " ");     //输出数组
        }
        System.out.println();
        sort(array);                         //将数组 array 按升序排序
        System.out.println("排序之后: ");
        for(int var:array){
            System.out.print(var + " ");     //输出排序后的数组
        }
        System.out.println();
        System.out.println("请输入要查找的整数: ");
        num = input.nextInt();               //输入待查值
        int index = binarySe(num,array,0,array.length - 1);
        //上面的语句调用递归方法 binarySe()在[0,array.length - 1]范围中进行二分查找
        if(index == -1)
            System.out.println("查无此数");
        else
            System.out.println(num + "在数组中的索引为: " + index);
    }
}
```

```
请输入整数的个数:
10
排序之前:
72 26 27 3 96 58 67 98 50 82
排序之后:
3 26 27 50 58 67 72 82 96 98
请输入要查找的整数:
27
27在数组中的索引为: 2
```

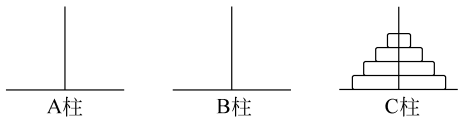
图 5.12　例 5.12 程序的运行结果

例 5.12 程序的运行结果如图 5.12 所示。

【例 5.13】 汉诺塔问题。有 A、B、C 三根柱子，在 A 柱上套着 n 个盘子，盘子的大小互不相同，且大盘在下、小盘在上，如图 5.13(a)所示。现在要借助 B 柱把这 n 个盘子从 A 柱移动到 C 柱，如图 5.13(b)所示。移动的规则是：①每次只能移动一个盘子；②移动中的任何时刻，在三根柱上的盘子必须大盘在下、小盘在上。请找出最佳的移动步骤。



(a) 移动前



(b) 移动后

图 5.13　汉诺塔问题

分析：把 n 个盘子从 A 柱借助 B 柱移动到 C 柱可以分为以下三个步骤进行。

(1) 把 A 柱上面的(n－1)个盘子从 A 柱借助 C 柱移动到 B 柱。

(2) 把 A 柱上剩下的一个盘子移动到 C 柱。

(3) 把 B 柱上的(n－1)个盘子借助 A 柱移动到 C 柱。

以上三步采用递归的方法对问题进行了分解。而递归终止条件应该是 n＝1，即 1 个盘子可以从 A 柱直接移动到 C 柱。

通过上面的分析，可以容易地写出移动盘子的 Java 方法。

程序代码如下：

```
public static void hanoi(int n,char a,char b,char c){
    If(n == 1)
        System.out.println(a + " to " + c );
    else
    {
        hanoi(n - 1,a,c,b);
        System.out.println(a + " to " + c );
        hanoi(n - 1,b,a,c);
    }
}
```

hanoi()方法的参数 n 存储盘子数，三个字符型的参数 a、b、c 用来存储代表三根柱子的大写字母'A'、'B'、'C'；方法中首先判断是否满足递归调用的终止条件(n＝＝1)，如果满足条件，则把 a 柱上的一个盘子直接移动到 c 柱上，方法运行结束，返回到上次调用处继续执行；如果不满足递归终止条件，则按上面的分析分三步进行，其中包含两次递归调用。以 n＝3 为例，方法递归调用的步骤如图 5.14 所示。

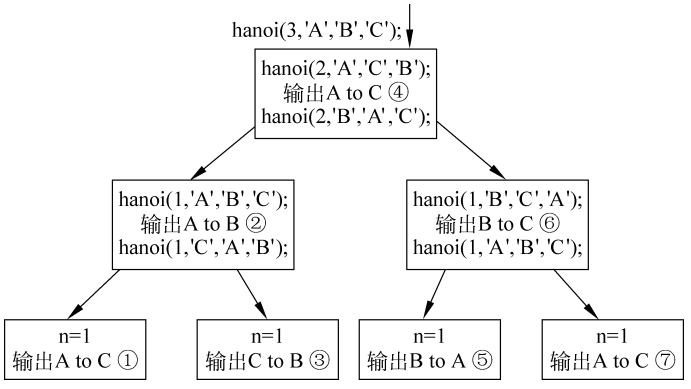


图 5.14　n＝3 时函数 hanoi()的递归调用过程

图 5.14 中的矩形代表一次方法调用，箭头代表方法调用的方向，箭头末端是方法调用语句，箭头指向调用的方法。图 5.14 中的标号①～⑦代表程序输出的顺序，即移动盘子的顺序。

完整的程序代码如下：

```
import java.util.Scanner;
public class Hanoi {
    public static void hanoi(int n,char a,char b,char c){
        if(n == 1)
            System.out.println(a + " to " + c);
        else{
            hanoi(n - 1,a,c,b);
            System.out.println(a + " to " + c);
            hanoi(n - 1,b,a,c);
        }
    }
    public static void main(String[ ] args){
        int num;
        System.out.print("请输入盘子的个数:");
        Scanner in = new Scanner(System.in);
        num = in.nextInt();
```

```
        System.out.println("开始移动盘子");
        hanoi(num,'A','B','C');
    }
}
```

例 5.13 程序的运行结果如图 5.15 所示。

```
请输入盘子的个数:4
开始移动盘子
A to B
A to C
B to C
A to B
C to A
C to B
A to B
A to C
B to C
B to A
C to A
B to C
A to B
A to C
B to C
```

图 5.15　例 5.13 程序的运行结果

5.7　方法的可变长参数

JDK 1.5 为方法引入了可变长参数，可变长参数使方法可以接收任意数目的参数，强化了方法处理数据信息的能力。声明可变长参数的语法格式如下：

数据类型 ...参数名

其中，参数名前的符号...代表此参数是一个可变长参数。可变长参数代表一系列数目不确定的参数。在方法内部使用数组处理可变长参数。例如：

```
int sum(int ...a){
    ...;
}
```

如上方法具有可变长参数 a，在方法内部，a 是一个整型数组。

注意：如果一个方法有多个参数，而且其中包含可变长参数，则可变长参数必须是参数列表中最右边的一个。

【例 5.14】 编写一个 Java 方法，计算并返回多个整数的和。

程序代码如下：

```
public class VLPTest {
public static int add(int ...a){
    int sum = 0;
    for(int var: a){
        sum = sum + var;
    }
        return sum;
}
public static void main(String[] args){
    int sum;
    sum = add(1,2,3,4);
    System.out.println("整数 1 到 4 的累加和为" + sum);
```

```
        sum = add(1,2,3,4,5,6,7,8,9,10);
        System.out.println("整数 1 到 10 的累加和为" + sum);
    }
}
```

例 5.14 程序的运行结果如图 5.16 所示。

```
整数1到4的累加和为10
整数1到10的累加和为55
```

图 5.16　例 5.14 程序的运行结果

5.8　包

包是 Java 中避免类名冲突的机制。在 Java 程序中，所有的类都位于包中，例如，前面程序中经常使用的 System 类位于 java.lang 包中，Scanner 类位于 java.util 包中。可以使用关键字 package 为自己设计的类打包。打包语句的语法格式如下：

```
package 包名;
```

这条打包语句必须是程序文件的第一条语句。例如：

```
package com.MyClass;
```

如上语句把当前类打包到 com.MyClass 包中。如果类文件中没有打包语句，则当前类位于一个无名包里。

如果一个类被打包，那么它必须存放在包里。假设包名是 com.MyClass，则计算机内必须包含 com/MyClass 这两级子目录，打包类的字节码文件必须保存在这两级子目录中。

程序中除了可以使用 JDK 中的类，还可以使用来自第三方包中的类，包括自己开发并打包的类。如果想使用某个包中的类，则必须使用 import 语句导入这个类。import 语句的语法格式如下：

```
import 包名.类名;
```

或

```
import 包名.*
```

其中的星号(*)表示导入这个包里所有的类，例如：

```
import java.util.Scanner;      //导入 java.util 包中的 Scanner 类
import java.util.*;            //导入 java.util 包中所有的类
import com.MyClass.*;          //导入自己打包的所有类
```

下面以一个实例介绍为类打包并使用打包类的过程。

【例 5.15】 使用 Windows 系统自带的文本编辑器“记事本”开发 Java 程序时，对类打包和使用自己打包的类的过程进行演示。

(1) 在计算机硬盘的某个子目录中创建 com/myClass 两级子目录。例如，在 D 盘的 class 子目录。

(2) 把在第 4 章中开发的 Person 类和 Circle 类的源文件复制到这个目录里，并在两个类中添加打包语句“package com.MyClass;”。注意，打包语句必须是源文件的第 1 条语句。

（3）打开“命令提示符”，进入路径 D:\class\表示的子目录。使用 JDK 命令编译两个类，如图 5.17 所示。

```
D:\class>javac com\MyClass\Circle.java

D:\class>javac com\MyClass\Person.java

D:\class>
```

图 5.17　在命令行编译 Circle 类和 Person 类

可以发现，经编译后，在目录 D:\class\com\MyClass 中出现了两个类的字节码文件 Person.class 和 Circle.class。到目前为止，Person 类和 Class 类已经被成功打包。

（4）使用打包的类。为了使打包的类可以被其他程序使用，需要设置环境变量 classpath。打开计算机“系统属性”对话框，如图 5.18 所示。单击“环境变量”按钮打开“环境变量”对话框，如图 5.19 所示。

图 5.18　“系统属性”对话框

如果环境变量中没有 classpath，则添加一个 classpath 环境变量，并把它的值设置为字符串".;D:\class"。环境变量 classpath 的作用是设置加载类的路径。环境变量 classpath 的值被分号分成两部分，第一部分是一个“.”，这个点代表程序所在的当前目录；第二部分是 D:\class，这是 com.MyClass 这个包所在的目录。程序在加载一个类时，会根据 classpath 的值先在当前路径查找要加载的类的字节码文件。如果没有找到，会到 D:\class 路径中查找；如果还没有找到，会去 JDK 默认路径中查找。

如果环境变量 classpath 已经存在，则编辑该环境变量，在它原有内容的末尾添加字符

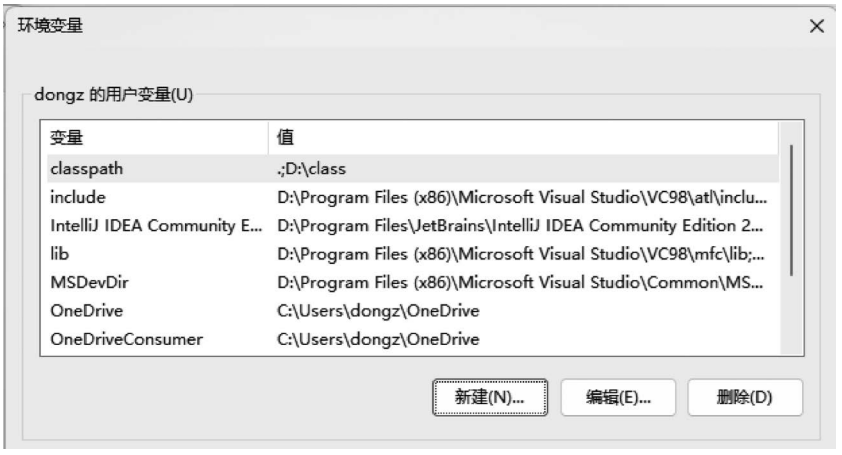


图 5.19　在“环境变量”对话框中添加或编辑环境变量 classpath

串";D:\class"即可。

(5) 编写程序使用前面打包的类。在计算机其他目录中编写如下测试类。

```
import com.MyClass.*;
public class Test{
    public static void main(String[] args){
        Person person = new Person("李明",20,'男');
        System.out.println(person.getName() + "的年龄是" + person.getAge() + "岁");
        Circle circle = new Circle(20.0);
        System.out.println("半径为" + circle.getRadius() + "的圆的面积是" + circle.getArea
());
    }
}
```

如上程序中的第 1 条语句使用关键字 import 导入 com.MyClass 包中所有的类,这样在程序中就可以使用第(3)步中打包的 Person 类和 Circle 类了。

编译并运行 Test 类,例 5.15 程序的运行结果如图 5.20 所示。

```
李明的年龄是20岁
半径为20.0的圆的面积是1256.0
```

图 5.20　例 5.15 程序的运行结果

如果使用 Eclipse 开发 Java 程序,则打包过程比较简单,只要在新建类时指定包名,Eclipse 会自动为该类添加打包语句,并根据包名创建包的目录结构。例如,如果指定的包名是 com.MyClass,则 Eclipse 会在程序所在目录的 src 子目录下自动创建 com\MyClass 两级子目录,并且把当前类放到该目录里;而编译之后生成的字节码文件位于程序所在目录的 bin\com\MyClass 子目录中。

那么如何在使用 Eclipse 开发 Java 程序时,使用已经打包好的类呢?下面以实例说明。

【例 5.16】 在 Eclipse 中使用本节前面打包的 Person 类和 Circle 类。

新建 Eclipse 工程 Exp5_16,在“Package Explorer(包资源管理器视图)”中右击该工程,在弹出的菜单中执行 Build Path(构建路径)→Configure Build Path(配置构建路径)菜单命令打开工程属性对话框,如图 5.21 所示。

在工程属性对话框中,选择 Java Build Path(Java 构建路径)→Libraries(库)→Classpath→Add External Class Folder(添加外部类文件夹)命令,打开 External Class Folder Selection(选择外部类文件夹)对话框,在其中选择 D 盘的 class 子目录(包 com.

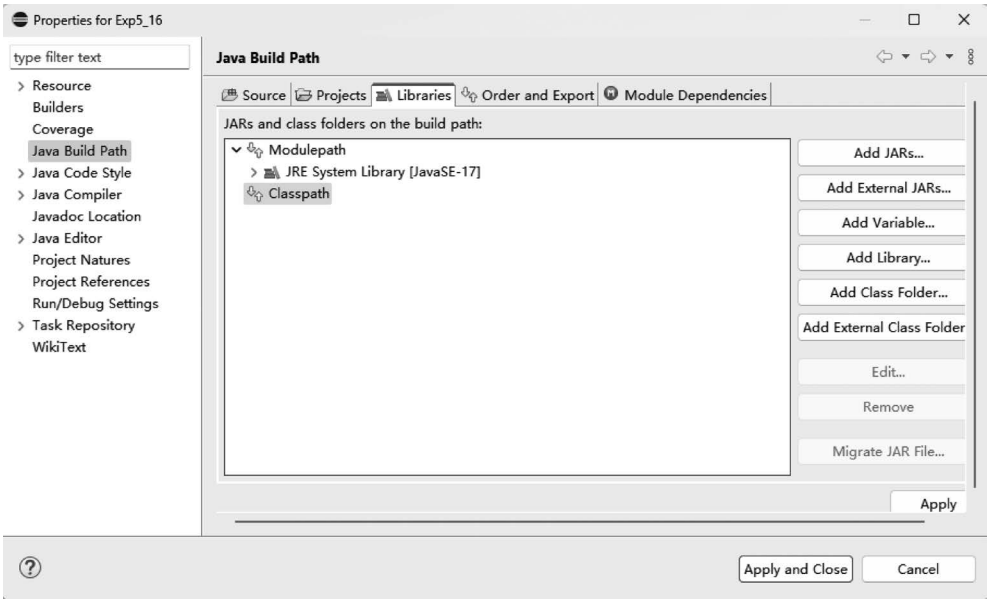


图 5.21　工程属性对话框

MyClass 所在的文件夹），如图 5.22 所示。

图 5.22　选择包所在的文件夹

单击“确定”按钮退出 External Class Folder Selection 对话框，在工程属性对话框中单击 Apply and Close 按钮完成配置。现在，程序中已经可以使用包 com.MyClass 中的类了。在工程中新建一个 Test 类进行测试，代码如下：

```
import com.MyClass.*;
public class Test{
    public static void main(String[] args){
```

```
        Person person = new Person("李明",20,'男');
        System.out.println(person.getName() + "的年龄是" + person.getAge() + "岁");
        Circle circle = new Circle(20.0);
        System.out.println("半径为" + circle.getRadius() + "的圆的面积是" + circle.getArea());
    }
}
```

运行如上程序,运行结果如图 5.20 所示。

5.9　编程实训

【例 5.17】 栈是一种“后进先出”的线性数据结构,要求创建并使用一个用于存放整数的栈类 Stack,当栈满时,可以自动为其追加容量。

在程序的 main()方法中,创建一个 Stack 对象,然后把整数 1～20 存入栈中,再逐一弹出所有栈中的元素。

Stack 类的代码如下:

```
//Stack.java
package com.MyClass;

public class Stack {
    private int[] elements;                 //保存栈中元素的数组
    private int numbers;                    //保存栈中元素的个数
    public Stack(){                         //默认构造方法,创建初始容量为 10 的栈
        elements = new int[10];
        numbers = 0;
    }
    public Stack(int size){                 //创建容量为 size 的栈
        elements = new int[size];
        numbers = 0;
    }
    private void append(){                  //此方法用来追加栈的容量
        int size = elements.length + 10;
        int[] temp = new int[size];
        for(int i = 0;i < elements.length;i++){
            temp[i] = elements[i];
        }
        elements = temp;
    }
    public void push(int e){                //元素入栈的方法
        if(numbers < elements.length)
            elements[numbers++] = e;
        else{
            append();
            elements[numbers++] = e;
        }
    }
    public int peek(){                      //返回栈顶元素的方法
        return elements[numbers - 1];
    }
    public int pop(){                       //弹出并返回栈顶元素
        return elements[ --numbers];
```

```
    }
    public boolean isEmpty(){              //判断栈是否为空
        return numbers == 0;
    }
    public int getNumbers(){               //返回栈中元素的个数
        return numbers;
    }
}
```

Stack 类中的 push()方法用来实现元素入栈，pop()方法弹出并返回栈顶元素，peek()方法返回栈顶元素，append()方法可以在栈满时扩充栈的容量，isEmpty()方法用于判断栈是否为空，getNumbers()方法用于返回栈中元素的个数。以下是使用栈的程序主类——Test 类。

```
//Test.java
import com.MyClass.Stack;
public class Test {
    public static void main(String[] args) {
        Stack s1 = new Stack();            //创建一个初始容量为 10 的栈 s1
        for(int i = 0;i < 20;i++){
            s1.push(i + 1);                //把整数 i + 1 存入栈中
        }
        System.out.println("栈中共有" + s1.getNumbers() + "个元素");
        while(!s1.isEmpty()){
            System.out.println("栈顶元素" + s1.pop() + "出栈");
        }
    }
}
```

例 5.17 程序的运行结果如图 5.23 所示。

```
栈中共有20个元素
栈顶元素20出栈
栈顶元素19出栈
栈顶元素18出栈
栈顶元素17出栈
栈顶元素16出栈
栈顶元素15出栈
栈顶元素14出栈
栈顶元素13出栈
栈顶元素12出栈
栈顶元素11出栈
栈顶元素10出栈
栈顶元素9出栈
栈顶元素8出栈
栈顶元素7出栈
栈顶元素6出栈
栈顶元素5出栈
栈顶元素4出栈
栈顶元素3出栈
栈顶元素2出栈
栈顶元素1出栈
```

图 5.23　例 5.17 程序的运行结果

5.10　小结

类的静态成员是属于类的成员，被类的所有实例(对象)共享，类的实例成员是属于某个具体实例(对象)的。

类的静态方法中只能访问类的静态成员，不能访问类的实例成员；类的实例方法中，既能访问类的静态成员，也能访问类的实例成员。

类的实例方法只能用对象调用；类的静态方法既可以用对象调用，也可以用类名调用。

关键字 final 用来声明常量。

关键字 this 只能出现在类的实例方法中，引用调用此方法的当前对象。

数组是相同类型的元素的集合，Java 的数组是一个对象。

Java 的递归方法可以用来解决递归问题。

Java 的方法可以处理可变长参数，可变长参数在方法内部用数组来处理。

包是一种避免类名冲突的机制，Java 中所有的类都位于某个包中。可以使用关键字 package 为类打包，关键字 import 用于导入包中的类。