第3章

面向对象基础

Java是面向对象的程序设计语言，Java提供了定义类、定义属性和方法等一些基本的功能，也提供了继承、多态、方法重载和访问控制等面向对象语言的其他特征，本章主要介绍用Java定义类和创建对象等基础知识。

本章要点

- 类和对象；
- 定义Java类；
- 创建对象；
- 成员变量；
- 构造方法；
- this关键字；
- static关键字；
- 访问控制；
- 对象清理。

3.1 类和对象

计算机科学中引入面向对象的程序设计方法，其思想是以模仿人类思考问题的方法解决现实问题，从现实世界中客观存在的对象出发来构建软件系统，将软件要解决的问题高度抽象化，并用相应的计算机语言去描述该问题的解决方法。

传统以C语言为代表的面向过程编程语言，数据和对数据的操作是分离的；而Java作为面向对象的程序设计语言，是用对象(Object，也称为实例，Instance)把数据和对数据的操作组合起来封装成一个整体，且抽象为类(Class)表示。类是对现实世界客观存在事物的一种统一的概括性描述。面向对象与面向过程的对比如表3.1所示。

表3.1 面向对象与面向过程的对比

	面向对象	面向过程
核心思想	分门别类，高度归纳	自顶向下，逐步求精
操作方式	数据与其操作封装为一体	数据与对数据的操作分离

续表

	面向对象	面向过程
优势	可维护性好,耦合度低	模块化实现,效率高
缺点	开销大,性能较低	扩展性差,后期维护成本高

从面向对象的思想来讲,软件建模时首先要考虑问题涉及的对象以及它们之间的关系。例如,图书管理系统如果使用面向对象方法来解决,需要将图书管理分为以下四类对象。

(1) 图书:描述图书信息及其行为。

(2) 读者:描述读者信息及其行为。

(3) 管理员:描述管理员信息及其行为。

(4) 借还书规则:负责图书管理中的借还书规则设置等。

然后,对这些对象具有哪些性质和行为,通过使用面向对象的编程语言定义为类的方式进行抽象化描述。类中的数据称为对象的属性(Field),代表对象所具有的性质或者存在的状态,使用变量来表示。对数据的操作(即行为)称为类的方法(Method)。总而言之,类是对现实世界中的某种事物高度抽象化的描述,借助于面向对象程序语言(如 Java)的载体进行表示与实现。

为什么要使用类呢? 原因在于采用简单数据类型表示现实世界一些概念存在局限性。例如,采用 int 类型数据表示一个日期概念,需要使用 3 个变量:

```
int day, month, year;
```

如果要表示 2 个人的生日,就要使用 6 个变量,如果描述更多人,那么将需要声明更多变量;并且在使用中必须时刻注意三者的联系和约束关系,同时在使用日期概念时要同时对三个变量进行访问。而使用类可以把现实问题中的对象映射为程序中的一个整体,那么像日期概念这种类似的问题就迎刃而解了。

注意:类和对象的关系可以理解成抽象和具体的关系。类是对现实世界中客观事物的高度抽象化描述,重点描述该类事物具有的属性和行为。一旦定义类,相当于工程项目多了一种引用数据类型。对象是按照类描述的规范生成的具体实例,对象是客观世界中具有唯一标识的逻辑单元,具有唯一性;一旦实例化一个对象,就会在内存的堆中分配相应的空间存储该对象,并且该对象具有一个唯一的 ID,用于区分其他对象。对象的思维导图如图 3.1 所示。

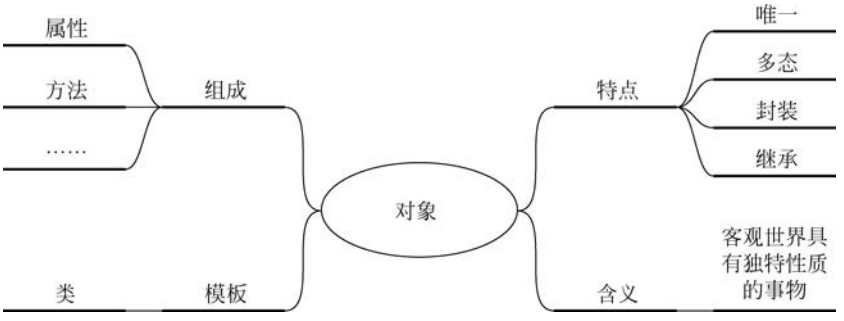

图 3.1 对象的思维导图

3.2　定义 Java 类

Java 中类的定义使用 class 关键字来实现。Java 中的类包括两部分：类首说明和类体。类定义的语法格式：

```
[修饰符] class 类名 [extends 父类名  implements 接口名 1,接口名 2,…]{
            声明属性;
            声明方法;
            声明构造方法;
}
```

定义类时需要注意以下几点。

- 修饰符。定义类时可使用修饰符限定，表示该类具有特定的含义，如当修饰符为 abstract 时，表示该类为抽象类，抽象类不能直接实例化对象，它只能被继承。注意有些修饰符不能同时使用，如 abstract 和 final 是相互矛盾的，这两个关键字不能同时修饰一个类。定义类时也可省略修饰符。
- 类名的命名遵循标识符的命名规则，推荐使用驼峰法命名类。类名后面还可以使用 extends 关键字继承一个父类（注意，只能继承一个父类），也可以使用 implements 关键字实现一个或多个接口，当然 extends 和 implements 可以同时使用，也可以都不使用。
- 类体可以由五种成员组成：属性、方法、构造方法、程序块以及内部类，这些成员可以是 0 个或多个。

1. 属性

类的属性也称为成员变量（Member Variables），声明属性的格式和声明变量的格式基本一致。下面是声明属性的格式：

```
[修饰符] 数据类型 属性名 = [值];
```

声明属性时需要注意以下几点。

- 修饰符可以是 public、private、protected、final、static，也可以省略。同理，public、private、protected 不能同时使用，但可以与 final、static 结合使用。若 final 修饰属性时，表明该属性相当于一个常量，其值不能改变；若 static 修饰属性时，该属性又称为类变量。
- 数据类型可以是基本数据类型，也可以是引用数据类型。
- 定义属性时可以直接赋值，如未赋值，实例化时编译器根据该属性的数据类型赋给它一个默认值。

请看一个声明属性的代码段：

```
public static final double PI = 3.1415926;
```

上述代码声明一个成员变量 PI，其中，public、static 和 final 为修饰词，double 指明该成员变量的数据类型为 double 类型。

注意：在面向对象的程序设计中，使用“属性”一词表达了对象的特征，在定义 Java 类时，属性基于成员变量来实现，成员变量的值表示该属性的状态。一般情况下，属性和成员

变量两个概念是等价的。

2. 方法

方法在类中是描述行为的重要载体,是为完成对数据的操作而组合在一起的语句组。可以重复调用定义的方法,提升了编程效率。第 2 章已经介绍了定义方法的语法格式,此处不再赘述。

【例 3.1】 Student.java

```
public class Student {
    //声明成员变量 name,age,gender
    String name;
    int age;
    String gender;
    //定义方法
    public void info(){
        System.out.println("Name:" + name + ", Age:" +
            age + ", Gender:" + gender);
    }
}
```

例 3.1 定义了学生类 Student,该类包括 3 个成员变量 name、age 和 gender,以及 1 个方法 info()。Student 类描述了学生这一类群体的基本特征。

注意:定义 Java 类时,类体中只能包括以下五种成员。

- 成员变量;
- 方法;
- 构造方法;
- 程序块;
- 内部类。

3.3 创建对象

3.3.1 创建对象概述

定义一个类,就像制定了一个产品规范,如中国的瓶装饮用纯净水卫生标准是 GB/T 17324—2003,那么瓶装饮用水厂商就必须按照此标准或规范生产饮用水。同样,在项目中定义了一个 Java 类,编译器将按照类的定义规范去实例化相应对象,该过程称为对象的创建。

Java 使用 new 运算符实例化对象,具体的格式如下。

```
类名 实例名称 = new 构造方法([参数]);
```

其中,实例名称表示实例化后的对象,实例名称遵循标识符的命名规范;new 运算符调用类的构造方法,如下代码创建一个 Student 类的实例 liming。

```
Student liming;
liming = new Student();
//上述两行代码等价于 Student liming = new Student();
```

上述代码首先声明一个 Student 类型的变量 liming,此时系统为变量 liming 在栈内存

中分配空间。使用 new 运算符调用构造方法 Student()，从而创建 Student 的实例对象 liming，并把该对象放在堆内存，为成员变量 name、age 和 gender 分配空间。变量 liming 并不存储该对象的数据，仅存储堆内存地址，与 C/C++中的指针类似，栈内存中 liming 指向堆内存地址，显而易见，变量 liming 为引用类型。此时的内存状态如图 3.2 所示。

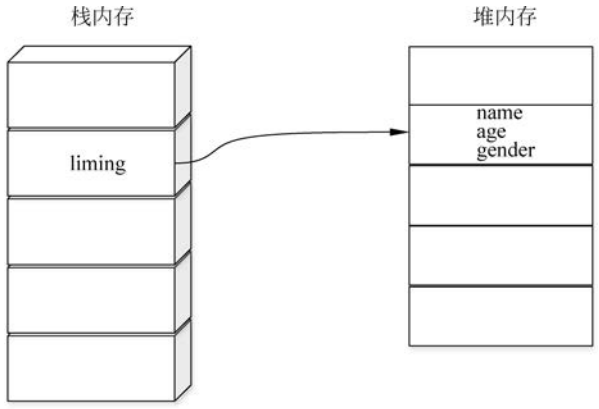

图 3.2　对象的内存状态

注意：Java 中堆和栈的区别如下。

栈与堆都是内存用于存放数据的区域。Java 自动管理栈和堆，程序开发人员不能直接设置栈或堆。堆是一个运行时数据区，类的对象在堆中分配空间，这些对象通过 new 运算符创建，并由 Java 垃圾回收器负责管理。堆的优势在于可以动态地分配内存大小，生存期也不必事先告诉编译器。缺点是要在运行时动态分配内存，存取速度较慢。

栈的优势在于存取速度比堆快，仅次于寄存器，栈数据可以共享。缺点是存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。栈中主要存放一些基本数据类型的变量和对象句柄。

3.3.2　访问成员

对象创建之后，可以使用点(.)运算符访问对象的成员变量和方法。语法格式如下：

```
对象.成员变量|方法
```

但是，如果使用 static 关键字修饰成员变量或方法时，其语法格式如下：

```
类名.类变量|类方法
```

例如，要访问对象 liming 的成员变量 name，可以用下面的代码实现：

```
//为对象 liming 的成员变量 name 赋值
liming.name = "Li Ming";
//调用对象 liming 的 info()方法
liming.info();
```

需要注意的是，用 static 修饰的成员变量称为类变量，相应的方法称为类方法(Class Method)。可以用类名来调用，也可以通过实例来调用；而没有 static 修饰的成员变量和方法称为实例变量和实例方法，只能用对象调用。

注意：为什么可以使用类名访问类变量和类方法？原因在于类变量和类方法属于类所

有,该类的所有对象共享使用类变量和类方法;而实例变量和实例方法属于特定对象所有,该类创建的各对象之间实例变量和实例方法是相互独立的,因此只能通过对象来访问。

综上所述,类是对现实世界中某种事物的高度抽象法描述,通过使用成员变量表示该类事物的特征,使用方法描述该类事物具有的行为。类是使用程序语言的方式制定了一种事物规范,以便重复性地按照这种规范创建对象。单纯地定义一个类而不去实例化,是没有任何意义的,从这个方面讲,对象复用是面向对象的程序设计的优势之一。

3.4 成员变量

3.4.1 变量及其分类

成员变量是类的重要组成部分之一。成员变量直接在类体中声明,其作用范围为整个类。除此之外,程序块内声明的变量、方法形参等称为局部变量。局部变量和成员变量的区别在于作用域的范围不同,成员变量的作用域为整个类,而局部变量作用域只在声明的程序块内有效,一般局部变量的作用域由花括号({})组成的程序块区域决定。

按照变量作用域将 Java 变量进行分类,如图 3.3 所示。

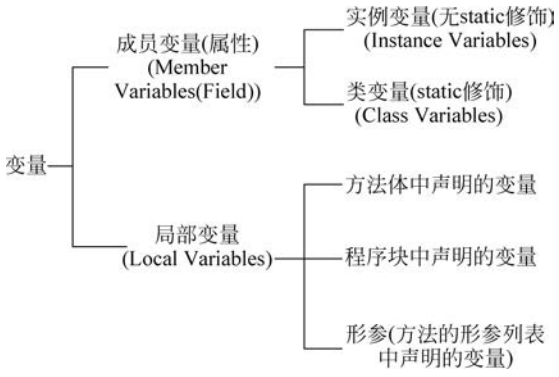

图 3.3 变量分类

注意:如无特殊说明,对下面的几组术语本书认为是等价的,不作区分:实例与对象,成员变量与属性,创建对象与实例化对象。

成员变量包括两种:实例变量和类变量。二者的区别如下:

- 从修饰符讲,实例变量无 static 修饰,而类变量则需要 static 修饰;
- 从生存期讲,实例变量的生存期从执行 new 运算后创建对象开始存在,直到 Java 垃圾回收器销毁这个对象结束,实例变量的生存期与对象的生存期一致;类变量的生存期从定义类开始存在,直到该类所在的程序停止执行为止,很明显,类变量的生存期要大于或等于实例变量的生存期。
- 从访问方式讲,实例变量属于相应的实例,只能通过“实例名.实例变量”的方式来访问,而类变量面向该类所有的对象共享使用,访问时使用“类名.类变量”或“实例名.类变量”两种方式访问。在类方法中,不能直接访问该类的实例变量,而类变量则不受限制。

注意:所谓类方法(Class Method)和实例方法形式上的区别是类方法有 static 修饰,类

方法的访问方式和类变量一样，可以使用“实例名.类方法”和“类名.类方法”两种访问方式。

在方法、构造方法或者程序块中声明的变量称为局部变量，局部变量的声明和初始化都是在方法等相应的花括号({})内。其作用域也在这对花括号内，花括号执行结束后，局部变量就会自动销毁。

局部变量在声明的同时需要显式地为该变量初始化，但成员变量声明时如未赋值，则程序编译时根据变量类型会赋给它一个默认值。下面是一个有关成员变量和局部变量使用的例子。

【例 3.2】 Example3_02.java

```
public class Example3_02 {
    //声明类变量 a
    static int a;
    public static void main(String[] args) {
        //声明局部变量 b,注意 b 并未显式赋值!
        int b;
        //在 for 循环语句中声明局部变量 i,并显式赋值
        for (int i = 0; i < 3; i++) {
            //尽管变量 a 未显式赋值,赋给 a 默认值
            System.out.println("a = " + a);
            //下行代码错误,错误原因在于 b 未显式赋值,需要给变量 b 赋值
            //System.out.println("b = " + b);
        }
        //下行代码错误,超出了声明变量 i 的作用域
        //System.out.println("i = " + i);
    }
}
```

在例 3.2 程序中声明 3 个变量，其中，a 是类变量，作用域最大，在整个类范围内有效；b 和 i 是局部变量，变量 b 的作用域为 main()方法范围内有效，而变量 i 的作用域最小，只在 for 循环语句块范围内有效。同时，若成员变量没有显式赋值，则取该成员变量数据类型的默认值，如 int 类型为 0，double 类型为 0.0，引用数据类型为 null 等。因此，例 3.2 是有错误的，请读者尝试改正本例中的错误。

3.4.2　成员变量和局部变量的区别

成员变量和局部变量有很大的区别，具体有以下 3 点。

- 作用域：成员变量的作用域较局部变量的作用域更大，成员变量作用于整个类、局部变量仅在声明的范围内有效；成员变量的生存期也相对来说较长一些。
- 变量的初始值：成员变量可以显式地赋值，如未赋值，则编译取它的默认值。但是局部变量必须显式地赋值后才能访问。
- 存储位置：成员变量中的实例变量存储在堆内存中，由 Java 垃圾回收机制回收其占用的空间。局部变量和类变量存储在栈内存，随着程序块运行结束而释放空间。

3.4.3　变量选择标准

在实际开发过程中，选择何种情况下声明为实例变量，何时又使用类变量，何种情况下又声明为局部变量呢？如果在程序中选取变量类型不适当，将会导致变量的作用域扩大，造

成变量命名冲突,不利于程序的内聚;另外,还会导致变量的生存期无意识地扩展,浪费内存空间,甚至导致系统崩溃。

请看对例 3.2 改进后的 3 个程序。

【例 3.3】 Example3_03.java

```
public class Example3_03 {
    //声明类变量 a
    static int a;
    public static void main(String[] args) {
        //在 while 循环语句中使用 a 为迭代变量
        for (a = 0; a < 5; a++) {
            System.out.println("a = " + a);
            a++;
        }
        //对 a 再加 2
        a = a + 2;
        System.out.println("最后: a = " + a);
    }
}
```

运行结果:

```
a = 0
a = 2
a = 4
最后: a = 8
```

【例 3.4】 Example3_04.java

```
public class Example3_04 {
    //声明类变量 a,但在整个程序中并没有用到 a,建议注释下行代码
    static int a;
    public static void main(String[] args) {
        int b = 0;
        //在 while 循环语句中使用 b 为迭代变量
        for (b = 0; b < 5; b++) {
            System.out.println("b = " + b);
            b++;
        }
        //对 b 加 2
        b = b + 2;
        System.out.println("最后: b = " + b);
    }
}
```

运行结果:

```
b = 0
b = 2
b = 4
最后: b = 8
```

【例 3.5】 Example3_05.java

```
public class Example3_05 {
    //声明类变量 a,但在整个程序中并没有用到 a,建议注释下行代码
```

```
    static int a;
    public static void main(String[] args) {
        //声明类变量 b,但在整个程序中并没有用到 b,建议注释下行代码
        int b = 0;
        //在 while 循环语句中使用 i 为迭代变量
        for (int i = 0; i < 5; i++) {
            System.out.println("i = " + i);
            i++;
        }
        //下面这两行将无法执行,因为超出了 i 的作用域
        //    i = i + 2;
        //    System.out.println("最后: i = " + i);
    }
}
```

运行结果：

```
i = 0
i = 2
i = 4
```

对比这 3 个程序：例 3.3 使用类变量作为迭代变量，例 3.4 使用 main()方法中声明的局部变量作为迭代变量，例 3.5 使用 for 循环语句中声明局部变量作为迭代变量。它们基本上都可以完成程序的循环语句，但又有所不同，针对这 3 个程序而言，例 3.4 是较好的选择，因为例 3.3 无疑扩大了迭代变量的作用域和生存期，增加内存开销，而例 3.5 则过度缩小变量 i 的生存期，从而 main()方法范围内无法对 i 进行操作。

针对变量作用域，总结以下几点。

- 能用局部变量实现，应尽量避免使用成员变量。尽可能地减小变量的生存期，以节省内存开销。
- 对于局部变量，在不影响功能的情形下，要尽可能缩小变量的作用域。
- 声明成员变量时，选择使用类变量还是实例变量，主要考虑该类的实例是否共享该变量值，如果多个实例共享使用该变量，那么应声明为类变量；如果该变量为实例独立使用，那么应声明为实例变量。

3.5　再论方法

方法是对行为的抽象化描述，它是类的重要组成部分。方法必须定义在类体内部，不能独立地存在。定义方法时应从以下几方面考虑。

- 方法的功能。方法是操纵数据的主要方式，除了从系统责任、问题域等方面重点考虑，还要分析对象的状态及追踪服务的执行路线等方面研究，从而确定如何定义方法。
- 考虑方法的类型。一般应从系统行为和对象自身的行为两方面考虑，研究对象的状态和转换图，确定方法的类型。
- 方法的必要性。定义方法要检查是否真正有用，以及方法的可见性如何设定，方法是否具有较强的内聚性，否则应调整类的方法，使代码保持简洁、安全、高效。

方法可以分为两种类型：实例方法和类方法。使用 static 修饰的方法称为类方法；反

之没有使用 static 修饰的方法称为实例方法。方法调用时应注意以下几点。

- 访问类方法时,可以用“类名.方法名(参数列表)”方式访问,不需要实例化对象就可以访问类方法。
- 类方法不能直接访问类的实例方法和实例变量。由于类方法无须实例化即可访问,而实例变量则必须实例化之后才分配堆内存,因此类方法不能直接访问实例方法和实例变量。
- 访问实例方法时,必须先实例化对象,然后通过“对象名.实例方法(参数列表)”的方式调用该实例方法。

【例 3.6】 Example3_06.java

```
public class Example3_06 {
    public static void main(String[] args) {
        //实例化 MethodDemo 类,创建了实例 md
        Circle circle = new Circle(6.0);
        //实例名.类方法名访问类方法
        System.out.println("实例名.类方法名访问类方法,半径为 6.0 的圆周长为: " +
            circle.getCircumference(6.0));
        //类名.类方法名访问类方法
        System.out.println("类名.类方法名访问类方法,半径为 6.0 的圆周长为: " +
            Circle.getCircumference(6.0));
        //实例名.实例方法
        System.out.println("半径为 6.0 的圆面积为: " + circle.getArea());
        //下行代码错误
        //Circle.getArea();
    }
}

class Circle {
    //声明成员变量 radius
    private double radius;
    //声明类变量 PI,并赋初始值
    public static final double PI = 3.1415926;

    /**
     * 该方法求圆的面积
     * 方法的修饰符为 public,static; 返回值为 double 类型
     *
     * @return 返回值为 double 类型
     */
    public double getArea() {
        //返回表达式 length * width 的值
        return PI * radius * radius;
    }

    /**
     * 求圆周的面积
     *
     * @param r 半径
     * @return double 类型,返回圆的周长
     */
    public static double getCircumference(double r) {
        //下面这行代码错误,类方法不能直接访问实例变量 radius
        //return 2 * PI * radius;
```

```
        //下面这行可以执行
        return 2 * PI * r;
    }

    /**
     * 构造方法
     *
     * @param r 初始化半径 radius
     */
    public Circle(double r) {
        radius = r;
    }
}
```

运行结果：

```
实例名.类方法名访问类方法,半径为 6.0 的圆周长为: 37.699111200000004
类名.类方法名访问类方法,半径为 6.0 的圆周长为: 37.699111200000004
半径为 6.0 的圆面积为: 113.09733360000001
```

例 3.6 验证了类方法不能直接访问本类的实例方法以及实例变量，而实例方法则可以。类方法既支持“类名.方法(参数列表)”，又支持“对象名.方法(参数列表)”，共有两种调用方式，而实例方法只能实例化后采用“对象名.实例方法(参数列表)”的方式访问。

3.6　构造方法

构造方法是一种特殊的方法，它和一般方法具有以下区别。

- 构造方法不能有返回值类型声明，即使是 void 类型也不行。
- 构造方法体不能使用 return 语句返回值。
- 构造方法名称必须与类名完全相同。
- 一个类中可以声明 0 个或多个构造方法，如果没有显式地声明构造方法，则 Java 编译器提供一个形参列表为空的默认构造方法，且构造方法体为空。
- 构造方法通过 new 运算符调用，而普通方法使用点(.)运算符调用。

【例 3.7】 Example3_07.java

```
class Student{
    String name;
    int age;
    //构造方法(1)
    public Student(String n,int a){
        name = n;
        age = a;
    }
    //构造方法(2)
    public Student(String n){
        name = n;
    }
}
public class Example3_07 {
    public static void main(String[] args) {
        //实例化对象 s1
        Student s1 = new Student("Melon");
```

```
        //实例化对象 s2
        Student s2 = new Student("Megan",12);
        System.out.println("s1.name = " + s1.name + ", s1.age = " + s1.age);
        System.out.println("s2.name = " + s2.name + ", s2.age = " + s2.age);
    }
}
```

运行结果：

```
s1.name = Melon, s1.age = 0
s2.name = Megan, s2.age = 12
```

本例中声明了两个构造方法，构造方法(1)初始化成员变量 name 和 age，在构造方法(2)中仅初始化成员变量 name。调用构造方法(2)实例化对象时，成员变量 age 由于没有显式初始化，那么将赋给 age 默认值 0。

注意：构造方法的主要作用：一是为创建对象分配存储空间，二是为成员变量初始化，如果构造方法没有显式初始化，则按成员变量的数据类型赋默认值。

任何一个类至少有一个构造方法，如果定义类时没有定义构造方法，那么 Java 编译器将提供一个默认构造方法，形式如下。

```
public 类名(){}
```

如果已显式定义构造方法，那么编译器将不再提供默认构造方法。

3.7 this 关键字

在介绍 this 关键字之前，请读者先看例 3.8。

【例 3.8】 Example3_08.java

```
public class Example3_08 {
    public static void main(String[] args) {
        Student s = new Student("Melon", 19);
        System.out.println("name:" + s.name + ", age:" + s.age);
    }
}

class Student {
    String name;
    int age;

    public Student(String name, int age) {
        name = name;
        age = age;
    }
}
```

运行结果：

```
name:null, age:0
```

本例与例 3.7 非常相似，但运行结果却不同。在本例中，构造方法的参数列表中有两个形参 name 和 age，局部变量与成员变量重名时，根据变量使用的就近原则，构造方法体中的如下代码，等号两端的变量是同一局部变量，没有为成员变量 name 和 age 赋值。

```
name = name;
age = age;
```

注意：如果类中的属性和方法或程序块中声明的局部变量重名，那么在这些方法或程序块中对同名变量的操作将只针对局部变量有效，根据变量的就近原则，局部变量的作用域屏蔽了外面的成员变量。

Java 引入了 this 关键字，this 相当于“第一人称”代词，指代本类中的成员，如成员变量、方法和构造方法。this 关键字语法比较灵活，其主要作用如下。

- 使用 this 关键字调用本类的成员变量。
- 使用 this 关键字调用本类的方法。
- 使用 this 关键字调用本类的构造方法。

this 调用本类的成员变量或方法的语法格式如下。

```
this.成员;
```

针对例 3.8，可以使用 this 关键字调用成员变量解决该问题，改进后的程序如下。

【例 3.9】 Example3_09.java

```
public class Example3_09 {
    public static void main(String[] args) {
        Student2 s = new Student2("Melon", 19);
        System.out.println("name:" + s.name + ", age:" + s.age);
    }
}

class Student2 {
    String name;
    int age;
    public Student2(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

运行结果：

```
name:Melon, age:19
```

本例构造方法为成员变量 name 和 age 赋值时，前面加上 this 指代成员变量，以示区分形参。使用 this 关键字调用构造方法的语法格式如下。

```
this(参数列表);
```

【例 3.10】 Example3_10.java

```
public class Example3_10 {
    public static void main(String[] args) {
        Student s1 = new Student("Melon");
        Student s2 = new Student(12);
    }
}

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

```
    /**
     * 构造方法(1),初始化属性 name
     * @param name String 类型
     */
    public Student(String name)                    //构造方法(1)
    {
        this.name = name;
    }

    /**
     * 构造方法(2),初始化属性 age,并且使用 this 关键字调用构造方法(1)
     * @param age
     */
    public Student(int age)                        //构造方法(2)
    {
        //this 调用构造方法,该语句必须放在构造方法体的第一行
        this("Melon");
        //初始化 age 属性
        this.age = age;
        //调用 print()方法
        this.print(this.name);
        this.print(this.age);
    }
    public void print(String str) {
        System.out.println("My name is " + str);
    }
    /**
     * 重载 print()方法,打印一个整型数值
     * @param i int 类型
     */
    public void print(int i) {
        System.out.println("My age is " + i);
    }
}
```

运行结果:

```
My name is Melon
My age is 12
```

构造方法(2)的第一行代码 this("melon");表示调用 Student 类的构造方法(1),它可以在构造方法体内调用本类的其他构造方法。使用 this 构造方法时,必须放在构造方法体的第一行。同时,this.print(this.name)表示调用本类的 print()方法打印成员变量 name。

注意:综上所述,this 关键字有以下两个基本用法。

- this 表示类的当前实例,可调用当前实例的成员变量和方法。
- 在构造方法中,使用 this(参数列表)可以调用同一类的其他构造方法,并且"this(参数列表);"语句必须放在构造方法的第一行。

3.8 static 关键字

static 可以修饰变量、方法、程序块,乃至内部类。当 static 修饰类的成员变量时,该成员变量称为类变量(Class Variables),也称为静态变量;相反,没有 static 修饰的成员变量

称为实例变量(Instance Variables)。同理,当 static 修饰方法时,该方法称为类方法(Class Method),没有 static 修饰的方法则称为实例方法(Instance Method)。另外,static 还可以修饰程序块、内部类。

3.8.1　static 修饰成员变量

使用 static 关键字修饰的成员变量称为类变量,类变量和实例变量最大的不同之处在于类变量属于类的所有实例共享,而实例变量属于某个对象。类变量占用的栈内存在类定义时分配,并且该类的所有对象共享使用类变量,任何对类变量的操作都会影响其他对象的使用。类变量从类定义开始生效到该类被卸载结束,因此类变量的生存期几乎与该类的生存期是一致的。

类变量的访问方式有两种:“类名.类变量”方式和类的“对象名.类变量”方式。推荐使用“类名.类变量”方式。

3.8.2　static 修饰方法

使用 static 修饰的方法称为类方法,类方法也是属于类的,而不是像实例方法那样属于某一个对象。像类变量一样,类方法的访问方式也有两种:“类名.类方法(参数列表)”和“对象名.类方法(参数列表)”。在类的内部调用类方法时,也可以把类名省去。

类方法不能直接访问本类的实例方法和实例变量,同样,this 关键字不能在类方法中使用。

【例 3.11】 Example3_11.java

```
class Student {
    String name;
    int age;
    static String school;                          //类变量

    //实例方法,可以直接访问本类中的成员变量和方法
    public void read() {
        System.out.println("Name:" + name + ",age:" + age + ",
            school:" + school);
    }

    //类方法,在类方法中不能直接访问本类的实例变量和实例方法
    public static void write() {
    //  name = "zhangsan";
        school = "HENU";
    }
}

public class Example3_11 {
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student();
        s1.name = "Megan";
        s1.age = 12;
        Student.write();
        s1.read();
        s2.read();
    }
}
```

运行结果：

```
Name:Megan,age:12,school:HENU
Name:null,age:0,school:HENU
```

Student 类的实例方法 read()可以直接访问该类的任何成员，但在类方法 write()中只能访问类变量和类方法。对于类变量 school，当通过 Student.write()修改 school 时，Student 类的实例 s1、s2 都受影响。

3.8.3 static 修饰程序块

程序块是由一对花括号({})包括的语句块，从形式上看是一个相对独立的模块。程序块也分为两类：成员程序块和静态程序块。使用 static 修饰的程序块称为静态程序块。

Java 程序通常使用程序块为成员变量初始化，程序块执行的顺序与它在源程序内的位置无关。例 3.12 是一个使用程序块的例子。

【例 3.12】 Example3_12.java

```
public class Example3_12 {
    //声明实例变量 name 和类变量 age
    String name;
    static int age;

    //成员程序块
    {
        name = "Eric";
        System.out.println("2. name:" + name);
    }

    //静态程序块
    static {
        //静态程序块无法直接访问实例变量 name
        //name = "Melon";
        age = 18;
        //声明一个局部变量 name,仅在该静态程序块中有效
        String name = "Melon";
        System.out.println("1. name:" + name + ", age:" + age);
    }

    public static void main(String[] args) {
        Example12 example = new Example3_12();
        System.out.println("3. name:" + example.name + ", age:" + age);
    }
}
```

运行结果：

```
1. name:Melon, age:18
2. name:Eric
3. name:Eric, age:18
```

本例声明了一个成员程序块和静态程序块。可以看到静态程序块首先被执行，然后是成员程序块。静态程序块在类加载时自动调用，而成员程序块则是在 new 运算符调用构造方法时执行。特别指出，静态程序块不能直接访问本类的实例变量和实例方法，而成员程序块则不受此限制。成员程序块只有在实例化对象时才被执行。

注意：成员程序块和静态程序块有以下区别。

- 执行的顺序不同，静态程序块在类加载时执行且整个生命周期仅执行一次，而成员程序块在使用 new 运算符实例化时执行，每次实例化对象时均会被执行一次；
- 静态程序块只能访问类成员，而成员程序块既可以访问实例成员，也可以访问类成员。

3.9　访问控制

Java 具有很好的安全性，如 Java 的访问控制机制有效地实现了信息隐藏和数据封装，从而避免了非法访问类成员的可能。package 和 import 关键字是 Java 实现访问控制的重要支撑。

3.9.1　访问控制修饰符

Java 提供了 private、protected、public 三种访问控制修饰符来控制对类、成员变量和方法的访问，另外还可省略访问控制修饰符(称为 default)，相当于 Java 类有四种对成员的访问控制方法。访问控制是实现封装的重要手段，这四种访问权限说明如下。

- private：它的访问权限最为严格，当指定类成员为 private 类型时，该成员只能在本类内部可见，其他任何类都无权访问。
- default：如果类或类成员没有使用任何访问控制修饰符时，则为默认类型，即 default 类型。默认类型可以被同一个包中的其他类访问，但不在同一包时则无权访问。
- protected：受保护的访问权限。protected 修饰的类成员，说明它可以被同一个包中的其他类访问，也可以被不在同一包中的子类访问。
- public：public 访问权限最为宽松，当类或类成员为 public 类型时，它可以在所有类中被访问，不管是否在同一个包。

表 3.2 列出了 4 种访问控制级别对比情况。

表 3.2　访问控制修饰符的访问级别

	同一个类	同一包中的类	不同包的子类	任意范围
private	★			
default(默认)	★	★		
protected	★	★	★	
public	★	★	★	★

请看一个有关访问控制修饰符的例子。

【例 3.13】 Example3_13.java

```
package chapter3;
public class Example3_13 {
    public static void main(String[] args) {
        //主类可以访问 Student 类
        Student stu = new Student();
        //不能访问 stu 的私有成员 name 和 age
        //stu.name = "Lisi";
        //stu.age = 20;
        //通过公有的 setter 方法为 name 和 age 赋值
```

```
        stu.setName("ZhangSan");
        stu.setAge(18);
        //使用 getter 方法获取 name 和 age 的值
        System.out.println(stu.getName());
        System.out.println(stu.getAge());
    }
}

//Student 类为 default 类型
class Student {
    private String name;
    private int age;
    //name 属性的 getter 方法
    public String getName() {
        return this.name;
    }
    //name 属性的 setter 方法
    public void setName(String name) {
        this.name = name;
    }
    //age 属性的 getter 方法
    public int getAge() {
        return this.age;
    }
    //age 属性的 setter 方法
    public void setAge(int age) {
        if (age > 0)
            this.age = age;
        else
            System.out.println("年龄必须大于 0!");
    }
}
```

运行结果：

```
ZhangSan
18
```

本例成员变量 name 和 age 都是 private 类型，但为之提供了 public 类型的 setter 和 getter 方法。因此主类 Example3_13 无法直接访问 Student 类的私有成员，但可以通过 setter 和 getter 方法来访问它们。同时，setAge()方法提供赋值校验，避免非法访问带来的问题。成员变量的 setter 和 getter 方法必须是 public 类型，否则 setter 和 getter 方法就没有存在的价值。

注意：只有 public 和默认修饰符能够修饰类，当类指定为 public 类型时，可以通过 import 语句对该类进行复用。如果类省略访问控制修饰符，那么该类是 default 类型，只能被同一包内的其他类访问，不能使用 private 和 protected 修饰类。

3.9.2 隐藏实现

封装的主要目标是实现访问控制，达到隐藏类的具体实现细节，只提供给外部适当的接口，通过这些接口实现类的相关操作，限制某些外部程序非法操作或破坏类的结构。

【例 3.14】 Example3_14.java

```
public class Example3_14 {
```

```
    public static void main(String[] args) {
        Person person = new Person();
        person.name = "zhangsan";
        //直接暴露一些属性,导致所赋的属性值不合法
        person.age = -5;
        person.info();
    }
}
class Person {
    public String name;
    public int age;
    public void info() {
        System.out.println("Name:" + name + ",Age:" + age);
    }
}
```

本例 Person 类的两个成员变量均为 public 类型，意味着其他类可以访问两个成员变量 name 和 age。很显然，这样将带来一些负面问题，如为 age 属性赋一个负数，违反了基本常识。

为避免非法访问问题，对类的成员访问权限加以控制是非常有必要的。通常将类的属性定义为 private 类型，然后为其提供相应的公有 setter 和 getter 方法，getter 和 setter 方法对访问属性加以限制，从而变相地为外部类访问类属性提供了途径，从而保证数据的安全。通过这样的方式把类中的数据封装起来，只对使用者开放特定的接口（如 setter 和 getter 方法），从而阻止了外部类直接操作类中脆弱的部分，该过程称为隐藏实现。

对上述例子的 Person 类做如下改进。

【例 3.15】 Person.java

```
public class Person {
    //声明成员变量并且为 private 类型
    private String name;
    private int age;
    //使用 setter 方法来给成员变量赋值
    public void setName(String name) {
        this.name = name;
    }
    //使用 getter 方法访问成员变量
    public String getName() {
        return this.name;
    }
    public void setAge(int age) {
        if(age > 0)
            this.age = age;
        else
            this.age = 0;
    }
    public int getAge() {
        return this.age;
    }
    public void info() {
        System.out.println("Name:" + name + ",age:" + age);
    }
}
```

本例将成员变量声明为 private 类型,避免其他类对 Person 类的成员变量随意地修改,同时提供 setter 和 getter 方法让其他类访问 Person 类的成员变量,setter 和 getter 方法提供了访问规则。注意声明的 setter 和 getter 方法必须为 public 类型。

隐藏实现是面向对象的程序设计中一个非常重要的概念,通过使用访问控制修饰符,把不需要公开的成员变量及方法封装起来,隐藏了类的具体实现细节,通过给对象发送相应的消息来为外部的类提供相应的服务,把类的功能与类的使用分离。另外,即使改变类的功能时也不会影响类的使用,提高了程序的安全性和可维护性。

3.10 对象清理

Java 无须依赖开发人员手动释放内存,而是由 Java 虚拟机的垃圾回收器自动管理内存。本节主要介绍 Java 的垃圾回收机制。

我们知道使用 new 运算符创建一个对象,当执行 new 运算时,调用类的构造方法并在堆中动态地为实例变量分配空间。new 运算是一个运行时(Running Time)概念,程序在运行时才会为类的实例变量分配空间。下面结合前面讲过的 Person 类,通过例 3.16 来讨论 Java 的垃圾回收机制。

【例 3.16】 Example3_16.java

```
public class Example3_16 {
    public static void main(String[] args) {
        Person jack = new Person();
        jack.setName("Jack");
        jack.setAge(18);
        Person tom = new Person();
        tom.setName("Tom");
        tom.setAge(20);
        //tom 和 jack 同时指向原来 jack 的堆地址
        tom = jack;
        System.out.println(tom.getName() + ", " + tom.getAge());
    }
}
```

运行结果:

```
Jack, 18
```

在本例中,实例化两个 Person 类型的对象 jack 和 tom。当程序运行时会为这两个对象在堆中开辟存储空间,具体如图 3.4 所示。

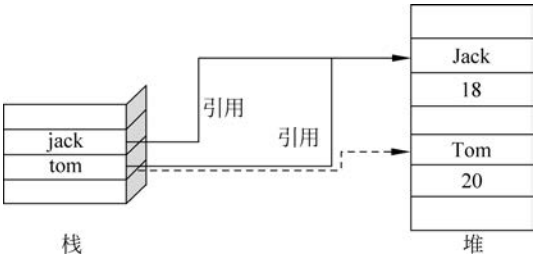

图 3.4 堆的分配与回收

当执行两个 new 运算后，在堆中为 jack 和 tom 两个实例分配了空间以存储其成员变量。栈中的变量 jack 和 tom 引用所指向的堆地址，当执行“tom＝jack;”代码时，表示 tom 也指向了 jack 的引用地址，不再指向原有的引用地址，也就意味着有变量对原有引用地址，那么此时“Tom”及 18 所占用的空间成为垃圾，因此 Java 虚拟机在适当的时候自动调用垃圾回收器清除实例 tom 原来所占用的空间，这种机制称为 Java 的自动垃圾回收机制。通过垃圾自动回收机制释放无用对象的内存空间，减轻了程序开发人员的负担，程序开发人员不必担心什么时候回收空间，垃圾回收器会监控堆中的对象，对那些没有被引用的对象，垃圾回收器会在适当的时候自动释放这部分对象所占用的空间。一般来说，对于 Java 虚拟机，在内存资源不够用时垃圾回收器才会开始工作，回收那些没有被引用的地址空间。

尽管一般情况下程序开发人员无须手动地强制垃圾回收，但是由于我们无法精确地控制究竟何时 Java 虚拟机会调用垃圾回收器，我们仍然可以强制系统进行垃圾回收，尽管这种手段并不推荐使用，但作为学习可以了解一些强制垃圾回收方法。

- 调用 System 类的 gc()方法强制回收无引用的堆空间，如 System. gc()。
- 调用 Runtime 对象的 gc()方法，如 Runtime. getRuntime. gc()。

而 finalize()方法在垃圾回收器工作之前调用，用来验证回收条件是否已经成熟，如对象可能还与其他对象存在某种联系（如继承关系），这时可在 finalize()方法中设定条件，阻止垃圾回收，使垃圾对象重新复活，从而导致垃圾回收器取消回收该垃圾对象。

注意：finalize()方法有以下几个特点。

- 不要主动调用 finalize()方法而是交给垃圾回收器调用。
- finalize()方法并一定被调用，以及何时被调用都无法确定。
- Java 虚拟机调用 finalize()方法时出现异常，垃圾回收器并不会报告异常，而是继续执行。
- Java 虚拟机调用 finalize()方法时，有可能取消垃圾回收，使垃圾对象可能重新复活。

3.11　思政案例：弘扬中华优秀文化——节气

中华优秀文化是中华民族的精神家园和优良传统，是团结中华儿女强大的纽带，是创造中华文明的不竭动力。中华民族有着五千多年连续不断的文明历史，创造了博大精神的中华文明。在 2022 年北京冬季奥运会开幕式上，创造性地以中国传统历法的时光轮转作为倒计时开场，从 24 倒数到 1，冬去春来，四季更替。开幕式当天 2 月 4 日恰逢第一个节气“立春”，诗意的偶然，浪漫的邂逅，巧妙的融合，将中华传统文化与国际体育盛会完美结合，再配上一首首中国古典诗篇，一重又一重的意境汇成全世界人民都看得懂的美好，让全世界观众领略了中华优秀传统文化的魅力。

春雨惊春清谷天，夏满芒夏暑相连。秋处露秋寒霜降，冬雪雪冬小大寒。两千多年前，我们的祖先通过观天时万物总结的二十四节气，蕴含了劳动人民的勤劳智慧与生命哲学。二十四节气，表示自然节律变化，指导农耕生产的时节体系，更包含丰富的民俗事象的民俗系统，例如，清明时节缅怀先烈，冬至日吃饺子。二十四节气蕴含着悠久的文化内涵和历史积淀，是中华民族悠久历史文化的重要组成部分。现行的“二十四节气”是依据太阳在回归黄道上的位置制定，即把太阳周年运动轨迹划分为 24 等份，每 15°为 1 等份，每 1 等份为一

个节气,始于立春,终于大寒。经历史发展,农历吸收了干支历的节气成分作为历法补充,并通过“置闰法”调整使其符合回归年,形成阴阳合历,“二十四节气”也就成为农历的一个重要部分。在国际气象界,二十四节气被誉为“中国的第五大发明”。2016 年 11 月 30 日,二十四节气被正式列入联合国教育、科学及文化组织、人类非物质文化遗产代表作名录。

太阳从黄经零度起,沿黄经每运行 15°所经历的时日称为“一个节气”。每年运行 360°,共经历 24 个节气,每月 2 个。其中,每月第一个节气为“节气”,即:立春、惊蛰、清明、立夏、芒种、小暑、立秋、白露、寒露、立冬、大雪和小寒等 12 个节气;每月的第二个节气为“中气”,即:雨水、春分、谷雨、小满、夏至、大暑、处暑、秋分、霜降、小雪、冬至和大寒等 12 个节气。“节气”和“中气”交替出现,各历时 15 天,现在人们已经把“节气”和“中气”统称为“节气”。二十四节气计算公式:

$$[Y \times D + C] - L$$

其中,Y=年份的后 2 位,D=0.2422,L=闰年数,C 取决于节气和年份。例如,21 世纪立春的 C 值为 3.87,21 世纪清明的 C 值为 4.81。

举例说明:

2022 年立春日期的计算:$[22 \times 0.2422 + 3.87] - [(22-1)/4] = 4$,则 2 月 4 日立春。

2022 年清明日期的计算:$[22 \times 0.2422 + 4.81] - [(22-1)/4] = 5$,则 4 月 5 日清明。

本章案例使用 Java 定义一个节气类 SolarTerms,该类实现计算一个指定年份二十四节气对应的日期。通过用户输入年份,计算该年的二十四节气分布情况。

【例 3.17】 Example3_17.java

```
class SolarTerms{
    //年
    int year;
    //月
    int month;
    //日
    int day;
    //节气(每月的第一个节气)
    String majorSolar;
    //中气(每月的第一个节气)
    String minarSolor;
    int dayOfMajor;
    int dayOfMinor;
    //所有的节气数组
    String[] majorSolarArr = {"", "小寒", "立春", "惊蛰", "清明", "立夏",
"芒种", "小暑", "立秋", "白露", "寒露", "立冬", "大雪"};
    //所有的中气数组
    String[] minorSolarArr = {"", "大寒", "雨水", "春分", "谷雨", "小满",
"夏至", "大暑", "处暑", "秋分", "霜降", "小雪", "冬至"};
    //getter, setter
    public int getYear() {
        return year;
    }
    public void setYear(int year) {
        this.year = year;
    }
    public int getMonth() {
```

```
    return month;
}
public void setMonth(int month) {
    this.month = month;
}
public int getDay() {
    return day;
}
public void setDay(int day) {
    this.day = day;
}
//构造方法
public SolarTerms(){ }
public SolarTerms(int year){
    this.year = year;
    int month = 1;
    setYearOfThousand();
    while (true){
        this.month = month;
        setDay();
        if(month < 10) {
            System.out.print(getYear() + "年 0" + getMonth() + "月" +
                    getDayOfMajor() + "日为" + getMajorSolars() + " ");
            System.out.println(getYear() + "年 0" + getMonth() + "月" +
                    getDayOfMinor() + "日为" + getMinorSolars());
        }
        else {
            System.out.print(getYear() + "年" + getMonth() + "月" +
                    getDayOfMajor() + "日为" + getMajorSolars() + " ");
            System.out.println(getYear() + "年" + getMonth() + "月" +
                    getDayOfMinor() + "日为" + getMinorSolars());
        }
        month ++;
        if(month > 12)
            break;
    }
}
public SolarTerms(int year,int month){
    this(year);
    this.month = month;
}
public SolarTerms(int year,int month,int day){
    this(year,month);
    this.day = day;
}
//年份的千位
int yearOfThousand = 0;
//年份的百位
int yearOfHundred = 0;
//年份的十位
int yearOfTen = 0;
//年份的个位
int yearOfBit = 0;
//临时变量
```

```
    int yearOfTemp;
    public void setYearOfBit(){
        yearOfTemp = getYear();
        yearOfBit = getYear() % 10;
        yearOfTemp /= 10;
    }
    public void setYearOfTen(){
        setYearOfBit();
        yearOfTen = yearOfTemp % 10;
        yearOfTemp /= 10;
    }
    public void setYearOfHundred(){
        setYearOfTen();
        yearOfHundred = yearOfTemp % 10;
        yearOfTemp /= 10;
    }
    public void setYearOfThousand(){
        setYearOfHundred();;
        yearOfThousand = yearOfTemp % 10;
        yearOfTemp /= 10;
    }
    public int getYearOfThousand(){
        return yearOfThousand;
    }
    public int getYearOfHundred(){
        return yearOfHundred;
    }
    public int getYearOfTen(){
        return yearOfTen;
    }
    public int getYearOfBit(){
        return yearOfBit;
    }
    public int getYearOfTemp(){
        return yearOfTemp;
    }

    public void setDay() {
        //[Y×D+C]-L,Y=年代数的后2位、D=0.2422、L=闰年数、C取决于节气和年份
        double c = 0;
        double d = 0.2422;
        int l = 0;
        //得到年份后两位
        int y = getYearOfTen() * 10 + getYearOfBit();
        //此处只计算21世纪的二十四节气,设定月份对应节气的C值
        double majorSolarValues[] = {0.0, 5.4055, 3.87, 5.63, 4.81, 5.52, 5.678, 7.108,
7.5, 7.646, 8.318, 7.438, 7.18};
        c = majorSolarValues[month];
        //1月,2月农历属于上一年
        if(month < 3) {
            l = (int) ((y - 1) / 4);
        }
        else {
            l = (int) (y / 4);
```

```
        }
        dayOfMajor = (int) (y * d + c) - l;
        if (getMonth() == 1 && y == 19)
            dayOfMajor -= 1;
        if (getMonth() == 7 && y == 16)
            dayOfMajor += 1;
        if (getMonth() == 8 && y == 02)
            dayOfMajor += 1;
        if (getMonth() == 11 && y == 89)
            dayOfMajor += 1;
        //此处只计算 21 世纪的二十四节气,设定月份对应中气的 C 值
        double minorSolarValues[] = {0.0, 20.12, 18.73, 20.646, 20.1, 21.04, 21.37, 22.83,
23.13, 23.042, 23.438, 22.36, 21.94};
        c = minorSolarValues[month];
        //1 月,2 月农历属于上一年
        if(month < 3) {
            l = (int) ((y - 1) / 4);
        }
        else {
            l = (int) (y / 4);
        }
        dayOfMinor = (int) (y * d + c) - l;
        if (getMonth() == 1 && y == 82)
            dayOfMinor += 1;
        if (getMonth() == 2 && y == 26)
            dayOfMinor -= 1;
        if (getMonth() == 5 && y == 8)
            dayOfMinor += 1;
        if (getMonth() == 10 && y == 89)
            dayOfMinor += 1;
    }
    public int getDayOfMajor(){
        return dayOfMajor;
    }
    public int getDayOfMinor(){
        return dayOfMinor;
    }

    public String getMajorSolars(){
        majorSolar = majorSolarArr[month];
        return majorSolar;
    }
    public String getMinorSolars(){
        minarSolor = minorSolarArr[month];
        return minarSolor;
    }
}

public class Example3_17 {
    public static void main(String[] args) {
        SolarTerms solarTerms = new SolarTerms(2022);
    }
}
```

运行结果：

```
2022年01月5日为小寒 2022年01月20日为大寒
2022年02月4日为立春 2022年02月19日为雨水
2022年03月5日为惊蛰 2022年03月20日为春分
2022年04月5日为清明 2022年04月20日为谷雨
2022年05月5日为立夏 2022年05月21日为小满
2022年06月6日为芒种 2022年06月21日为夏至
2022年07月7日为小暑 2022年07月23日为大暑
2022年08月7日为立秋 2022年08月23日为处暑
2022年09月7日为白露 2022年09月23日为秋分
2022年10月8日为寒露 2022年10月23日为霜降
2022年11月7日为立冬 2022年11月22日为小雪
2022年12月7日为大雪 2022年12月22日为冬至
```

小结

本章介绍面向对象的程序设计方法，面向对象的方法学是我们分析、设计和实现一个系统尽可能地接近认识一个系统的方法。面向对象的程序设计围绕对象、类、继承、多态、封装等概念来阐述。类是描述对象的“基本原型”，它定义一类对象所能拥有的数据和能完成的操作，类是程序的基本组成单元。成员变量用于表示对象的属性或者具有的状态；方法实现对数据的操作，是对象的功能单元；消息是软件对象通过相互间传递消息来相互作用和通信，消息一般通过方法的参数来传递。一个类可以由5部分组成：变量、方法、构造方法、程序块以及内部类，但这5部分都可以是0个或多个。

对象的具体隐藏实现是指隐藏类的成员变量及实现细节，仅提供一些公用方法，外部的类只能通过公用方法访问类的成员变量，从而保障类中数据的安全和避免外部的非法操作。本章还介绍了Java对象的清理、垃圾回收机制及static、this等关键字的使用方法。