第 5 章　进一步讨论对象和类

5.1　抽象数据类型

5.1.1　概述

绝大多数程序设计语言都预定义了一些基本数据类型，并相应定义了对那些类型的实例执行的操作。比如，对整型、实型等数值类型，有加、减、乘、除等操作；对逻辑类型，有逻辑与、逻辑或、逻辑非等操作。

对于用户自定义的复合数据类型，需要由程序员自己定义一些方法，对该类型的实例进行所需的操作。在早期许多程序设计语言中，复合数据类型及其相关操作的代码之间没有特殊的联系。比如，用户定义日期 Date 类型，并定义一个方法 tomorrow()，它接收一个 Date 类型的参数，据此推断其后继日是哪一天。程序中定义变量的代码和 tomorrow()方法的代码可以是分离的。

有些编程语言改进了这种处理方式，允许数据类型说明和欲对该类型变量进行操作的代码说明之间有较紧密的联系。通常数据类型加上其上的操作称为抽象数据类型。严格地说，抽象数据类型是指基于一个逻辑类型的数据类型以及这个类型上的一组操作。每一个操作由它的输入、输出定义。一个抽象数据类型的定义并不涉及它的实现细节，这些实现细节对于抽象数据类型的用户是隐藏的。

程序 5-1 给出了 Date 类型和 tomorrow 操作间建立的一种联系。

程序 5-1

```
public class Date {
    private int day, month, year;
    Date ( int i, int j, int k) {
         day = i;
         month = j;
         year = k;
    }

    Date() {              //这是个构造方法，显式初始化
         day = 1;
         month = 1;
         year = 1998;
    }

    Date ( Date d) {     //这是带一个参数的构造方法
```

```
            day = d.day;
            month = d.month;
            year = d.year;
        }

        public Date tomorrow() {
            Date d = new Date(this);   //说明一个对象
            d.day++;
            if (d.day > d.daysInMonth()){//daysInMonth()返回每个月中不同的天数
                d.day = 1;
                d.month ++;
                if (d.month > 12) {
                    d.month = 1;
                    d.year ++;
                }
            }
            return d;
        }
    }
```

名为 tomorrow 的代码段在 Java 中叫作方法，也可以称为成员函数。

在有些程序设计语言中，tomorrow()方法的定义或许会要求带一个参数，例如：

```
void tomorrow(Date d);
```

像 Java 这种支持抽象数据类型的语言在数据和操作间建立了较严格的联系，即把方法与数据封装在一个类中。在程序中不是把方法描述为对数据的操作，而是认为数据知道如何修改自己，然后要求数据对它自己执行操作。相应的语句如下：

```
Data d = new Date ( 20, 11, 1998); //已初始化的 date 对象
d.tomorrow();
```

这种写法表明，数据自己执行操作，tomorrow()方法作用于变量 d。要访问 Date 类的域，可使用点操作符“.”：

```
d.day
```

它的意思是“d 所指的 Date 对象中的 day 域”。类似地，d.tomorrow()是指“调用 d 所指的 Date 对象中的 tomorrow()方法”，即对 d 对象进行 tomorrow 操作。

把方法看作是数据的特性，而不把数据与方法分开，这种思想是建立面向对象系统过程中的重要步骤。

5.1.2 定义方法

定义一个抽象数据类型后，还需要为这个类型的对象定义相应的操作，也就是方法。在 Java 中，方法的定义方式类似于其他语言，尤其与 C 和 C++ 很类似。定义的一般格

式如下：

<修饰符> <返回类型> <名字>(<参数列表>)<块>

其中：

- <名字>是方法名，它必须使用合法的标识符。
- <返回类型>说明方法返回值的类型。如果方法不返回任何值，它应该声明为 void。Java 对待返回值的要求很严格，方法返回值必须与所说明的类型相匹配。如果方法说明有返回值，比如说是 int，那么方法从任何一个分支返回时都必须返回一个整数值。
- <修饰符>段可以含几个不同的修饰符，其中限定访问权限的修饰符包括 public、protected 和 private。public 访问修饰符表示该方法可以被任何其他代码调用，而 private 表示方法只能被类中的其他方法调用。关于其他修饰符的说明请参考 2.5.3 节。
- <参数列表>是传送给方法的参数表。表中各元素间以逗号分隔，每个元素由一个类型和一个标识符组成。
- <块>表示方法体，是要实际执行的代码段。

在例 5-1 中，为程序 2-4 中的 Customer 类定义了方法 setName()和 setAddress()。

例 5-1 方法定义示例。

```
void setName (String name) {
    this.name = name;
}
String getAddress() {
    return address;
}
```

下面在 Date 类中增加 daysInMonth()和 printDate()方法，以便完善 Date 类。

程序 5-2

```
public class Date {
    private int day, month, year;
    Date ( int i, int j, int k) {
        day = i;
        month = j;
        year = k;
    }

    Date() {        //构造方法
        day = 1;
        month = 1;
        year = 1998;
    }
```

```
Date ( Date d) {    //带一个参数的构造方法
    day = d.day;
    month = d.month;
    year = d.year;
}

public void printDate() {
    System.out.print(day + "/"  + month + "/" + year);
}

public Date tomorrow() {
    Date d = new Date(this);
    d.day++;
    if (d.day > d.daysInMonth()) {
        d.day = 1;
        d.month ++;
        if (d.month > 12) {
            d.month = 1;
            d.year ++;
        }
    }
    return d;
}

public int daysInMonth() {
    switch (month) {
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        return 31;
    case 4: case 6: case 9: case 11:
        return 30;
    default:
        if ( year % 100 != 0 && year % 4 == 0 ) {
            return 29;
        }
        else return 28;
    }
}

public static void main (String args[]) {
    Date d1 = new Date();
    System.out.print("The current date is (dd / mm / yy): ");
    d1.printDate();
    System.out.println();
    System.out.print("Its tomorrow is (dd / mm / yy): ");
    d1.tomorrow().printDate();
```

```
        System.out.println();

        Date d2 = new Date(28, 2, 1964);
        System.out.print("The current date is (dd / mm / yy): ");
        d2.printDate();
        System.out.println();
        System.out.print("Its tomorrow is (dd / mm / yy): ");
        d2.tomorrow().printDate();
        System.out.println();
    }
}
```

程序 5-2 的执行结果如图 5-1 所示。

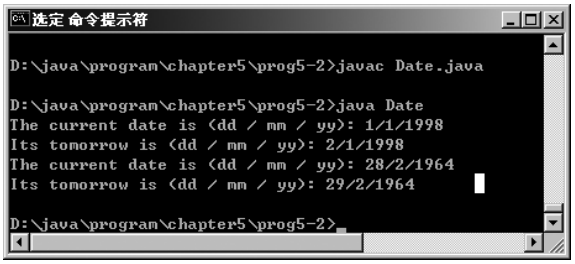

图 5-1　程序 5-2 的执行结果

5.1.3　按值传送

Java 只“按值”传送自变量，即方法调用不会改变自变量的值。当对象实例作为自变量传送给方法时，自变量的值是对对象的引用，也就是说，传送给方法的是引用值。在方法内，这个引用值是不会被改变的，但可以修改该引用指向的对象内容。因此，当从方法中退出时，所修改的对象内容可以保留下来。

程序 5-3 可以说明这个问题。

程序 5-3

```
public class PassTest {
    float ptValue;

    public static void main (String args[]) {
        String str;
        int val;

        //创建类的实例
        PassTest pt = new PassTest ();

        //给整型量 val 赋值
        val = 11;
```

```
        //改变 val 的值
        pt.changeInt (val);

        //val 当前的值是什么呢？打印出来看看
        System.out.println ("Int value is: " + val);

        //给字符串 str 赋值
        str = new String ("hello");

        //改变 str 的值
        pt.changeStr (str);

        //str 当前的值是什么呢？打印出来看看
        System.out.println ("Str value is: " + str);
        //现在给 ptValue 赋值
        pt.ptValue = 101f;

        //现在通过对象引用改值
        pt.changeObjValue (pt);

        //当前的值是什么
        System.out.println ("Current ptValue is: " + pt.ptValue);
    }

    //修改当前值的方法
    public void changeInt (int value) {
        value = 55;
    }

    public void changeStr (String value) {
        value = new String ("different");
    }

    public void changeObjValue (PassTest ref) {
        ref.ptValue = 99f;
    }
}
```

程序执行时，创建 pt 对象，方法内局部变量 val 赋初值 11。调用方法 changeInt()后，val 的值没有改变。字符串变量 str 作为 changeStr 的参数传入方法内。当从方法中退出后，其内容也没有变化。当对象 pt 作为参数传给 changeObjValue()后，该引用所保存的地址不改变，而该地址内保存的内容可以变化，因此退出方法后，pt 对象中的 ptValue 改变为 99f。

输出内容如图 5-2 所示。

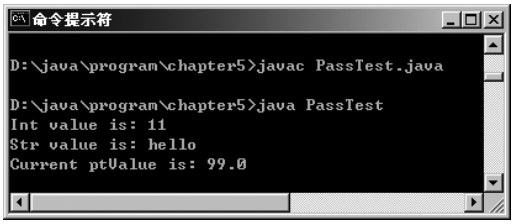

图 5-2　程序 5-3 的执行结果

changeStr()不改变 String 对象，但改变了 PassTest 对象的内容。

5.1.4　重载方法名

如果需要在同一个类中写多个方法，让它们对不同类型的变量进行同样的操作，就需要重载方法名。下面以一个输出文本表示的简单方法为例来说明这个问题。该方法名为 print()。

现在假定需要打印 int、float 和 String 类型的值。每种类型的打印方式不同，这是合情合理的，因为不同的数据类型需要不同的格式，可能要进行不同的处理。按惯例，此时可以建立三个方法，分别叫作 printInt()，printFloat()和 printString()。当然，如果要处理的情况更复杂，则需要建立更多方法，显然这比较麻烦，特别是在调用时容易混淆。可喜的是，面向对象的程序设计方法为我们提供了良好的解决方案，避免了这种麻烦。

在 Java 和其他几种面向对象的程序设计语言中，允许对多个方法使用同一个方法名，这就是方法名的重载。当然，前提条件是系统能够区分实际调用的是哪个方法，才可用这种方式。也就是说，在真正调用方法之前，系统能够根据已知的条件正确判定该调用哪个方法，从几个同名的方法中选出真正要调用的那一个。一个方法区别于另一个方法的要素有这样几个：方法名、参数列表及返回值。因为方法名都是一样的，实际调用之前并不知道返回值是什么，那么只有靠参数列表来区分方法了。实际上也正是如此，Java 根据参数列表来查找适当的方法并调用，这包括参数的个数及各参数的类型。一般地，方法名称和方法的参数列表（包括方法的参数的个数、顺序和类型）称为方法签名。

方法的重载允许 Java 在同一个类中可以定义多个有相同名字的方法，但需要具有不同的参数表，也就是方法的签名不同。不只如此，在不同的类中也可以这样定义。

程序 5-2 中，已经见过方法名的重载了，即 Date 类的构造函数。在一个类的定义中，往往会有多个构造函数，根据初始化时的不同条件调用不同的构造函数，以生成不同的对象。

在前面提到的打印这个例子中，可以根据参数自变量的类型来区分这些方法。要重载方法名，可以如下说明三个方法：

```
public void print(int i)
public void print(float f)
public void print(String s)
```

当调用 print 方法时，可根据自变量的类型选中相应的一个方法。

重载方法有两条规则：

- 调用语句的自变量列表必须足够判明要调用的是哪个方法。自变量的类型可能要进行正常的扩展提升（如浮点变为双精度），但在有些情况下这会引起混淆。
- 方法的返回类型可能不同。即使两个同名方法只有返回类型不同，而自变量列表完全相同则是不够的，因为在方法执行前不知道能得到什么类型的返回值，因此也就不能确定要调用的是哪个方法。重载方法的参数列表必须不同，即参数个数或参数类型或参数的顺序不同。

5.2 对象的构造和初始化

前面已经提到，在说明了一个引用后，要调用 new 为新对象分配空间，也就是要调用构造函数。在 Java 中，使用构造函数（constructor，也称为构造方法）是生成实例对象的唯一途径。在调用 new 时，既可以带有变量，也可以不带变量，这要视具体的构造方法而定。例如，在程序中可以写：new Button("Press me")。这里，Button()就是这个类的构造方法，括号中的字符串是参数值。系统根据所带参数的个数和类型，调用相应的构造方法。调用构造方法时，步骤如下：

（1）分配新对象的空间，并进行默认的初始化。在 Java 中，这两步是不可分的，从而可确保不会有没有初值的对象。

（2）执行显式的成员初始化。

（3）执行构造方法，构造方法是一个特殊的方法。

5.2.1 显式成员初始化

如果在成员说明中写有简单的赋值表达式，就可以在构造对象时进行显式的成员初始化。

例 5-2 成员变量初始化示例。

```
public class Initialized {
    private int x = 5;
    private String name = "Fred";
    private Date created = new Date();
    ...
}
```

如果创建了 Initialized 的实例，那么，在系统为其进行默认的初始化之后，还要给实例中的变量 x 赋值整数 5，给变量 name 赋值字符串"Fred"。

5.2.2 构造方法

显式初始化是为对象域设定初值的一种简单方法。因为设定的初值不具有变化性，所以这种简单的方法有其局限性。实际上，我们可能想处理更一般的情况，此时要执行一个方法来完成初始化。比如，创建按钮对象时，可能想在按钮上显示一个字符串，这

样需用具体字符串进行初始化。显式初始化显然做不到这一点。

为了实现这样的功能，系统定义了默认的构造方法，同时允许程序员编写自己的构造方法完成不同的操作。构造方法是特殊的类方法，有着特殊的功能。它的名字与类名相同，没有返回值，在创建对象实例时由 new 运算符自动调用。同时为了创建实例的方便，一个类可以有多个具有不同参数列表的构造方法，即构造方法可以重载。事实上，不论是系统提供的标准类，还是用户自定义的类，往往都含有多个构造方法。

例 5-3 构造方法示例。

```
public class Xyz {
    // 成员变量
    int x;
    public Xyz() {          //参数表为空的构造方法
        // 创建对象
        x = 0;
    }
    public Xyz(int i) {     //带一个参数的构造方法
        // 使用参数创建对象
        x = i;
    }
}
```

在类 Xyz 中定义了两个构造方法，其中一个的参数表是空的，另一个带有一个 int 型参数。在创建 Xyz 的实例时，可以使用两种形式：

```
Xyz Xyz1 = new Xyz();
Xyz Xyz2 = new Xyz(5);
```

因为构造方法的特殊性，它不允许程序员按通常调用方法的方式来调用，实际上它只用于生成实例时由系统自动调用。构造方法中参数列表的说明方式就决定了该类实例的创建方式。例如在 Xyz 类中，不能像下面这样创建实例：

```
Xyz err1 = new Xyz(1,1);
```

因为，类中没有定义 Xyz(int i, int j)这样的构造方法。

构造方法不能说明为 native、abstract、synchronized 或 final，也不能从父类继承构造方法。

构造方法的特性总结如下：

- 构造方法的名字与类名相同。
- 没有返回值类型。
- 必须为所有的变量赋初值。
- 通常要说明为 public 类型的，即公有的。
- 可以按需包含所需的参数列表。

5.2.3 默认的构造方法

每个类都必须至少有一个构造方法。如果程序员没有为类定义构造方法，则系统会自动为该类生成一个默认的构造方法。默认构造方法的参数列表及方法体均为空，所生成的对象的属性值也为零或空。如果程序员定义了一个或多个构造方法，则将自动屏蔽掉默认的构造方法。构造方法不能继承。

默认构造方法的参数列表是空的，在程序中可以使用 new Xxx()来创建对象实例，这里 Xxx 是类名。如果程序员定义了构造方法，那么，最好包含一个参数表为空的构造方法，否则，调用 new Xxx()时会出现编译错误。默认构造方法的调用请看例 5-4。

例 5-4 调用默认构造方法。

```
class BankAccount{
    String ownerName;
    int accountNumber;
    float balance;
}

public class BankTester{
    public static void main(String args[]){
        BankAccount myAccount = new BankAccount();
        System.out.println("ownerName=" + myAccount.ownerName);
        System.out.println("accountNumber=" + myAccount.accountNumber);
        System.out.println("balance=" + myAccount.balance);
    }
}
```

例 5-4 中定义了一个类 BankAccount，并在程序 BankTester 中创建了一个实例 myAccount。因为在类定义中没有写任何构造方法，所以这里要调用系统给出的默认的构造方法，也就是说 myAccount 的各个域的值为空或零。例 5-4 的输出结果为：

```
ownerName=null
accountNumber=0
balance=0.0
```

我们可以在调用默认的构造函数之后直接对其状态进行初始化，如例 5-5。

例 5-5 对象的初始化。

```
BankAccount myAccount;
myAccount = new BankAccount();
myAccount.ownerName = "Wangli";
myAccount.accountNumber = 1000234;
myAccount.balance = 2000.00f;
```