


第 9 章

类的高级特性

( 视频讲解：34 分钟)

类除了具有普通的特性之外，还具有一些高级特性，如包、内部类等。包在整个管理中起到了非常重要的作用，使用包可以有效地管理繁杂的类文件，解决类重名问题，当在类中配合包与权限修饰符使用时，可以控制其他人对类成员的访问。同时在 Java 语言中一个更为有效的隐藏实现细节的技巧是使用内部类，通过使用内部类机制可以向上转型为被内部类实现的公共接口。由于在类中可以定义多个内部类，所以实现接口的方式也不止一个，只要将内部类中的方法设置为类最小范围的修饰权限即可将内部类的实现细节有效地隐藏。

通过阅读本章，您可以：

- » 掌握抽象类的使用
- » 掌握内部类的分类
- » 掌握成员内部类的使用
- » 掌握局部内部类的使用
- » 掌握匿名内部类的使用
- » 掌握静态内部类的使用
- » 掌握内部类的继承
- » 掌握 Class 类与 Java 反射



9.1 抽 象 类

 视频讲解：光盘\TM\1\9\抽象类.exe

所谓抽象类就是只声明方法的存在而不去具体实现它的类。抽象类不能被实例化，也就是不能创建其对象。在定义抽象类时，要在 `class` 关键字前面加上 `abstract` 关键字。定义抽象类，语法格式如下：

```
abstract class 类名 {
    类体
}
```

【例 9.1】 定义一个名称为 `Fruit` 的抽象类。

```
abstract class Fruit {                                //定义抽象类
    public String color;                             //定义颜色成员变量
    //定义构造方法
    public Fruit() {
        color="绿色";                               //对变量 color 进行初始化
    }
}
```

在抽象类中创建的，没有实际意义的，必须要子类重写的方法称为抽象方法。抽象方法只有方法的声明，而没有方法的实现，用 `abstract` 关键字进行修饰。声明一个抽象方法的基本格式如下：

```
abstract <方法返回值类型> 方法名(参数列表);
```

- ☑ 方法返回值类型：必选参数，用于指定方法的返回值类型，如果该方法没有返回值，可以使用关键字 `void` 进行标识。方法返回值的类型可以是任何 Java 数据类型。
- ☑ 方法名：必选参数，用于指定抽象方法的名称，方法名必须是合法的 Java 标识符。
- ☑ 参数列表：可选参数，用于指定方法中所需的参数。当存在多个参数时，各参数之间应使用逗号分隔。方法的参数可以是任何 Java 数据类型。

例如，在例 9.1 中定义的抽象类中添加一个抽象方法，代码如下：

```
//定义抽象方法
public abstract void harvest();                      //收获的方法
```

注意

抽象方法不能使用 `private` 或 `static` 关键字进行修饰。

包含一个或多个抽象方法的类必须被声明为抽象类。因为抽象方法没有定义方法的实现部分，如果不声明为抽象类，这个类将可以生成对象，这时当用户调用抽象方法时，程序就不知道如何处理了。

【例 9.2】 本实例主要实现定义一个水果类 `Fruit`，该类为抽象类，并在该类中定义一个抽象方法，同时在其子类中通过重写的方法实现该抽象方法。（实例位置：光盘\TM\1\9\1）

(1) 使用 `abstract` 关键字创建抽象类 `Fruit`，在该类中定义相应的变量和方法。代码如下：

```
abstract class Fruit {                                //定义抽象类
    public String color;                             //定义颜色成员变量
    //定义构造方法
    public Fruit(){
        color="绿色";                               //对变量 color 进行初始化
    }
    //定义抽象方法
    public abstract void harvest();                  //收获的方法
}
```

(2) 创建 `Fruit` 类的子类 `Apple`，并实现 `harvest()` 方法。代码如下：

```
public class Apple extends Fruit {
    @Override
    public void harvest() {
        System.out.println("苹果已经收获！");        //输出字符串“苹果已经收获！”
    }
}
```

(3) 创建一个 `Fruit` 类的子类 `Orange`，并实现 `harvest()` 方法。代码如下：

```
public class Orange extends Fruit {
    @Override
    public void harvest() {
        System.out.println("桔子已经收获！");        //输出字符串“桔子已经收获！”
    }
}
```

(4) 创建一个包含 `main()` 方法的公共类 `Farm`，在该类中执行 `Fruit` 类的两个子类的 `harvest()` 方法。代码如下：

```
public class Farm {
    public static void main(String[] args) {
        System.out.println("调用 Apple 类的 harvest() 方法的结果：");
        Apple apple=new Apple();                //声明 Apple 类的一个对象 apple，并为其分配内存
        apple.harvest();                          //调用 Apple 类的 harvest() 方法
        System.out.println("调用 Orange 类的 harvest() 方法的结果：");
        Orange orange=new Orange();              //声明 Orange 类的一个对象 orange，并为其分配内存
        orange.harvest();                        //调用 Orange 类的 harvest() 方法
    }
}
```

运行结果如图 9.1 所示。



图 9.1 定义抽象类及抽象方法

9.2 内部类

 视频讲解：光盘\TM\lx\9\内部类.exe

如果在一个类中再定义一个类，就将在类中再定义的那个类称为内部类。内部类可分为成员内部类、局部内部类以及匿名内部类。

9.2.1 成员内部类

1. 成员内部类简介

在一个类中使用内部类可以在内部类中直接存取其所在类的私有成员变量。本节首先介绍成员内部类。成员内部类的语法格式如下：

```
public class OuterClass{                               //外部类
    private class InnerClass{                          //内部类
        //...
    }
}
```

在内部类中可以随意使用外部类的成员方法以及成员变量，尽管这些类成员被修饰为 `private`。图 9.2 充分说明了内部类的使用，尽管成员变量 `i` 以及成员方法 `g()` 都在外部类中被修饰为 `private`，但在内部类中可以直接使用外部类中的类成员。

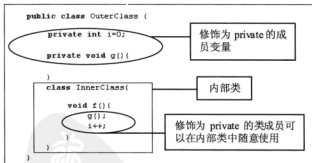


图 9.2 内部类可以使用外部类的成员

内部类的实例一定要绑定在外部类的实例上，如果在外部类中初始化一个内部类对象，那么内部类对象就会绑定在外部类对象上。内部类初始化方式与其他类初始化方式相同，都是使用 `new` 关键字。

【例 9.3】 创建 `OuterClass` 类，在类中定义 `innerClass` 内部类和 `doit()` 方法，在主方法中创建 `OuterClass` 类的实例对象并调用 `doit()` 方法。（实例位置：光盘\TM\lx\9\2）

```

public class OuterClass {
    innerClass in=new innerClass();           //在外部类实例化内部类对象引用
    public void out(){                         //在外部类方法中调用内部类方法
        in.inf();
    }
    class innerClass{
        innerClass(){                         //内部类构造方法
        }
        public void inf(){                   //内部类成员方法
        }
        int y=0;                             //定义内部类成员变量
    }
    public innerClass doit(){                  //外部类方法，返回值为内部类引用
        //y=4;                               //外部类不可以直接访问内部类成员变量
        in.y=4;
        return new innerClass();             //返回内部类引用
    }
    public static void main(String args[]){
        OuterClass out=new OuterClass();
        //内部类的对象实例化操作必须在外部类或外部类中的非静态方法中实现
        OuterClass innerClass in=out.doit();
    }
}

```

例 9.3 中的外部类创建内部类实例时与其他类创建对象引用时相同。内部类可以访问它的外部类的成员，但内部类的成员只有在内部类的范围之内是可知的，不能被外部类使用。例如，在例 9.3 中如果将内部类的成员变量 `y` 再次赋值时将会出错，但是如果使用内部类对象引用调用成员变量 `y` 即可。图 9.3 说明了内部类 `innerClass` 对象与外部类 `OuterClass` 对象的关系。

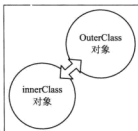


图 9.3 内部类对象与外部类对象的关系

从图 9.3 中可以看出，内部类对象与外部类对象关系非常紧密，内外可以交互使用彼此类中定义的变量。

注意 如果在外部类和非静态方法之外实例化内部类对象，需要使用“外部类.内部类”的形式指定该对象的类型。

在例 9.3 的主方法中如果不使用 `doit()` 方法返回内部类对象引用，可以直接使用内部类实例化内部

类对象,但由于是在主方法中实例化内部类对象,必须在 new 操作符之前提供一个外部类的引用。

【例 9.4】 在主方法中实例化一个内部类对象。

```
public static void main(String args[]){
    OuterClass out=new OuterClass();
    OuterClass.InnerClass in2=out.new InnerClass();    //实例化内部类对象
}
```

在实例化内部类对象时,不能在 new 操作符之前使用外部类名称那种形式实例化内部类对象,而是应该使用外部类的对象来创建其内部类的对象。

注意 内部类对象会依赖于外部类对象,除非已经存在一个外部类对象,否则类中不会出现内部类对象。

2. 使用 this 关键字获取内部类与外部类的引用

如果在外部类中定义的成员变量与内部类的成员变量名称相同,可以使用 this 关键字。

【例 9.5】 创建 TheSameName 类,在类中定义成员变量 x,定义一个内部类 Inner,并在内部类中也创建 x 变量,在内部类的 doit()方法中分别操作两个 x 变量。(实例位置:光盘\TM\9\3)

```
public class TheSameName {
    private int x;
    private class Inner{
        private int x=9;
        public void doit(int x){
            x++;
            this.x++;
            TheSameName.this.x++;
        }
    }
}
```

//调用的是形参 x
//调用内部类的变量 x
//调用外部类的变量 x

在类中如果内部类与外部类遇到成员变量重名的情况可以使用 this 关键字进行处理,例如在内部类中使用 this.x 语句可以调用内部类的成员变量 x,而使用 TheSameName.this.x 语句可以调用外部类的成员变量 x,即使用外部类名称后跟一个点操作符和 this 关键字便可获取外部类的一个引用。

读者应该明确一点,在内存中所有对象被放置在堆中,将方法以及方法中的形参或局部变量放置在栈中,如图 9.4 所示。在栈中的 doit()方法指向内部类的对象,而内部类的对象与外部类的

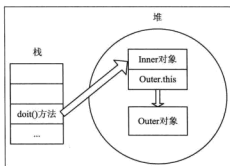


图 9.4 内部类对象与外部类对象在内存中的分布情况

对象是相互依赖的，Outer.this 对象指向外部类对象。

9.2.2 局部内部类

局部内部类是指在类的方法中定义的内部类，它的作用范围也是在这个方法体内。下面将通过一个具体的例子来说明如何定义局部内部类。

【例 9.6】 在外部类的 sell()方法中创建 Apple 局部内部类，然后创建该内部类的实例，并调用其定义的 price()方法输出单价信息。（实例位置：光盘\TM\9\4）

```
public class SellOutClass {
    private String name;           //私有成员变量
    public SellOutClass() {        //构造方法
        name = "苹果";
    }
    public void sell(int price) {
        class Apple {              //局部内部类
            int innerPrice = 0;
            public Apple(int price) { //构造方法
                innerPrice = price;
            }
            public void price() {     //方法
                System.out.println("现在开始销售" + name);
                System.out.println("单价为: " + innerPrice + "元");
            }
        }
        Apple apple = new Apple(price); //实例化 Apple 类的对象
        apple.price();                  //调用局部内部类的方法
    }
    public static void main(String[] args) {
        SellOutClass sample = new SellOutClass(); //实例化 SellOutClass 类的对象
        sample.sell(100);                       //调用 SellOutClass 类的 sell()方法
    }
}
```

运行结果如图 9.5 所示。

在上述代码中可以看到，笔者将内部类定义在 sell()方法内部。但是有一点值得注意，内部类 Apple 是 sell()方法的一部分，并非 SellOutClass 类的一部分，所以在 sell()方法的外部不能访问该内部类，但是该内部类可以访问当前代码块的常量以及此外部类的所有成员。

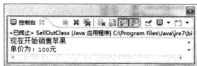


图 9.5 通过调用局部内部类输出单价信息

9.2.3 匿名内部类

在编写程序代码时，不一定要给内部类取一个名字，可以直接以对象名来代替。匿名内部类的所

有实现代码都需要在大括号之间进行编写。



说明

在图形化编程的事件监控器代码中，会大量使用匿名内部类，这样可以大大简化代码，并增强代码的可读性。

匿名内部类的语法格式如下：

```
return new A() {
    ...//内部类体
};
```

其中，A 表示对象名。

由于匿名内部类没有名称，所以匿名内部类使用默认构造方法来生成匿名内部类的对象。在匿名内部类定义结束后，需要加分号标识，这个分号并不代表定义内部类结束的标识，而代表创建匿名内部类的引用表达式的标识。



说明

匿名内部类编译以后，会产生以“外部类名\$序号”为名称的.class 文件，序号以 1~n 排列，分别代表 1~n 个匿名内部类。

【例 9.7】 在 main() 方法中编写匿名内部类去除字符串中的全部空格。(实例位置：光盘\TM\9\5)

```
public class OutString {
    public static void main(String[] args) {
        final String sourceStr = "吉林省 明日 科技有限公司——编程 词典！";
        IStringDeal s = new IStringDeal() {                //编写匿名内部类
            @Override
            public String filterBlankChar() {
                String convertStr = sourceStr;
                convertStr = convertStr.replaceAll(" ", "");    //替换全部空格
                return convertStr;                                //返回转换后的字符串
            }
        };
        System.out.println("源字符串：" + sourceStr);          //输出源字符串
        System.out.println("转换后的字符串：" + s.filterBlankChar()); //输出转换后的字符串
    }
}
```

编写接口 IStringDeal，在该接口中声明一个方法 filterBlankChar()。代码如下：

```
public interface IStringDeal {
    public String filterBlankChar();    //声明过滤字符串中的空格的方法
}
```

运行结果如图 9.6 所示。

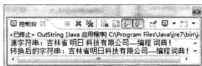


图 9.6 去除字符串中的全部空格

9.2.4 静态内部类

在内部类前添加修饰符 `static`，这个内部类就变为静态内部类。一个静态内部类中可以声明 `static` 成员，但是在非静态内部类中不可以声明静态成员。静态内部类有一个最大的特点，就是不可以使用外部的非静态成员，所以静态内部类在程序开发中比较少见。

可以这样认为，普通的内部类对象隐式地在外部保存了一个引用，指向创建它的外部类对象，但如果内部类被定义为 `static` 时，它应该具有更多的限制。静态内部类具有以下两个特点：

- ☑ 创建静态内部类的对象，不需要其外部类的对象。
- ☑ 不能从静态内部类的对象中访问非静态外部类的对象。

【例 9.8】 定义一个静态内部类 `StaticInnerClass`。

```
public class StaticInnerClass {
    int x=100;
    static class Inner{
        void doitInner(){
            System.out.println("外部类"+x);           //调用外部的成员变量 x
        }
    }
}
```

在例 9.8 中，在内部类的 `doitInner()` 方法中调用成员变量 `x`，由于 `Inner` 被修饰为 `static` 形式，而成员变量 `x` 却是非 `static` 类型的，所以在 `doitInner()` 方法中不能调用 `x` 变量。

进行程序测试时，如果在每一个 Java 文件中都设置一个主方法，将出现很多额外代码，而程序本身并不需要这些主方法，为了解决这个问题，可以将主方法写入静态内部类中。

【例 9.9】 在静态内部类中定义主方法，并访问内部类中的方法。（实例位置：光盘\TM\st\9\6）

```
public class StaticInnerClass {
    static int x=100;
    static class Inner{
        static void doitInner(){
            System.out.println("外部类的成员变量"+x);           //调用外部的成员变量 x
        }
        public static void main(String args[]){
            doitInner();                                           //定义主方法
                                                                //访问内部类的方法
        }
    }
}
```

运行结果如图 9.7 所示。

如果编译例 9.9 中的类, 将编译生成一个名称为 `StaticInnerClass$Inner` 的独立类和一个 `StaticInnerClass` 类, 只要使用 `java StaticInnerClass$Inner` 就可以运行主方法中的内容, 这样当测试完成需要将所有 `.class` 文件打包时, 只要删除 `StaticInnerClass$Inner` 独立类即可。



图 9.7 访问静态内部类中的方法

9.2.5 内部类的继承

内部类也和其他普通类一样可以被继承, 但是继承内部类比继承普通类要复杂一些, 需要设置专门的语法来完成。

【例 9.10】 创建 `OutputInnerClass` 类, 使 `OutputInnerClass` 类继承 `ClassA` 类中的内部类 `ClassB`。

```
public class OutputInnerClass extends ClassA.ClassB {           //继承内部类 ClassB
    public OutputInnerClass(ClassA a) {
        a.super();
    }
}
class ClassA {
    class ClassB {
    }
}
```

在某个类继承内部类时, 必须硬性给予这个类一个带参数的构造方法, 并且该构造方法的参数为需要继承内部类的外部类的引用, 同时在构造方法体中使用 `a.super()` 语句, 这样才为继承提供了必要的对象引用。

9.2.6 范例 1: 局部内部类设置闹钟

日常生活中, 闹钟的应用非常广泛, 使用它可以更好地帮助人们安排时间。本范例将实现一个非常简单的闹钟。运行本范例, 控制台会不断输出当前的时间, 并且每隔一秒钟会发出提示音。用户可以单击“确定”按钮来退出程序。运行结果如图 9.8 所示。

(实例位置: 光盘\TM\9\9.7)

在项目中创建 `AlarmClock` 类, 在该类中首先定义两个属性, 一个是 `delay`, 表示延迟的时间; 另一个是 `flag`, 表示是否需要发出提示声音。然后在 `start()` 方法中使用 `Timer` 类来安排动作发出事件。代码如下:

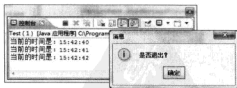


图 9.8 局部内部类设置闹钟

```
public class AlarmClock {
    private int delay;                                           //表示延迟时间
```

```

private boolean flag;                                //表示是否要发出声音
public AlarmClock(int delay, boolean flag) {          //使用构造方法初始化各个域
    this.delay = delay;
    this.flag = flag;
}
public void start() {
    class Printer implements ActionListener {          //定义内部类实现动作监听接口
        @Override
        public void actionPerformed(ActionEvent e) {
            SimpleDateFormat format = new SimpleDateFormat("k:m:s");//定义时间的格式
            String result = format.format(new Date());          //获得当前的时间
            System.out.println("当前的时间是: " + result);      //显示当前的时间
            if (flag) {                                         //根据 flag 来决定是否要发出声音
                Toolkit.getDefaultToolkit().beep();
            }
        }
    }
    new Timer(delay, new Printer()).start();           //创建 Timer 对象并启动
}
}

```

9.2.7 范例 2: 静态内部类求极值

当对元素进行排序时,需要明确各个元素如何比较大小。使用既定的比较方式,就可以求出一个数组中的最大值和最小值。本范例使用静态内部类来实现使用一次遍历求最大和最小值。运行结果如图 9.9 所示。(实例位置:光盘\TM\sl\9\8)

(1) 在项目中创建 MaxMin 类,在该类中首先定义一个静态内部类 Result。然后在 Result 类中定义两个浮点型属性,一个是 max,表示最大值;另一个是 min,表示最小值。再使用构造方法为其初始化,并提供 getXXX()方法来获得这两个值。最后定义一个静态方法 getResult(),该方法的返回值是 Result 类型,这样就可以既保存最大值,又保存最小值了。代码如下:

```

public class MaxMin {
    public static class Result {
        private double max;
        private double min;
        public Result(double max, double min) {
            this.max = max;
            this.min = min;
        }
        public double getMax() {
            return max;
        }
    }
    //表示最大值
    //表示最小值
    //使用构造方法进行初始化
    //获得最大值
}

```



图 9.9 求数组中的最大值和最小值

```

    }
    public double getMin() {
        return min;
    }
}
public static Result getResult(double[] array) {
    double max = Double.MIN_VALUE;
    double min = Double.MAX_VALUE;
    for (double i : array) {
        if (i > max) {
            max = i;
        }
        if (i < min) {
            min = i;
        }
    }
    return new Result(max, min);
}
}

```

//获得最小值

//遍历数组获得最大值和最小值

//返回 Result 对象

(2) 在项目中创建 Test 类进行测试，该类的主方法中，使用随机数初始化了一个长度为 5 的数组，并求得该数组的最大值和最小值。代码如下：

```

public class Test {
    public static void main(String[] args) {
        double[] array = new double[5];
        for (int i = 0; i < array.length; i++) {
            array[i] = 100*Math.random();
        }
        System.out.println("源数组: ");
        for (int i = 0; i < array.length; i++) {
            System.out.println(array[i]);
        }
        System.out.println("最大值: " + MaxMin.getResult(array).getMax());
        System.out.println("最小值: " + MaxMin.getResult(array).getMin());
    }
}

```

//初始化数组

//显示数组中的各个元素

//显示最大值

//显示最小值

9.3 Class 类与 Java 反射

 视频讲解：光盘\TM\1\9\Class 类与 Java 反射.exe

通过 Java 反射机制，可以在程序中访问已经装载到 JVM 中的 Java 对象的描述，实现访问、检测和修改描述 Java 对象本身信息的功能。Java 反射机制的功能十分强大，在 java.lang.reflect 包中提供了对该功能的支持。

众所周知，所有 Java 类均继承了 Object 类，在 Object 类中定义了一个 getClass() 方法，该方法返

回一个类型为 Class 的对象。代码如下：

```
Class textFieldC = textField.getClass(); //textField 为 JTextField 类的对象
```

利用 Class 类的对象 textFieldC，可以访问用来返回该对象的 textField 对象的描述信息。主要的描述信息如表 9.1 所示。

表 9.1 通过反射可访问的主要描述信息

组 成 部 分	访 问 方 法	返回值类型	说 明
包路径	getPackage()	Package 对象	获得该类的存放路径
类名称	getName()	String 对象	获得该类的名称
继承类	getSuperclass()	Class 对象	获得该类继承的类
实现接口	getInterfaces()	Class 型数组	获得该类实现的所有接口
构造方法	getConstructors()	Constructor 型数组	获得所有权限为 public 的构造方法
	getConstructor(Class<?>... parameterTypes)	Constructor 对象	获得权限为 public 的指定构造方法
	getDeclaredConstructors()	Constructor 型数组	获得所有构造方法，按声明顺序返回
	getDeclaredConstructor(Class<?>... parameterTypes)	Constructor 对象	获得指定构造方法
方法	getMethods()	Method 型数组	获得所有权限为 public 的方法
	getMethod(String name, Class<?>... parameterTypes)	Method 对象	获得权限为 public 的指定方法
	getDeclaredMethods()	Method 型数组	获得所有方法，按声明顺序返回
	getDeclaredMethod(String name, Class<?>... parameterTypes)	Method 对象	获得指定方法
成员变量	getFields()	Field 型数组	获得所有权限为 public 的成员变量
	getField(String name)	Field 对象	获得权限为 public 的指定成员变量
	getDeclaredFields()	Field 型数组	获得所有成员变量，按声明顺序返回
	getDeclaredField(String name)	Field 对象	获得指定成员变量
内部类	getClasses()	Class 型数组	获得所有权限为 public 的内部类
	getDeclaredClasses()	Class 型数组	获得所有内部类
内部类的声明类	getDeclaringClass()	Class 对象	如果该类为内部类则返回它的成员类，否则返回 null

说明

在通过 getFields()和 getMethods()方法依次获得权限为 public 的成员变量和方法时，还包含从超类中继承到的成员变量和方法；而通过 getDeclaredFields()和 getDeclaredMethods()方法只是获得在本类中定义的所有成员变量和方法。

9.3.1 访问构造方法

在通过下列一组方法访问构造方法时，将返回 Constructor 类型的对象或数组。每个 Constructor 对象代表一个构造方法，利用 Constructor 对象可以操纵相应的构造方法。

- ☒ `getConstructors()`
- ☒ `getConstructor(Class<?>... parameterTypes)`
- ☒ `getDeclaredConstructors()`
- ☒ `getDeclaredConstructor(Class<?>... parameterTypes)`

如果是访问指定的构造方法，需要根据该构造方法的入口参数的类型来访问。例如访问一个入口参数类型依次为 `String` 和 `int` 型的构造方法，可以通过下面两种方式实现。

```
objectClass.getDeclaredConstructor(String.class, int.class);
objectClass.getDeclaredConstructor(new Class[] { String.class, int.class });
```

`Constructor` 类中提供的常用方法如表 9.2 所示。

表 9.2 `Constructor` 类中的常用方法

方 法	说 明
<code>isVarArgs()</code>	查看该构造方法是否允许带有可变数量的参数，如果允许则返回 <code>true</code> ，否则返回 <code>false</code>
<code>getParameterTypes()</code>	按照声明顺序以 <code>Class</code> 数组的形式获得该构造方法的各个参数的类型
<code>getExceptionTypes()</code>	以 <code>Class</code> 数组的形式获得该构造方法可能抛出的异常类型
<code>newInstance(Object... initargs)</code>	通过该构造方法利用指定参数创建一个该类的对象，如果未设置参数则表示采用默认无参数的构造方法
<code>setAccessible(boolean flag)</code>	如果该构造方法的权限为 <code>private</code> ，默认为不允许通过反射利用 <code>newInstance(Object... initargs)</code> 方法创建对象。如果先执行该方法，并将入口参数设为 <code>true</code> ，则允许创建
<code>getModifiers()</code>	获得可以解析出该构造方法所采用修饰符的整数

通过 `java.lang.reflect.Modifier` 类可以解析出 `getModifiers()` 方法的返回值所表示的修饰符信息，在该类中提供了一系列用来解析的静态方法，既可以查看是否被指定的修饰符修饰，还可以以字符串的形式获得所有修饰符。该类常用的静态方法如表 9.3 所示。

表 9.3 `Modifier` 类中的常用静态方法

静 态 方 法	说 明
<code>isPublic(int mod)</code>	查看是否被 <code>public</code> 修饰符修饰，如果是则返回 <code>true</code> ，否则返回 <code>false</code>
<code>isProtected(int mod)</code>	查看是否被 <code>protected</code> 修饰符修饰，如果是则返回 <code>true</code> ，否则返回 <code>false</code>
<code>isPrivate(int mod)</code>	查看是否被 <code>private</code> 修饰符修饰，如果是则返回 <code>true</code> ，否则返回 <code>false</code>
<code>isStatic(int mod)</code>	查看是否被 <code>static</code> 修饰符修饰，如果是则返回 <code>true</code> ，否则返回 <code>false</code>
<code>isFinal(int mod)</code>	查看是否被 <code>final</code> 修饰符修饰，如果是则返回 <code>true</code> ，否则返回 <code>false</code>
<code>toString(int mod)</code>	以字符串的形式返回所有修饰符

例如，判断 `constructor` 对象所代表的构造方法是否被 `private` 修饰，以及以字符串形式获得该构造方法的所有修饰符。代码如下：

```
int modifiers = constructor.getModifiers();
boolean isEmbellishByPrivate = Modifier.isPrivate(modifiers);
String embellishment = Modifier.toString(modifiers);
```

【例 9.11】 访问构造方法。（实例位置：光盘\TM\sl\9\9）

（1）在项目中创建一个 `MoreConstructor` 类，在该类中声明一个 `String` 型成员变量和 3 个 `int` 型成

员变量，并提供3个构造方法。代码如下：

```
public class MoreConstructor {
    String s;
    int i, i2, i3;
    private MoreConstructor() {                //不带参数的构造方法
    }
    protected MoreConstructor(String s, int i) {    //带两个参数的构造方法
        this.s = s;
        this.i = i;
    }
    //带可变参数的构造方法
    public MoreConstructor(String... strings) throws NumberFormatException {
        if (0 < strings.length)
            i = Integer.valueOf(strings[0]);
        if (1 < strings.length)
            i2 = Integer.valueOf(strings[1]);
        if (2 < strings.length)
            i3 = Integer.valueOf(strings[2]);
    }
    public void print() {
        System.out.println("s=" + s);
        System.out.println("i=" + i);
        System.out.println("i2=" + i2);
        System.out.println("i3=" + i3);
    }
}
```

(2) 编写测试类 `AccessConstructor`，在该类中通过反射访问 `MoreConstructor` 类中的所有构造方法，并将该构造方法是否允许带有可变数量的参数、入口参数类型和可能抛出的异常类型信息输出到控制台。代码如下：

```
Constructor[] declaredConstructors = exampleC.getDeclaredConstructors(); //获得所有构造方法
for (int i = 0; i < declaredConstructors.length; i++) {
    Constructor constructor = declaredConstructors[i];                //遍历构造方法
    System.out.println("查看是否允许带有可变数量的参数: " + constructor.isVarArgs());
    System.out.println("该构造方法的入口参数类型依次为: ");
    Class[] parameterTypes = constructor.getParameterTypes();        //获得所有参数类型
    for (int j = 0; j < parameterTypes.length; j++) {
        System.out.println(" " + parameterTypes[j]);
    }
    System.out.println("该构造方法可能抛出的异常类型为: ");
    //获得所有可能抛出的异常信息类型
    Class[] exceptionTypes = constructor.getExceptionTypes();
    for (int j = 0; j < exceptionTypes.length; j++) {
        System.out.println(" " + exceptionTypes[j]);
    }
    MoreConstructor example2 = null;
    while (example2 == null) {
        //如果该成员变量的访问权限为 private，则抛出异常，即不允许访问
    }
}
```

```

try {
    if (i == 0) //通过执行默认没有参数的构造方法创建对象
        example2 = (MoreConstructor) constructor.newInstance();
    else if (i == 1) //通过执行具有两个参数的构造方法创建对象
        example2 = (MoreConstructor) constructor.newInstance("7", 5);
    else { //通过执行具有可变数量参数的构造方法创建对象
        Object[] parameters = new Object[] { new String[] { "100", "200", "300" } };
        example2 = (MoreConstructor) constructor.newInstance(parameters);
    }
} catch (Exception e) {
    System.out.println("在创建对象时抛出异常, 下面执行 setAccessible()方法");
    constructor.setAccessible(true); //设置为允许访问
}
}
example2.print();
System.out.println();
}
}

```

运行本例, 当通过反射访问构造方法 `MoreConstructor()` 时, 输出信息如图 9.10 所示。

当通过反射访问构造方法 `MoreConstructor(String s, int i)` 时, 输出信息如图 9.11 所示。



图 9.10 访问 `MoreConstructor()` 输出的信息

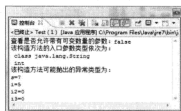


图 9.11 访问 `MoreConstructor(String s, int i)` 输出的信息

当通过反射访问构造方法 `MoreConstructor(String... strings)` 时, 输出信息如图 9.12 所示。



图 9.12 访问 `MoreConstructor(String... strings)` 输出的信息

9.3.2 访问成员变量

在通过下列一组方法访问成员变量时, 将返回 `Field` 类型的对象或数组。每个 `Field` 对象代表一个成员变量, 利用 `Field` 对象可以操纵相应的成员变量。

- ☒ `getFields()`

- ☒ getField(String name)
- ☒ getDeclaredFields()
- ☒ getDeclaredField(String name)

如果是访问指定的成员变量，可以通过该成员变量的名称来访问。例如访问一个名称为 birthday 的成员变量，代码如下：

```
object.getDeclaredField("birthday");
```

Field 类中提供的常用方法如表 9.4 所示。

表 9.4 Field 类中的常用方法

方 法	说 明
getName()	获得该成员变量的名称
getType()	获得表示该成员变量类型的 Class 对象
get(Object obj)	获得指定对象 obj 中该成员变量的值，返回值为 Object 型
set(Object obj, Object value)	将指定对象 obj 中该成员变量的值设置为 value
getInt(Object obj)	获得指定对象 obj 中类型为 int 的该成员变量的值
setInt(Object obj, int i)	将指定对象 obj 中类型为 int 的该成员变量的值设置为 i
getFloat(Object obj)	获得指定对象 obj 中类型为 float 的该成员变量的值
setFloat(Object obj, float f)	将指定对象 obj 中类型为 float 的该成员变量的值设置为 f
getBoolean(Object obj)	获得指定对象 obj 中类型为 boolean 的该成员变量的值
setBoolean(Object obj, boolean z)	将指定对象 obj 中类型为 boolean 的该成员变量的值设置为 z
setAccessible(boolean flag)	如果该构造方法的权限为 private，默认为不允许通过反射利用 newInstance(Object... initargs)方法创建对象。如果先执行该方法，并将入口参数设为 true，则允许创建
getModifiers()	获得可以解析出该成员变量所采用修饰符的整数

【例 9.12】 访问成员变量。（实例位置：光盘\TM\sl\9\10）

(1) 在项目中创建一个 MoreFields 类，在该类中依次声明一个 int、float、boolean 和 String 型的成员变量，并将它们设置为不同的访问权限。代码如下：

```
public class MoreFields {
    int i;
    public float f;
    protected boolean b;
    private String s;
}
```

(2) 通过反射访问 MoreFields 类中的所有成员变量，将成员变量的名称和类型信息输出到控制台，并分别将各个成员变量在修改前后的值输出到控制台。代码如下：

```
Field[] declaredFields = exampleC.getDeclaredFields();           //获得所有成员变量
for (int i = 0; i < declaredFields.length; i++) {
    Field field = declaredFields[i];                               //遍历成员变量
    System.out.println("名称为: " + field.getName());             //获得成员变量名称
    Class fieldType = field.getType();                             //获得成员变量类型
    System.out.println("类型为: " + fieldType);
```

```

boolean isTurn = true;
while (isTurn) {
    try { //如果该成员变量的访问权限为 private，则抛出异常，即不允许访问
        isTurn = false;
        System.out.println("修改前的值为: " + field.get(example)); //获得成员变量值
        //判断成员变量的类型是否为 int 型
        if (fieldType.equals(int.class)) {
            System.out.println("利用方法 setInt()修改成员变量的值");
            field.setInt(example, 168); //为 int 型成员变量赋值
            //判断成员变量的类型是否为 float 型
        } else if (fieldType.equals(float.class)) {
            System.out.println("利用方法 setFloat()修改成员变量的值");
            field.setFloat(example, 99.9F); //为 float 型成员变量赋值
            //判断成员变量的类型是否为 boolean 型
        } else if (fieldType.equals(boolean.class)) {
            System.out.println("利用方法 setBoolean()修改成员变量的值");
            field.setBoolean(example, true); //为 boolean 型成员变量赋值
        } else {
            System.out.println("利用方法 set()修改成员变量的值");
            field.set(example, "mingri"); //可以各种类型变量赋值
        }
        System.out.println("修改后的值为: " + field.get(example)); //获得成员变量值
    } catch (Exception e) {
        System.out.println("在设置成员变量值时抛出异常，下面执行 setAccessible()方法!");
        field.setAccessible(true); //设置为允许访问
        isTurn = true;
    }
}
System.out.println();
}

```

运行结果如图 9.13 所示。

从该信息中可以看出，要访问权限为 `private` 的成员变量 `s`，需要执行 `setAccessible()` 方法，并将入口参数设为 `true`，否则不允许访问。

9.3.3 访问方法

在通过下列一组方法访问方法时，将返回 `Method` 类型的对象或数组。每个 `Method` 对象代表一个方法，利用 `Method` 对象可以操纵相应的方法。

- ☒ `getMethods()`
- ☒ `getMethod(String name, Class<?>... parameterTypes)`
- ☒ `getDeclaredMethods()`
- ☒ `getDeclaredMethod(String name, Class<?>... parameterTypes)`

如果是访问指定的方法，需要根据该方法的名称和入口参数的类型来访问。例如访问一个名称为



图 9.13 通过反射访问成员变量

print、入口参数类型依次为 String 和 int 型的方法，可以通过下面两种方式实现。

```
objectClass.getDeclaredMethod("print", String.class, int.class);
objectClass.getDeclaredMethod("print", new Class[] {String.class, int.class });
```

Method 类中提供的常用方法如表 9.5 所示。

表 9.5 Method 类中的常用方法

方 法	说 明
getName()	获得该方法的名称
getParameterTypes()	按照声明顺序以 Class 数组的形式获得该方法的各个参数的类型
getReturnType()	以 Class 对象的形式获得该方法的返回值的类型
getExceptionTypes()	以 Class 数组的形式获得该方法可能抛出的异常类型
invoke(Object obj, Object... args)	利用指定参数 args 执行指定对象 obj 中的该方法，返回值为 Object 型
isVarArgs()	查看该构造方法是否允许带有可变数量的参数，如果允许则返回 true，否则返回 false
getModifiers()	获得可以解析出该方法所采用修饰符的整数

【例 9.13】 访问方法。（实例位置：光盘\TM\sl\9\11）

（1）在项目中创建一个 MoreMethod 类，并编写 4 个典型的方法。代码如下：

```
public class MoreMethod {
    static void staticMethod() {
        System.out.println("执行 staticMethod()方法");
    }
    public int publicMethod(int i) {
        System.out.println("执行 publicMethod()方法");
        return i * 100;
    }
    protected int protectedMethod(String s, int i) throws NumberFormatException {
        System.out.println("执行 protectedMethod()方法");
        return Integer.valueOf(s) + i;
    }
    private String privateMethod(String... strings) {
        System.out.println("执行 privateMethod()方法");
        StringBuffer stringBuffer = new StringBuffer();
        for (int i = 0; i < strings.length; i++) {
            stringBuffer.append(strings[i]);
        }
        return stringBuffer.toString();
    }
}
```

（2）通过反射访问 MoreMethod 类中的所有方法，将各个方法的名称、入口参数类型、返回值类型等信息输出到控制台，并执行部分方法。代码如下：

```
Method[] declaredMethods = exampleC.getDeclaredMethods();           //获得所有方法
for (int i = 0; i < declaredMethods.length; i++) {
    Method method = declaredMethods[i];                               //遍历方法
    System.out.println("名称为: " + method.getName());               //获得方法名称
```

```

System.out.println("是否允许带有可变数量的参数: " + method.isVarArgs());
System.out.println("入口参数类型依次为: ");
Class[] parameterTypes = method.getParameterTypes(); //获得所有参数类型
for (int j = 0; j < parameterTypes.length; j++) {
    System.out.println(" " + parameterTypes[j]);
}
System.out.println("返回值类型为: " + method.getReturnType()); //获得方法返回值类型
System.out.println("可能抛出的异常类型有: ");
//获得方法可能抛出的所有异常类型
Class[] exceptionTypes = method.getExceptionTypes();
for (int j = 0; j < exceptionTypes.length; j++) {
    System.out.println(" " + exceptionTypes[j]);
}
boolean isTurn = true;
while (isTurn) {
    //如果该方法的访问权限为 private, 则抛出异常, 即不允许访问
    try {
        isTurn = false;
        if (i == 0)
            method.invoke(example); //执行没有入口参数的方法
        else if (i == 1)
            System.out.println("返回值为: " + method.invoke(example, 168)); //执行方法
        else if (i == 2)
            System.out.println("返回值为: " + method.invoke(example, "7", 5)); //执行方法
        else {
            Object[] parameters = new Object[] { new String[] { "M", "W", "Q" } }; //定义二维数组
            System.out.println("返回值为: " + method.invoke(example, parameters)); //执行方法
        }
    } catch (Exception e) {
        System.out.println("在执行方法时抛出异常, 下面执行 setAccessible()方法!");
        method.setAccessible(true); //设置为允许访问
        isTurn = true;
    }
}
System.out.println();
}

```

在反射中执行具有可变数量的参数的构造方法时, 需要将入口参数定义成二维数组。运行本例, 访问 `staticMethod()` 方法, 运行结果如图 9.14 所示。

访问 `publicMethod()` 方法, 运行结果如图 9.15 所示。



图 9.14 访问 `staticMethod()` 方法输出的信息

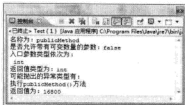


图 9.15 访问 `publicMethod()` 方法输出的信息

访问 `protectedMethod()` 方法, 运行结果如图 9.16 所示。

访问 `privateMethod()` 方法, 运行结果如图 9.17 所示。



图 9.16 访问 `protectedMethod()` 方法输出的信息



图 9.17 访问 `privateMethod()` 方法输出的信息

9.3.4 范例 3: 运用反射查看类的成员

通常类的声明包括常见修饰符 (`public`、`protected`、`private`、`abstract`、`static`、`final` 和 `strictfp` 等)、类的名称、类的泛型参数、类的继承类 (实现的接口) 和类的注解等。本范例将演示如何用反射获得这些信息。运行结果如图 9.18 所示。(实例位置: 光盘\TM\9\9\12)



图 9.13 查看类的成员

在项目中创建 `ClassDeclarationViewer` 类, 在类的主方法中输出与类声明相关的各个项。代码如下:

```
public class ClassDeclarationViewer {
    public static void main(String[] args) throws ClassNotFoundException {
        Class<?> clazz = Class.forName("java.util.ArrayList"); //获得 ArrayList 类对象
        System.out.println("类的标准名称: " + clazz.getCanonicalName());
        System.out.println("类的修饰符: " + Modifier.toString(clazz.getModifiers()));
        //输出类的泛型参数
        TypeVariable<?>[] typeVariables = clazz.getTypeParameters();
        System.out.print("类的泛型参数: ");
        if (typeVariables.length != 0) {
            for (TypeVariable<?> typeVariable : typeVariables) {
                System.out.println(typeVariable + " ");
            }
        } else {

```

```

        System.out.println("空");
    }
    //输出类所实现的所有接口
    Type[] interfaces = clazz.getGenericInterfaces();
    System.out.println("类所实现的接口: ");
    if (interfaces.length != 0) {
        for (Type type : interfaces) {
            System.out.println("I" + type);
        }
    } else {
        System.out.println("I" + "空");
    }
    //输出类的直接继承类，如果是继承自 Object 则返回空
    Type superClass = clazz.getGenericSuperclass();
    System.out.print("类的直接继承类: ");
    if (superClass != null) {
        System.out.println(superClass);
    } else {
        System.out.println("空");
    }
    //输出类的所有注释信息，有些注释信息是不能用反射获得的
    Annotation[] annotations = clazz.getAnnotations();
    System.out.print("类的注解: ");
    if (annotations.length != 0) {
        for (Annotation annotation : annotations) {
            System.out.println("I" + annotation);
        }
    } else {
        System.out.println("空");
    }
}
}
}

```

9.3.5 范例 4：动态调用类中方法

在 Java 语言中，调用类的方法有两种方式：对于静态方法可以直接使用类名调用，对于非静态方法必须使用类的对象调用。反射机制提供了比较另类的调用方式，可以根据需要指定要调用的方法，而不必在编程时确定。调用的方法不仅只限于 public 的，还可以是 private 的。本范例将使用反射机制调用 Math 类的静态方法 sin() 和 String 类的非静态方法 equals()。运行结果如图 9.19 所示。（实例位置：光盘\TM\9\13）

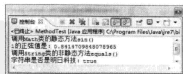


图 9.19 动态调用类中的方法

在项目中创建 MethodTest 类，在该类的主方法中分别调用

了 Math 类的静态方法 sin() 和 String 类的非静态方法 equals()。代码如下：

```
public class MethodTest {
    public static void main(String[] args) {
        try {
            System.out.println("调用 Math 类的静态方法 sin()");
            Method sin = Math.class.getDeclaredMethod("sin", Double.TYPE);
            Double sin1 = (Double) sin.invoke(null, new Integer(1));
            System.out.println("1 的正弦值是: " + sin1);
            System.out.println("调用 String 类的非静态方法 equals()");
            Method equals = String.class.getDeclaredMethod("equals", Object.class);
            Boolean mrsoft = (Boolean) equals.invoke(new String("明日科技"), "明日科技");
            System.out.println("字符串是否是明日科技: " + mrsoft);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

9.4 经典范例

9.4.1 经典范例 1：利用反射重写 toString()方法

 视频讲解：光盘\TM\lx\9\利用反射重写 toString()方法.exe

为了输出对象方便，Object 类提供了 toString()方法。但是该方法的默认值是由类名和哈希码组成的，实用性并不强，通常需要重写该方法以提供更多的信息。本范例主要实现重写类的 toString()方法，用于在调用 toString()方法时，使用反射输出类的包、类的名字、类的公共构造方法、类的公共域和类的公共方法。另外，在重写该方法时，为其传递一个 Object 型的参数，这样可以避免多次重写 toString()方法。运行结果如图 9.20 所示。（实例位置：光盘\TM\sl\9\14）

在项目中创建 StringUtils 类，在该类中定义两个方法，一个是 toString()方法，用于输出类的公共方法和域等信息；另一个是 main()方法，用来进行测试。代码如下：

```
public class StringUtils {
    @SuppressWarnings("unchecked")
    public String toString(Object object) {
        Class clazz = object.getClass();
        StringBuilder sb = new StringBuilder();
        Package packageName = clazz.getPackage();
        sb.append("包名: " + packageName.getName() + "lt");

        //获得代表该类的 Class 对象
        //利用 StringBuilder 来保存字符串
        //获得类所在的包
        //输出类所在的包
    }
}
```

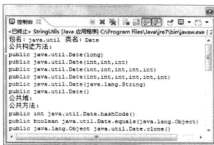


图 9.20 利用反射重写 toString()方法

```

String className = clazz.getSimpleName();           //获得类的简单名称
sb.append("类名: " + className + "\n");           //输出类的简单名称
sb.append("公共构造方法: \n");
//获得所有代表构造方法的 Constructor 数组
Constructor[] constructors = clazz.getDeclaredConstructors();
for (Constructor constructor : constructors) {
    String modifier = Modifier.toString(constructor.getModifiers()); //获得修饰符
    if (modifier.contains("public")) {             //查看修饰符是否含有 public
        sb.append(constructor.toGenericString() + "\n");
    }
}
sb.append("公共域: \n");
Field[] fields = clazz.getDeclaredFields();         //获得代表所有域的 Field[] 数组
for (Field field : fields) {
    String modifier = Modifier.toString(field.getModifiers());
    if (modifier.contains("public")) {             //查看修饰符是否含有 public
        sb.append(field.toGenericString() + "\n");
    }
}
sb.append("公共方法: \n");
Method[] methods = clazz.getDeclaredMethods();     //获得代表所有方法的 Method[] 数组
for (Method method : methods) {
    String modifier = Modifier.toString(method.getModifiers());
    if (modifier.contains("public")) {             //查看修饰符是否含有 public
        sb.append(method.toGenericString() + "\n");
    }
}
return sb.toString();
}
public static void main(String[] args) {
    System.out.println(new StringUtils().toString(new java.util.Date()));
}
}

```

9.4.2 经典范例 2：普通内部类的简单应用

 视频讲解：光盘\TM\lx\9\普通内部类的简单应用.exe

在使用图形界面程序时，用户总是希望界面是丰富多彩的，这就要求程序员根据不同的情况为界面设置不同的颜色。本范例定义了 3 个按钮，用户通过单击不同的按钮，可以为面板设置不同的颜色。运行结果如图 9.21 所示。（实例位置：光盘\TM\lx\9\15）

（1）编写 ButtonTest 类，该类继承了 JFrame。在框架中包含了 3 个按钮，分别用来为面板设置不同的颜色。

（2）编写 ColorAction 类，该类继承自 ActionListener 接口。在该类的构造方法中，需要为其指定一种颜色，在 actionPerformed() 方法中将面板设置成指定的颜色。代码如下：

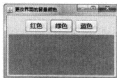


图 9.21 运行结果


```
private class ColorAction implements ActionListener {  
    private Color background;  
    public ColorAction(Color background) {  
        this.background = background;  
    }  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        panel.setBackground(background);  
    }  
}
```

9.5 本章小结

本章首先讲解了抽象类，之后又分别讲解了内部类中的成员内部类、局部内部类、匿名内部类和静态内部类，最后讲解了反射。通过学习本章，读者可以对内部类有一个更深层次的了解，并且能够在编写程序时熟练应用内部类。

9.6 实战练习

1. 编写 Java 程序，创建一个类，并在类中创建一个成员内部类，通过成员内部类计算 1 到任意数的和的操作，并在外部类中进行测试。（答案位置：光盘\TM\9\16）
2. 编写 Java 程序，创建一个类，在该类中定义一个方法，并在方法内部创建一个局部内部类，通过局部内部类求任意两个数的乘积，并在外部类中进行测试。（答案位置：光盘\TM\9\17）
3. 编写 Java 程序，创建一个接口，在该接口中定义一个方法，然后创建一个类，在类中定义一个形参为接口类型的方法，并调用接口中的方法，在类的方法中创建一个实现接口的匿名内部类，用于实现接口中的方法。（答案位置：光盘\TM\9\18）



