第5章

面向对象基础

前面4章是Java语言的基础内容，掌握了之后，就可以实现“贪吃蛇”“俄罗斯方块”等经典小游戏的基础功能。但是随着功能越来越复杂，面临的挑战也越来越大，因此需要学习面向对象相关的内容来应对。在前面的章节中，读者已经接触到了面向对象的思想，接下来的章节将进一步介绍面向对象的思想。

5.1 面向对象概述

面向对象思想是在结构化设计方法出现很多问题的情况下应运而生的。随着问题规模越来越大、应用系统越来越复杂、需求变更越来越频繁，结构化设计方法面临着维护和扩展的困难，甚至一个微小的变动，都会波及整个系统。这驱使人们寻求一种新的程序设计方法，以适应现代社会对软件开发的更高要求，面向对象由此产生。面向对象是一种符合人类思维习惯的编程思想，现实生活中人们习惯将物体进行分类。例如，见到一只狗的时候，第一时间会将其进行归类，柯基、泰迪或者其他，然后根据其类型进行反馈。

面向对象的思想是将所有问题都看作对象，设计程序的时候无论要实现什么功能，首先就是要找到合适的对象。例如，生活中执行洗衣的任务，无论是通过保洁人员，还是通过洗衣机清洗，都是通过具体对象去实现。面向对象将功能封装进对象之中，然后让对象去实现具体的细节，这非常符合人类自然思考的习惯。

面向对象的思想使复杂的事情变得简单化，程序设计者无须关注实现的细节，只需要通过对象去完成即可，从以前的执行者变成指挥者。例如，前面章节实现的“贪吃蛇”游戏中方块移动的功能，程序设计者无须了解方块上、下、左、右运动是如何实现，只需要会使用相应的对象即可。面向对象更利于对复杂系统进行分析、设计与编程，能有效提高编程的效率，通过封装技术及消息机制可以像搭积木一样快速开发出一个全新的系统。

5.2 类和对象

5.2.1 对象的创建与使用

类和对象是面向对象思想中两个最基本的概念，类是对某一类事物的抽象描述，是现实世界或思维世界中的实体在计算机中的反映。而对象用于表示现实中该类事物的实例个体。简而言之，类是对象的模板，对象是类的实例，一个类可以对应多个对象。

在面向对象的编程思想中，一切皆为对象。在程序中创建对象，必须先定义一个类，然后通过类生成具体的对象。Java 语言使用 new 关键字来创建对象，格式如下：

```
类名 对象名称 = new 类名();
```

例如，创建 Cell 类的对象代码如下：

```
Cell cell = new Cell(2,4);
```

上述代码中，“Cell cell”声明了一个 Cell 类型的变量 cell，“new Cell(2,4)”用于创建 Cell 类的对象。

成功创建对象之后，可以访问对象的成员，格式如下：

```
对象引用.对象成员
```

例如：

```
cell.moveUp();
```

上述代码访问了变量 cell 的成员方法“moveUp()”，作用是使方块向上运动一格。

为了对类和对象有更直观的感受，“模拟电子屏”项目中新增加了 HuskyDog 类和 TeddyDog 类，通过这两个类可以生成对象“哈士奇”和“泰迪”，并且有对应的成员方法，用于设置它们在屏幕上的位置。

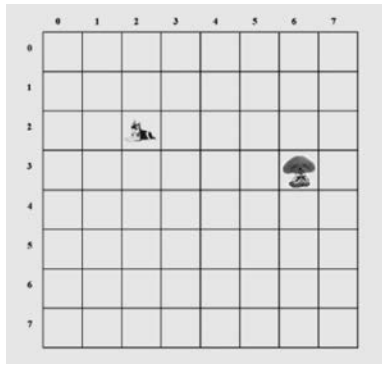

图 5.1　在屏幕上显示两种狗

视频讲解

【例 5.1】 编写程序，将“泰迪”“哈士奇”显示在屏幕上，如图 5.1 所示。

屏幕上有两种不同类型的狗，分别是“哈士奇”和“泰迪”，项目提供了 HuskyDog 类和 TeddyDog 类，通过这两个类可以产生具体的对象，然后将其显示在屏幕上，代码如下：

```
public class Main {
    public static void main(String[ ] args){
        final int SIZE = 8;
```

```
        Screen screen = new Screen(SIZE);

        HuskyDog husky = new HuskyDog();                    //创建哈士奇对象
        husky.setPoint(2,2);                                //设置在屏幕上的位置
        screen.add(husky);

        TeddyDog teddy = new TeddyDog();
        teddy.setPoint(3,6);
        screen.add(teddy);
    }
}
```

通过上述例子可以看到，类就是创建对象的模板，不同的类会产生不同的对象。如果想在屏幕上显示“哈士奇”，就通过 HuskyDog 类产生具体的对象；如果想在屏幕上显示“泰迪”，就通过 TeddyDog 类产生具体的对象。

5.2.2 类的定义

在掌握了如何通过已经设计好的类创建对象之后，接下来就要学习如何设计类。类是对象的抽象概念用来描述一组对象的共同特征和行为。如果想创建对象，首先必须定义类。定义一个新的类，就创建了一种新的数据类型。在学习设计类之前，先看看已经设计好的类有哪些组成部分，以 HuskyDog 类为例，代码如下：

```
public class HuskyDog{
    private int row;
    private int col;
    public void setPoint(int r, int c){
        row = r;
        col = c;
    }
    public void moveUp(){
        row--;
    }
    public void moveDown(){
        row++;
    }
    public void moveLeft(){
        col--;
    }
    public void moveRight(){
        col++;
    }
    public int getRow() {
        return row;
    }
    public int getCol() {
        return col;
    }
}
```

程序定义了一个 HuskyDog 类表示“哈士奇”狗，在该类中定义了两个变量 row、col 分别表示

“哈士奇”在屏幕上的行、列信息。另外，定义了 7 个方法，其中方法 setPoint()用于设置狗在屏幕上的位置，方法 moveUp()、moveDown()、moveLeft()、moveRight()用于控制狗上、下、左、右运动，方法 getRow()和 getCol()用于获得位置信息。通过上述例子可知，类主要包含属性和方法两部分。

(1) 属性也称为成员变量，用来描述类的静态特征，比如位置信息、狗的品种、大小等状态信息。

(2) 方法也称为成员方法，用来描述类的动态特征，包括功能或者行为，比如狗的上、下、左、右运动。

熟悉了类的基本组成部分之后，接下来对类的定义格式、类的成员变量和成员方法进行详细讲解。

1. 类定义格式

Java 语言中，类是通过 class 关键字来定义，一般格式如下：

```
[修饰符] class 类名[extends 父类名][implements 接口名]{
    // 成员变量声明
    // 成员方法声明
}
```

1) 类的修饰符

类的修饰符用于说明类的特殊性质，主要分为两类：访问修饰符和非访问修饰符。访问修饰符能够有效进行访问控制，实现程序“高内聚，低耦合”的目标。访问控制一方面防止用户接触到不该接触的内容，另一方面将变与不变的部分分离，允许库设计者改变类的内部工作机制，而不必担心会对使用者产生影响。

class 前面的访问修饰符可以是 public 关键字或者默认。若修饰符是 public，则该类是公共类，可以被任何包中的类使用。若为默认，也就是没有显性设置访问控制符，则该类只能被同一包其他类使用，因此这一访问特性又称为包访问性。

对于非访问修饰符，如抽象类修饰符 abstract 和最终类修饰符 final 会在后面的章节详细介绍。

2) 类名

Java 类命名符合标识符的命名规则，并且约定类名首字母要大写。

3) extends 和 implements

关键字 extends 和 implements 是可选项，extends 用于说明所定义的类继承于哪个父类。implements 用于说明当前类实现了哪些接口。extends 和 implements 的内容将在下一章详细讲解。

2. 声明成员变量

类声明结束后是一对大括号，大括号里的内容称为类体，主要包括类的成员变量和成员方法。成员变量的声明格式如下：

```
[修饰符] 数据类型 变量名 [ = 值];
```

成员变量的修饰符也分为两类：访问修饰符和非访问修饰符。修饰符会在后文中详细介绍。

数据类型可以为 Java 中的任意类型，既可以为基本数据类型，也可以为引用类型。变量名的命名必须符合标识符的命名规则。在声明变量的时候可以赋初值，也可以不赋初值。

3. 声明成员方法

成员方法是类的行为特征，类似于 C 语言中的函数，方法的声明如下：

```
[修饰符] 返回值类型 方法名 ([参数列表]){
}
```

1）方法的修饰符

方法的修饰符也分为两类：访问修饰符和非访问修饰符。修饰符是可选项，一个方法如果缺省访问修饰符，则可以被同一个类的方法访问或者同一个包中的类访问。方法的修饰符较多，有静态修饰符 static、最终修饰符 final 等，这些修饰符会在后面的内容中逐一介绍。

2）返回值类型

返回值类型为方法返回值的数据类型，可以是任何数据类型，若一个方法没有返回值，则使用关键字 void。

3）方法名

方法名的命名规则符合标识符命名规则。

4）参数列表

参数列表是调用方法时传递的参数信息，方法可以没有参数，也可以有 1 个或多个参数。如果有多个参数，参数声明中间用逗号分隔，而且每个参数都必须包含数据类型和参数名。

5）返回值

当方法需要返回数据的时候，使用 return 语句。当方法不需要返回数据的时候，可以省略 return 语句。

熟悉了类的定义方式之后，接下来通过具体案例来进一步掌握类的定义。


视频讲解

【例 5.2】 为 HuskyDog 类增加一个移动到屏幕某个位置的方法 moveTo(int r,int c)。

新增成员方法 moveTo()，作用是将“哈士奇”移动到屏幕某个位置，也就是将其成员变量 row 和 col 的值设置为参数传递的值，代码如下：

```
public class HuskyDog{
    private int row;
    private int col;
    public void setPoint(int r, int c){
        row = r;
        col = c;
    }
    public void moveUp(){
        row--;
    }
    public void moveDown(){
        row++;
    }
```

```
    public void moveLeft(){
        col--;
    }
    public void moveRight(){
        col++;
    }
    public void moveTo(int r, int c){
        row = r;
        col = c;
    }
    public int getRow() {
        return row;
    }
    public int getCol() {
        return col;
    }
}
```

给 HuskyDog 类增加了 moveTo()方法，可以通过该方法控制“哈士奇”的移动，测试代码如下：

```
public class Main {
    public static void main(String[ ] args){
        final int SIZE = 8;
        Screen screen = new Screen(SIZE);

        HuskyDog husky = new HuskyDog();
        husky.setPoint(3,6);                    //设置的位置为第 3 行第 6 列
        screen.add(husky);
        screen.delay();
        husky.moveTo(4,6);                      //移动到第 4 行第 6 列
    }
}
```

运行程序，“哈士奇”从屏幕中的第 3 行第 6 列移动到第 4 行第 6 列位置。

【例 5.3】 设计一个坐标 Point 类，包含两个成员变量 row、col，用于设置物体的位置信息。


视频讲解

Point 类有两个成员变量 row、col，接下来需要考虑该类需要哪些成员方法，首先需要设计方法设置和获得成员变量 row 和 col 的值，代码如下：

```
public class Point{
    private int row;
    private int col;
    public void setPoint(int r, int c){
        row = r;
        col = c;
    }
    public int getRow(){
```

```
        return row;
    }
    public int getCol(){
        return col;
    }
}
```

有了 Point 类之后，可以将 HuskyDog 类的成员方法 moveTo()修改为：

```
public class HuskyDog{
    private int row;
    private int col;
    public void setPoint(int r,int c){
        row = r;
        col = c;
    }

    public void moveUp(){
        row--;
    }
    public void moveDown(){
        row++;
    }
    public void moveLeft(){
        col--;
    }
    public void moveRight(){
        col++;
    }

    public void moveTo(Point point){
        row = point.getRow();
        col = point.getCol();
    }
    public int getRow() {
        return row;
    }
    public int getCol() {
        return col;
    }
}
```

测试代码如下：

```
public class Main {
    public static void main(String[ ] args){
        final int SIZE = 8;
        Screen screen = new Screen(SIZE);

        HuskyDog husky = new HuskyDog();
        husky.setPoint(3,6);
        screen.add(husky);
```

```
            screen.delay();
            Point point = new Point();
            point.setPoint(5,6);

            husky.moveTo(point);
        }
    }
```

运行程序，“哈士奇”从第 3 行第 6 列移动到了第 5 行第 6 列。

5.2.3　访问控制符

通过前面类的定义可知，无论是类本身，还是成员变量和成员方法都可以通过访问控制符控制访问权限。Java 语言提供了 4 种访问权限，控制级别从大到小依次为 public、protected、default 和 private，访问权限如表 5.1 所示。

表 5.1　访问权限

访 问 权 限	同一个类内部	同一个包内部	不同包中的子类	不同包的非子类
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

1. 公共访问控制符 public

public 访问权限最宽松，如果一个类被 public 修饰，则该类为公共类，可被任何类访问。如果类的成员变量或成员方法被 public 修饰，则任何类都能够访问该成员变量或方法。

2. 保护访问控制符 protected

如果类的成员变量或成员方法被 protected 修饰，则同一个包里的类或者不同包中的子类可以访问该成员变量或方法。

3. 缺省访问控制符 default

如果一个类或者类的成员（包括成员变量和成员方法）没有使用任何访问修饰符修饰，说明它具有缺省访问控制特性。缺省访问控制权限规定该类只能被同一个包中的类访问，这种访问特性被称为包访问性。

4. 私有访问控制符 private

如果类的成员变量或成员方法被 private 修饰，则只有该类自身的成员可以访问该成员变量或方法。而其他任何类，包括该类的子类都不能访问。private 修饰非常重要，类的良好封装就是通过它来实现。

访问控制符能够有效实现访问权限的控制，如同发送微信朋友圈信息时设置访问权限一样，对于所有朋友都可以访问的信息，将其设置为公开；对于只能部分朋友访问的信息，则将其设置为部分可见。

通过设置访问权限，可以很好地对类进行封装，将对象的状态信息隐藏在对象内容中，实现信息隐藏。封装后不允许外部程序直接访问对象的内部信息，而是通过该类所提供的方法来实现对

内部信息的操作访问。封装是面向对象的三大特征之一，是面向对象的核心思想。通过封装可以保护好内部信息，避免外界的干扰和保持类的独立性。

例如，HuskyDog 类的成员变量 row 和 col 的访问权限为私有访问权限，如果对象直接访问程序就会出错，代码如下：

```
HuskyDog husky = new HuskyDog();
husky.row = 25;
husky.col = 24;
```

运行程序，提示出相应的错误信息。

这样就避免直接访问数据导致出现各种风险情况，例如设计各种账户类时，如果能直接访问用户密码，后果不堪设想。在设计类的时候，通常情况下会将类的成员变量访问权限都设置为 private，这样可以很好地实现类的封装。

5.2.4 方法的重载

在设计类的成员方法时，会遇到这样一种情况：在同一个类中，需要设计一系列功能相似，但是应用场景有差别的方法。例如，设计一个计算类的求和方法，需要根据数字的个数和类型，设计多个求和方法适用于不同情况。如果每一个方法都用不同的方法名，使用时容易带来不便。Java 语言提供了方法重载机制，允许在一个类中定义多个同名的方法，这称之为方法重载。实现方法重载，要求同名的方法参数个数或者参数类型不同。例如，Screen 类的 add()方法就使用了方法重载机制，参数既可以是单个的物体对象，也可以是数组。


视频讲解

【例 5.4】 编写程序，为 HuskyDog 类设计 moveTo()方法，使得既可以通过坐标类 Point 类对象作为参数，也可以使用两个整型变量作为参数。

moveTo()方法可以通过不同类型的参数实现相似的功能，这是典型的方法重载应用场景，代码如下：

```
public class HuskyDog{
    private int row;
    private int col;
    public void setPoint(int r, int c){
        row = r;
        col = c;
    }

    public void moveUp(){
         row--;
    }
    public void moveDown(){
        row++;
    }
    public void moveLeft(){
        col--;
    }
    public void moveRight(){
        col++;
    }
```

```
    public void moveTo(int r,int c){
        row = r;
        col = c;
    }

    public void moveTo(Point point){
        row = point.getRow();
        col = point.getCol();
    }

    public int getRow() {
        return row;
    }
    public int getCol() {
        return col;
    }
}
```

测试代码如下：

```
public class Main {
    public static void main(String[ ] args){
        final int SIZE = 8;
        Screen screen = new Screen(SIZE);

        HuskyDog husky = new HuskyDog();
        screen.add(husky);
        screen.delay();
        Point point = new Point();
        point.setPoint(5,6);
        husky.moveTo(point);

        screen.delay();
        husky.moveTo(7,6);
    }
}
```

运行程序，“哈士奇”在屏幕上不断地移动。

方法重载的优点是，功能类似的方法使用同一名字更容易记住，调用起来更简单。通过方法重载，可以实现编译时多态，编译器根据参数的不同调用相应的方法，具体调用哪个方法由编译器在编译阶段静态决定，所以也称之为静态多态。对于 C 语言较为熟悉的读者应该很快能够感受到方法重载的好处。需要注意的是，如果只有返回值不同，不能实现方法的重载。

5.2.5　构造方法

在上述设计的 HuskyDog 类中，每次实例化对象之后，都需要调用 setPoint()方法为位置属性赋值。如果想同 Screen 类一样，实例化对象的同时就为属性赋值，该如何实现？类的构造方法可以实现这种要求。构造方法是类的一种特殊的方法，作用是在创建对象时初始化对象状态。构造方法的定义格式如下：

```
[修饰符] 方法名([参数列表]){
    //方法体
}
```

构造方法的定义与普通方法定义格式相似，但区别是：

（1）构造方法的名称必须与类名相同。

（2）在方法名前没有返回值类型的声明。

熟悉了构造方法的定义格式后，通过一个案例了解构造方法的设计。

视频讲解

【例 5.5】 编写程序，为 HuskyDog 类设计构造方法。

构造方法与类同名，并且没有返回值类型说明，代码如下：

```
public class HuskyDog{
    private int row;
    private int col;
    public HuskyDog(int r, int c){
          row = r;
          col = c;
    }

    public void moveUp(){
         row--;
    }
    public void moveDown(){
        row++;
    }
    public void moveLeft(){
        col--;
    }
    public void moveRight(){
        col++;
    }

    public int getRow() {
        return row;
    }
    public int getCol() {
        return col;
    }
}
```

测试代码如下：

```
public class Main {
    public static void main(String[ ] args){
        final int SIZE = 8;
        Screen screen = new Screen(SIZE);

        HuskyDog husky = new HuskyDog(3,6);
        screen.add(husky);
        screen.delay();
        husky.moveUp();
    }
}
```

运行程序，“哈士奇”向上运动了一格。

每一个类至少有一个构造方法，在定义一个类的时候，若没有显式地定义构造方法，编译器将自动为类提供一个默认构造方法。

1. 默认构造方法

默认的构造方法没有参数，在其方法体中没有任何代码。如例 5.1 中 HuskyDog 类没有显式定义构造方法，编译器就提供了默认构造方法：

```
public HuskyDog(){
}
```

使用 new HuskyDog ()创建 HuskyDog 类的对象时，系统自动调用了该类的默认构造方法。

2. 带参数的构造方法

由于系统自动提供的默认构造方法往往不能满足要求，因此可以在设计类的时候显式地定义构造函数。例如，HuskyDog 类中增加一个构造函数，通过构造函数初始化类的成员变量。需要注意的是，如果类定义了带参数的构造函数，编译器将不再提供默认的构造方法。例如，如果 HuskyDog 类只提供了带参数的构造函数，使用下列语句创建对象：

```
HuskyDog huskyDog = new HuskyDog ();
```

编译器会提示错误信息。

3. 构造方法的重载

如果想使用多种方式创建对象，则可以使用重载构造方法，通过这些重载的构造方法，在创建对象的时候可以灵活选择不同的构造方法，例如：

```
public class HuskyDog{
    private int row;
    private int col;
    public HuskyDog(){
    }

    public HuskyDog(int r, int c){
          row = r;
          col = c;
    }

    public HuskyDog(Point point){
          row = point.getRow();
          col = point.getCol();
    }

    public void moveUp(){
         row--;
    }
    public void moveDown(){
        row++;
    }
    public void moveLeft(){
```

```
            col--;
        }
        public void moveRight(){
            col++;
        }
        public int getRow() {
            return row;
        }
        public int getCol() {
            return col;
        }
    }
```

测试代码如下：

```
public class Main {
    public static void main(String[ ] args){
        final int SIZE = 8;
        Screen screen = new Screen(SIZE);

        HuskyDog huskyA = new HuskyDog(3,6);
        screen.add(huskyA);

        Point point = new Point(5,5);
        HuskyDog huskyB = new HuskyDog(point);
        screen.add(huskyB);
    }
}
```

运行程序，屏幕上出现两只“哈士奇”。

4. this 关键字的使用

在 HuskyDog 类中使用成员变量 row 表示行信息，而在构造方法中使用参数 r 表示行信息，r 表示的信息不够明确，导致程序的可读性较差。如果将参数 r 也修改成 row，又会导致成员变量与局部变量同名冲突的问题，Java 语言中提供了 this 关键字可以很好地解决此问题。this 关键字表示对象本身，准确地说是对象的引用，使用方法如下：

```
this.属性名称
```

语句表示访问类中的成员变量，用来区分成员变量和方法参数重名问题。

例如：

```
public HuskyDog(int row, int col){
    this.row = row;
    this.col = col;
}
```

参数名与成员变量同名时，this. row 表示的是成员变量，而没带 this 的 row 表示的是方法参数。

this 关键字的另外一个用途是在一个构造方法中调用该类的另一个构造方法。例如，在

HuskyDog 类中定义无参构造方法调用了该类的另一个构造方法，代码如下：

```
public HuskyDog(){
    this(0,0);
}
```

使用 this 调用类的构造方法时，需要注意的是：该语句必须是第一条语句。

例如：

```
public HuskyDog(){
    int row = 0;
    int col = 0;
    this(row,col);
}
```

上述代码中，由于 this()不是第一条语句，所以编译时会提示出现错误。

另外，只能在构造方法中使用 this 语句调用其他的构造方法，而不能在其他成员方法中使用。

5.2.6　static 关键字

前面完成的各种程序，如“贪吃蛇”的运动或者“俄罗斯方块”的运动中都忽视了越界的问题。解决越界问题的关键是需要知道屏幕大小。例如，HuskyDog 类的 MoveTo()方法，若要保证不能运动到屏幕外，则需要根据屏幕的大小设置判断条件。在 HuskyDog 类中如何获得屏幕的大小？最直接的方法是新增一个参数，将 Screen 的对象引用传递进来，通过该对象获得屏幕的大小。这个方法虽然能解决问题，但是较为麻烦。Java 语言提供了 static 关键字，可以实现在没有创建任何对象的前提下，仅仅通过类本身来调用成员方法或者变量。

接下来通过一个例子来学习 static 关键字的作用。

【例 5.6】 设计一个 Config 配置类，保存游戏的基本信息。


视频讲解

游戏的配置信息包括游戏名称、版本信息、字体信息等基本信息，都可放置在配置类中，本例中只需要设置屏幕大小信息，代码如下：

```
public class Config {
    public static final int SCREENSIZE = 8;
}
```

测试代码如下：

```
public class Main {
    public static void main(String[ ] args){
        Screen screen = new Screen(Config.SCREENSIZE);
    }
}
```

运行程序，出现一个 8 行 8 列大小的屏幕。

static 关键字又称为静态修饰符，可以修饰类中的成员变量、方法以及修饰代码块优化程序性能。被 static 关键字修饰的变量或方法不需要依赖于对象来进行访问，可以通过类名直接去进行访问。

1. 静态变量

静态变量是被 static 关键字修饰的变量，这类变量属于类的变量，是一个公共的存储单元，被这个类所有的对象共享。任何一个对象操作它的时候，都是对同一个内存单元进行操作。例如，一个游戏的最高分，被所有玩家所共享，每一个玩家打破纪录，都是对同一数据进行操作。


视频讲解

【例 5.7】 设计一个 Player 玩家类，保存游戏玩家的 ID 信息。

每一款游戏都会有用户管理系统，管理游戏玩家的账号信息，如游戏 ID、用户名、账户密码等。本例设计的 Player 玩家信息类只需要设计游戏 ID，为了保证游戏 ID 唯一，ID 号为注册用户的总人数，代码如下：

```
public class Player{
    public static int count = 0;
    private int id;
    public Player(){
        id = count;
        count ++ ;
    }

    public int getID(){
        return id;
    }
}
```

测试代码如下：

```
public class Main {
    public static void main(String[ ] args){
        final int SIZE = 8;
        Screen screen = new Screen(SIZE);

        Player player1 = new Player();
        Player player2 = new Player();

        System.out.println(Player.count);
    }
}
```

运行程序，在控制台上可以观察到结果为 2，意味着已经注册的用户数为 2。System 类是系统提供的系统类，该类定义了三个静态类变量，其中变量 out 是标准输出设备，通过 println()方法输出信息。

2. 静态方法

静态方法是被 static 关键字修饰的方法，与静态变量一样，可以在不创建对象的情况下直接调用。例如，在 Config 类中增加静态方法获得屏幕的大小，代码如下：

```
public class Config {
    public static final int SCREENSIZE = 8;
    public static int getScreenSize(){
        return SCREENSIZE;
    }
}
```

没有被 static 修饰的成员需要先创建对象，然后通过对象才能访问，而静态方法可以在不创建任何对象情况下调用，所以静态方法只能访问用 static 修饰的成员。

5.3 综合案例：重构“贪吃蛇”游戏


视频讲解

通过不断重构改善代码的设计，即使是很糟糕的代码，也可以改进成设计良好的代码。

Java 语言是面向对象的语言，接下来使用面向对象的思想重构“贪吃蛇”游戏。

编写程序，设计“贪吃蛇”类，实现按键“w”“s”“a”“d”控制“贪吃蛇”上、下、左、右运行，如图 5.2 所示。

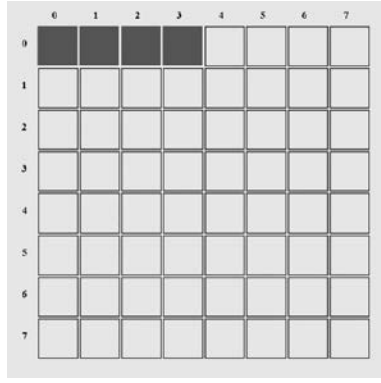

图 5.2　“贪吃蛇”游戏

类的主要组成部分是成员变量和成员方法，设计类的时候需要先思考该类有哪些成员变量和方法。

1. 成员变量

“贪吃蛇”类需要的数据信息包括每个方块的位置、“贪吃蛇”的长度以及运动方向。所以“贪吃蛇”类的成员变量包括一个 Cell 类型的数组和两个 int 类型的变量，数组存储方块的位置信息，int 类型变量分别存储“贪吃蛇”的长度和运动方向。

2. 成员方法

成员方法包括控制“贪吃蛇”运动方向的方法、控制“贪吃蛇”运动的方法和获取方块的位置信息的方法。

根据分析，“贪吃蛇”类的代码如下：

```
public class Snake {
    private Cell [ ] cells;
    private int len;                    //存储蛇的长度
    private int dir;                    //存储运动方向

    public Snake() {                    //无参构造函数
        cells = new Cell[4];
        len = 4;
        dir = 2;                        //初始方向向下
        for(int i = 0; i < cells.length; i++ ) {
            cells[i] = new Cell(0,i);
        }
    }

    public Snake(int len, int dir) {   //有参构造函数
        cells = new Cell[len];
        this.len = len;
        this.dir = dir;
        for(int i = 0; i < cells.length; i++ ) {
            cells[i] = new Cell(0,i);
        }
    }
```

```
    public Cell[ ] getCell(){
        return cells;
    }
    public void setDirection(int d) {                    //设置运动方向
        dir = d;
    }

    public void move() {
        for(int i = len - 1; i > 0; i -- ) {             //蛇身运动
            cells[i].moveTo(cells[i - 1]);
        }
        if(dir == 1) {                                   //向上运动
            cells[0].moveUp();
        }
        if(dir == 2) {                                   //向下运动
            cells[0].moveDown();
        }
        if(dir == 3) {                                   //向左运动
            cells[0].moveLeft();
        }
        if(dir == 4) {                                   //向右运动
            cells[0].moveRight();
        }
    }
}
```

测试代码如下：

```
public class Main {
    public static void main(String[ ] args){
        final int SIZE = 8;
        Screen screen = new Screen(SIZE);
        Snake snake = new Snake();
        screen.add(snake.getCell());

        char key;
        while(true) {
            screen.delay();
            key = screen.getKey();
            if(key == 'w') {
                snake.setDirection(1);
            }
            if(key == 's') {
                snake.setDirection(2);
            }
            if(key == 'a') {
                snake.setDirection(3);
            }
            if(key == 'd') {
                snake.setDirection(4);
            }
```

```
                snake.move();
            }
        }
    }
```

类的设计，往往不是一蹴而就，而是需要反复重构，才能使之结构清晰，具有灵活性以适应不断变化的需求。

例如，可以设计游戏 SnakeGame 类，成员变量包括“贪吃蛇”，成员方法包括初始化游戏、运行游戏，代码如下：

```
public class SnakeGame {
    private Snake snake;
    private Screen screen;
    public void init() {
        final int SIZE = 8;
        screen = new Screen(SIZE);
        snake = new Snake();
        screen.add(snake.getCell());
    }

    public void run() {
        char key = screen.getKey();

        while(true) {
            screen.delay();
            key = screen.getKey();
            if(key == 'w') {
                snake.setDirection(1);
            }
            if(key == 's') {
                snake.setDirection(2);
            }
            if(key == 'a') {
                snake.setDirection(3);
            }
            if(key == 'd') {
                snake.setDirection(4);
            }

            snake.move();
        }
    }
}
```

测试代码如下：

```
public class Main {
    public static void main(String[ ] args){
        SnakeGame game = new SnakeGame();
        game.init();
        game.run();
    }
}
```

设计一款新的游戏，比如“俄罗斯方块”游戏，会发现基本框架都与上面相似，先设计一个游戏类，然后设计游戏中的各种角色类。通过该案例可知，采用面向对象思维方式设计程序，更易扩展。

重构的每个步骤都很简单，比如修改变量的命名，或者删除多余的一句代码，修改一条语句。这些小改变看起来微不足道，但是聚沙成塔，累积起来就能形成质变，从根本上改善程序。

习题

5.1 以下关于构造方法描述正确的是_____。

A. 构造方法的返回类型可以是 void 型

B. 一个类只能拥有一个构造方法

C. 构造方法是类的一种特殊方法，它的方法名必须与类名相同

D. 构造方法的调用与其他方法相同

5.2 在类的定义中可以有两个同名的方法，这种面向对象程序特性称为_____。

A. 封装　　B. 重载　　C. 继承　　D. 重写

5.3 构造方法的作用是_____。

A. 访问类的成员变量　　B. 初始化成员变量

C. 描述类的特征　　D. 保护成员变量

5.4 访问修饰符作用范围由大到小是_____。

A. private-default-protected-public　　B. public-default-protected-private

C. private-protected-default-public　　D. public-protected-default-private

5.5 以下对重载描述错误的是_____。

A. 方法重载只能发生在一个类的内部　　B. 构造方法不能重载

C. 重载要求方法名相同，参数列表不同　　D. 方法的重载与返回值类型无关

5.6 若一个无返回值无参数的方法 method()是静态方法，则该方法正确的定义形式为_____。

A. public void method(){}　　B. static void method(){}

C. abstract void method(){}　　D. void method(){}

5.7 Java 语言中，只限子类或者同一包中的类的方法能访问的访问权限是_____。

A. public　　B. private

C. protected　　D. 无修饰

5.8 如果一个类的成员变量只能在所在类中使用，则该成员变量的修饰符必须是_____。

A. public　　B. protected

C. private　　D. static

5.9 设计“俄罗斯方块”某一种形状的类，如图 5.3 所示，实现按键控制其上、下、左、右运动，并编写测试程序。

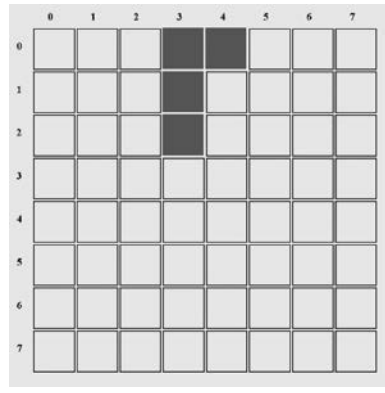

图 5.3 “俄罗斯方块”