课程练习

第 3 章　算法基础：穷举、迭代与递归

算法(Algorithm)是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令。算法具有以下五个重要的特征。

(1) 有穷性(Finiteness)：算法必须能在执行有限个步骤之后终止。

(2) 确切性(Definiteness)：算法的每一步骤必须有确切的定义。

(3) 输入项(Input)：一个算法有零个或多个输入，输入是在执行算法时需要从外界取得的一些必要信息。

(4) 输出项(Output)：一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的。

(5) 可行性(Effectiveness)：算法中执行的任何计算步骤都是可以被分解为基本的可执行的操作步骤，即每个计算步骤都可以在有限时间内完成(也称之为有效性)。

算法的描述与所采用的工具有关。这里讨论的是采用电子数字计算机进行问题求解的算法，它通常由操作、控制结构和数据结构 3 要素组成。其中，操作包括算术逻辑操作、关系操作和输入/输出操作；控制结构包括模块结构和流程控制结构(顺序、分支和重复)；数据结构指数据的组织形式。通常，求解同一类问题，算法也会由于思维模式不同而不同。这些不同的算法，往往会表现出不同的执行时间和不同的系统资源(主要是存储空间)占用，分别称为算法的时间效率和空间效率。

本章介绍三种最常用、最基本的算法：穷举、迭代和递归。

3.1 素数序列产生器

3.1　素数序列产生器

3.1.1　问题描述与对象建模

素数又称质数，是指在大于 1 的整数中除了 1 和它本身外不再有其他约数的数。素数序列产生器的功能是输出一个自然数区间中的所有素数。

1. 素数序列产生器建模

1) 现实世界中的素数序列计算对象

本题的意图是建立一个自然数区间，如图 3.1 所示的[11，101]、[350，5500]、[3，1000]等区间内的素数序列。每一个正整数区间的素数序列就是一个对象。

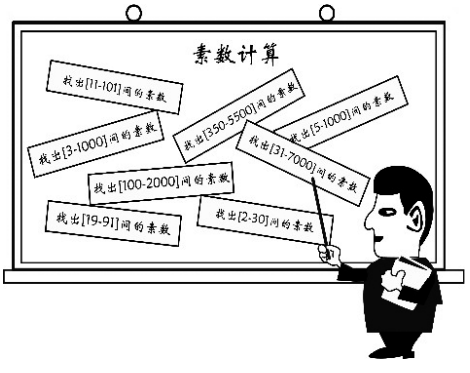

图 3.1　不同的求素数对象

2) 用类图描述的素数序列产生器

对这个问题建模，就是考虑定义一个具有一般性的素数产生器——PrimeGenerator 类。这个类的区间下限为 lowerNaturalNumber，区间

上限为 upperNaturalNumber。这两个值分别用一个变量存储，作为类 PrimeGenerator 的两个成员变量。

类 PrimeGenerator 成员方法除了构造器和主方法外，还需要 getPrimeSequence()——给出素数序列，于是可以得到如图 3.2 所示的 PrimeGenerator 类初步模型。

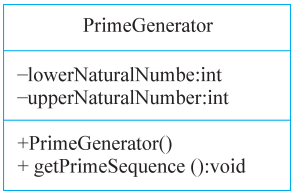

图 3.2　PrimeGenerator 类初步模型

2. getPrimeSequence()方法的基本思路

getPrimeSequence()方法的功能是给出[lowerNaturalNumber，upperNaturalNumber]区间内的素数序列。基本思路是从 lowerNaturalNumber 到 upperNaturalNumber 逐一对每一个数进行测试，看其是否为素数，如果是则输出(用不带回车的输出，以便显示出一个序列)；否则继续对下一个数进行测试。

每次测试使用的代码相同，只是被测试的数据不同，也就是说，这样一个方法中的代码要不断重复执行，直到达到目的为止，这种程序结构称为重复结构，也称循环结构。

在实现 getPrimeSequence()方法时有以下两种考虑。

(1) 用 isPrime()判定一个数是否为素数。

为了将 getPrimeSequence()方法设计得比较简单，把测试一个数是否为素数的工作也用一个方法 isPrime()进行，所以 getPrimeSequence()方法就是重复地对区间内的每个数用 isPrime()方法进行测试。

isPrime()方法用来对某个自然数进行测试，看其是否为素数。其原型应当为：

```
boolean isPrime(int number);
```

测试一个自然数是否为素数的基本方法是把这个数 number 依次用 2～number/2 去除，只要有一个能整除，该数就不是素数。

所以，这两个方法都要采用重复结构。

(2) 在 getPrimeSequence()方法中直接判定一个数是否为素数。

3.1.2　isPrime()判定素数方法的实现

1. 复合赋值运算符

复合赋值是指先执行运算符指定的运算，然后再将运算结果存储到运算符左边操作数指定的变量中。在 Java 中，表达式 m=m+1 可以简化为 m+=1。+=是加和赋值的组合操作符，称为赋值加。例如，i+=5 相当于 i=i+5。除赋值加外，复合赋值操作符还有-=、*=、/= 等，见表 3.1。

表 3.1　复合赋值运算符

操作符	含　义	示　例
+=	加和赋值操作符，它把左操作数和右操作数相加赋值给左操作数	a+=40 等同于 a=a+40
-=	减和赋值操作符，它把左操作数和右操作数相减赋值给左操作数	a-=40 等同于 a=a-40
*=	乘和赋值操作符，它把左操作数和右操作数相乘赋值给左操作数	a*=40 等同于 a=a*40
/=	除和赋值操作符，它把左操作数和右操作数相除赋值给左操作数	a/=40 等同于 a=a/40
%=	取模和赋值操作符，它把左操作数和右操作数取模后赋值给左操作数	a%=40 等同于 a=a%40

复合赋值运算符同简单赋值运算符一样，也是双目运算符，需要两个操作数。不同的是，复合赋值运算符要先执行运算符自身要求的运算后，再将运算后的结果赋值给左边的操作数指定的变量。例如，a *＝b＋20 的等价形式是 a＝a *（b＋20），而不是 a＝a * b＋20。注意，在变换时右边表达式要先加上括号。复合赋值操作符的优先级别与赋值操作符相同。

2. 自增/自减运算符

自增（＋＋）/自减（－－）运算符是一种特殊的算术运算符，在算术运算符中需要两个操作数来进行运算，而自增/自减运算符是一个操作数。＋＋ 与 －－ 的作用是使变量的值增 1 或减 1。操作数必须是一个整型或浮点型变量。自增、自减运算的含义及其使用实例如表 3.2 所示。

表 3.2　自增、自减运算的含义及其使用实例

操作符	含　义	示　例	运 算 结 果
i＋＋	将 i 的值先使用，再加 1 赋值给 i 变量本身	int i ＝ 1； int j ＝ i＋＋；	i ＝ 2 j ＝ 1
＋＋i	将 i 的值先加 1 赋值给变量 i 本身后，再使用	int i ＝ 1； int j ＝ ＋＋i；	i ＝ 2 j ＝ 2
i－－	将 i 的值先使用，再减 1 赋值给变量 i 本身	int i ＝ 1； int j ＝ i－－；	i ＝ 0 j ＝ 1
－－i	将 i 的值先减 1 后赋值给变量 i 本身，再使用	int i ＝ 1； int j ＝ －－i；	i ＝ 0 j ＝ 0

说明：

（1）运算的基本规则为：前缀自增/自减法（＋＋a，－－a），先对变量 a 进行自增或者自减运算，再引用变量 a 的值进行表达式运算；后缀自增/自减法（a＋＋，a－－），先引用变量 a 的值进行表达式运算，再对变量 a 进行自增或者自减运算。

（2）自增/自减只能作用于变量，不允许对常量、表达式或其他类型的变量进行操作。

（3）自增/自减运算可以用于整数类型 byte、short、int、long，浮点类型 float、double，以及字符类型 char。

（4）在 Java 1.5 以上版本中，自增/自减运算可以用于基本类型对应的包装器类 Byte、Short、Integer、Long、Float、Double 和 Character。

（5）自增/自减运算结果的类型与被运算的变量类型相同。

3. 采用 do-while 结构的 isPrime()方法

while 结构的执行特点是“符合条件才进入”；do-while 结构的执行特点是“先执行一次再说”。所以 while 结构的循环体可能一次也不执行，而 do-while 结构最少要执行一次。图 3.3 为 do-while 结构的程序流程图。其基本格式如下：

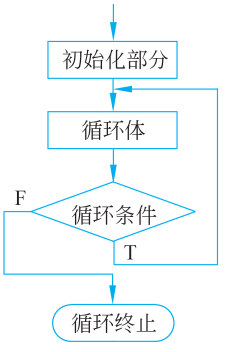

图 3.3　do-while 结构的程序流程图

```
do {
    循环体
} while (循环条件);
```

注意： do-while 结构的最后要以分号结束。

下面先实现素数序列产生器的 isPrime()方法，然后再实现整个 PrimeGenerator 类。

isPrime()方法的基本思路是用 2～number/2 中的数 m 依次去除被检测的数 number，只要有一次能被整除，就证明 number 不是素数，循环除就不再进行，这种解决问题的方法就是穷举法。

穷举法的基本思路是：对于要解决的问题，列举出它所有可能的情况，逐个判断有哪些是符合问题所要求的条件，从而得到问题的解。穷举一般采用重复结构，并由以下 3 个要素组成。

(1) 穷举范围：问题所有可能的解，应尽可能缩小穷举范围。

(2) 判定条件：用于判断可能的解是否是问题真正的解。

(3) 穷举结束条件：用于判断是否结束穷举。

穷举算法是一种最简单、最直接且易于实现的算法，但运算量大，其依赖于计算机的强大计算能力来穷尽每一种可能的情况，从而达到求解的目的。穷举算法效率不高，但适用于解决一些规模不是很大的问题。

【代码 3-1】 采用 do-while 结构实现 isPrime()方法。

```
1  public class PrimeGenerator {
2      /** 主方法 */
3      public static void main(String[] args) {
4          PrimeGenerator ps1 =new PrimeGenerator();
5          System.out.println(ps1.isPrime(2));
6          System.out.println(ps1.isPrime(14));
7      }
8
9      /** 判定素数方法 */
10     private boolean isPrime(int number) {
11         int m =2;
12
13         //1 及小于 1 的数都不是素数,2 是素数
14         if (number <=1) {
15           return false;
16         } else if (number ==2) {
17           return true;
18         }
19
20         do {
21           //若能找到用来整除 number 的数 m,则 number 不是素数
22           if (number %m ==0) {
23               return false;
24           }
25           //取下一个数
26           ++m;
27         } while (m <number / 2);
28
29         return true;
30     }
31 }
```

程序执行结果：

```
true
false
```

说明： 在本代码中，穷举范围是 2～number/2，判定条件是 number % m == 0，穷举结束条件是 m < number / 2。

3.1.3 PrimeGenerator 类的实现

PrimeGenerator 类的实现主要是实现其 getPrimeSequence()方法。

1. 使用 isPrime()判定素数的 PrimeGenerator 类的实现

如前所述,循环结构是通过初始化部分、循环条件和修正部分来控制循环过程的。while 结构和 do…while 结构将这 3 个部分分别放在不同位置,而 for 结构则把这 3 个部分放在一起,形成如下形式:

```
for (初始化部分; 循环条件; 修正部分) {
    循环体
}
```

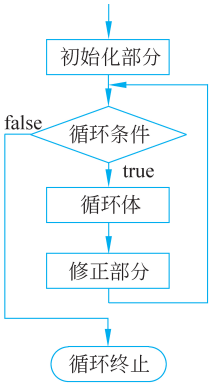

图 3.4 for 结构的程序流程图

这样可以使人对循环过程的控制一目了然,特别适合用在循环次数可以预先确定的情况,所以也把 for 循环称为计数循环。图 3.4 为 for 结构的程序流程图。

使用 isPrime()判定素数的 PrimeGenerator 类的实现,就是在实现 getPrimeSequence()方法时会使用到 isPrime()方法。getPrimeSequence()方法要实现的功能是将[lowerNaturalNumber, upperNaturalNumber]区间内的每一个数依次用 isPrime()方法进行测试,判断是否是素数,具有明显的计数特征,所以应采用 for 结构。当然,isPrime()方法也适合用 for 结构。

下面介绍使用 for 结构实现 getPrimeSequence()方法。

【代码 3-2】 采用 for 结构的 getPrimeSequence()方法,PrimeGenerator 类的完整代码。

```
1  public class PrimeGenerator {
2      /** 主方法 */
3      public static void main(String[] args) {
4          //创建 PrimeGenerator 的对象
5          PrimeGenerator ps1 =new PrimeGenerator(2, 20);
6          //获取素数序列
7          ps1.getPrimeSequence();
8      }
9
10     /** 区间下限 */
11     private int lowerNaturalNumber;
12     /** 区间上限 */
13     private int upperNaturalNumber;
14
15     /** 带参构造器 */
16     public PrimeGenerator(int lowerNaturalNumber, int upperNaturalNumber) {
17         this.lowerNaturalNumber =lowerNaturalNumber;
18         this.upperNaturalNumber =upperNaturalNumber;
19     }
20
21     public void getPrimeSequence() {
22         System.out.print(lowerNaturalNumber +"到" +upperNaturalNumber +
23                                                 "之间的素数序列为: ");
24         //循环控制
25         for (int m =lowerNaturalNumber; m <=upperNaturalNumber; m++) {
26           if (isPrime(m))
```

```
27                 System.out.print(m +",");
28             }
29         }
30
31         /** 判定素数方法 * /
32         private boolean isPrime(int number) {
33             int m =2;
34
35             //1 及小于 1 的数都不是素数,2 是素数
36             if (number <=1) {
37                 return false;
38             } else if (number ==2) {
39                 return true;
40             }
41
42             do {
43               //若能找到用来整除 number 的数 m,则 number 不是素数
44               if (number %m ==0) {
45                   return false;
46               }
47               //取下一个数
48               ++m;
49             } while (m <number / 2);
50
51             return true;
52         }
53     }
```

程序执行结果：

```
2 到 20 之间的素数序列为: 2,3,5,7,11,13,17,19,
```

2. 重复结构中的 continue 语句

前面设计的 getPrimeSequence()代码疏忽了一个问题，即没有考虑用户给出的区间下限小于 2 的情况，也没有考虑给出的区间上、下限反了的情况。下面的代码弥补了这一缺陷。

【代码 3-3】 进一步完善的 getPrimeSequence()代码。

```
1   public void getPrimeSequence() {
2       System.out.print(lowerNaturalNumber +"到" +upperNaturalNumber +
3                                                   "之间的素数序列为:");
4       //区间反了时交换
5       if (lowerNaturalNumber >upperNaturalNumber) {
6           int temp =lowerNaturalNumber;
7           lowerNaturalNumber =upperNaturalNumber;
8           upperNaturalNumber =temp;
9       }
10
11      //循环控制
12      for (int m =lowerNaturalNumber; m <=upperNaturalNumber; m++) {
13          if (m <2) {
14            //短路一次循环中后面的部分,减少 isPrime()方法的调用次数
15            continue;
16          }
17          if (isPrime(m)) {
18              System.out.print(m +",");
19          }
20      }
21  }
```

continue 语句是跳过循环体中剩余的语句而强制执行下一次循环，其作用为结束本次

循环，即跳过循环体中下面尚未执行的语句，接着进行下一次是否执行循环的判定。

3. 不用 isPrime()判定素数的 PrimeGenerator 类的实现

若不使用 isPrime()函数，则 getPrimeSequence()函数成为一个嵌套的循环结构。

【代码 3-4】 采用嵌套循环结构的 getPrimeSequence()函数。

```
1   public void getPrimeSequence() {
2       System.out.print(lowerNaturalNumber +"到" +upperNaturalNumber +
3                                                    "之间的素数序列为: ");
4       for (int m =lowerNaturalNumber; m <=upperNaturalNumber; m++) {
5           boolean flag =true;
6
7           //1 及小于 1 的数都不是素数
8           if (m <=1) {
9               continue;
10          }
11
12          for (int n =2; n <m; n++) {
13            if (m %n ==0) {
14                //发现 number 能被一个数整除，就断定它不是素数
15                flag =false;
16                break;
17            }
18          }
19          if (flag)
20              System.out.print(m +",");
21      }
22  }
```

说明：

(1) 在代码 3-4 中，为了测试一个数是否为素数，采用了一个标记变量 flag。流程一进入外 for 循环中，就将定义一个 flag 并初始化为 true。在内 for 循环中，一旦发现被测试数不是素数，就将 flag 置为 false，并用 break 跳出内循环，否则一直到对被测试数进行完全测试后退出内循环。在内循环外，首先检测 flag 有无改变，如果无，则打印被测试数，然后跳到外循环的增量处取下一个数测试；如果有，则直接跳到外循环的增量处取下一个数测试。

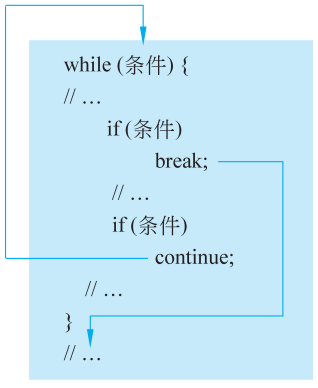

图 3.5 break 与 continue 的作用

(2) 在代码 3-4 中使用了 break 语句，它的作用是强行结束当前的循环。不管是哪种循环，一旦在循环体中遇到 break，系统将完全结束该循环，开始执行循环之后的代码。图 3.5 对 break 和 continue 的作用进行了比较。可以看出，二者有如下区别与联系。

① break 是对循环和 switch-case 结构有效，而 continue 只对循环结构有效。

② 当结构嵌套时，break 语句只对当前层循环或当前层 switch-case 结构有效。continue 也是只对当前层循环有效。

③ break 的作用是跳出，continue 的作用是短路。

④ 这两种操作都是在一定的条件下才能执行，所以在循环体中这两个语句常与 if-else

结构相配合。

(3) 在这个方法中,变量 m 定义在 for 循环体之前(属初始化部分),其作用域为函数作用域。n 和 flag 都定义在内 for 循环体前、外循环内,具有语句作用域。

习题

1. 设 $x = 1$, $y = 2$, $z = 3$,则表达式 $y+=z--/++x$ 的值是(　　)。

   A. 3　　B. 3.5　　C. 4　　D. 5

2. 在 Java 语言中,下列哪些语句关于内存回收的说明是正确的?(　　)

   A. 程序员必须创建一个线程来释放内存

   B. 内存回收程序负责释放无用内存

   C. 内存回收程序允许程序员直接释放内存

   D. 内存回收程序可以在指定的时间释放内存对象

3. 下面有关 for 循环的描述正确的是(　　)。

   A. for 循环体语句中,可以包含多条语句,但要用大括号括起来

   B. for 循环只能用于循环次数已经确定的情况

   C. 在 for 循环中,不能使用 break 语句跳出循环

   D. for 循环是先执行循环体语句,后进行条件判断

4. 下列关于变量作用域的描述,错误的是(　　)。

   A. 在某个作用域定义的变量,仅在该作用域内是可见的,而在该作用域外是不可见的

   B. 在类中定义的变量的作用域在该类中的方法内是可以使用的

   C. 在方法中定义的变量的作用域仅在该方法内

   D. 在方法中作用域可嵌套,在嵌套的作用域中可以定义同名变量

5. 编写程序。一个球从 100m 高度自由落下,每次落地后反跳回原高度的一半;再落下,求它在第 10 次落地时,共经过多少米?第 10 次反弹多高?

3.2　阶乘计算器的迭代实现

3.2 阶乘计算器的迭代实现

3.2.1　问题描述与对象建模

1. 阶乘计算器建模

一个非负整数 n 的阶乘是所有小于或等于 n 的正整数之积,即 n! =1×2×3×…×(n−1)×n。

本题的意图是每一个非负整数都是一个对象,可以计算其阶乘。对于这个问题的建模,就是考虑定义一个具有一般性的阶乘计算器——FactorialCalculator。这个类有一个数据成员 n,用来存储非负整数,其成员方法除了构造器、get 方法、set 方法和主方法外,还需要 fact()——用来计算阶乘,于是可以得到如图 3.6 所示的 FactorialCalculator 类

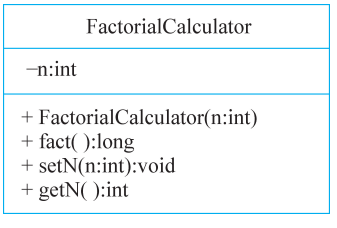

图 3.6　FactorialCalculator 类初步模型

初步模型。

2. fact()方法的基本思路

fact()方法的功能是计算非负整数的阶乘。其基本思路采用累乘法，就是反复地把 1，2，3，…，n－1，n 累乘起来，也体现了循环的思想。累乘是迭代算法策略的基础应用，迭代法也称“辗转法”，是一种不断用变量的旧值递推出新值的解决问题的方法。采用累乘法计算 n 的阶乘步骤如下。

（1）初始化：i＝1，factorial＝1

（2）factorial * i→factorial

（3）i＋1→i

（4）转至步骤(2)重复执行，直到 i 大于 n 结束。

3.2.2 FactorialCalculator 类的实现

FactorialCalculator 类的实现，主要是实现其 fact()方法。

【代码 3-5】 FactorialCalculator 类的完整代码。

```
1  public class FactorialCalculator {
2      /** 主方法 */
3      public static void main(String[] args) {
4          //创建 FactorialCalculator 类对象
5          FactorialCalculator fc=new FactorialCalculator(6);
6          //调用对象的 fact()方法计算阶乘
7          System.out.println(fc.getN()+"!="+fc.fact());
8          fc.setN(10);
9          System.out.println(fc.getN()+"!="+fc.fact());
10     }
11
12     private int n;
13
14     /** 带参构造器 */
15     public FactorialCalculator(int n) {
16         this.n =n;
17     }
18
19     /** 用迭代法计算阶乘 */
20     public long fact() {
21         long factorial =1;
22
23         if (n <0) {
24             //抛出不合理参数异常
25             throw new IllegalArgumentException("必须为非负整数!");
26         }
27
28         //循环控制
29         for (int i =1; i <=n; i++) {
30           //每循环一次进行乘法运算
31           factorial *=i;
32         }
33
34         return factorial;
```

```
    }

    public int getN() {
        return n;
    }

    public void setN(int n) {
        this.n =n;
    }
}
```

程序运行结果：

```
6!=720
10!=3628800
```

习题

1. 一个数如果恰好等于它的因子之和，这个数就称为“完数”。例如，6＝1＋2＋3。编写程序找出 1000 以内的所有完数。

2. 编写程序。求 s ＝ a ＋ aa ＋ aaa ＋ aaaa ＋ aa…a 的值，其中，a 是一个数字。例如 2＋22＋222＋2222＋22222(此时共有 5 个数相加)，几个数相加由键盘输入值控制。


3.3 阶乘计算器的递归实现

3.3 阶乘计算器的递归实现

3.3.1 什么是递归

若一个算法直接或者间接地调用自己本身，则称这个算法是递归算法。递归可以把一个大型复杂的问题层层转换为一个或多个与原问题相似的规模较小的问题来求解，通过少量语句，实现重复计算。在函数实现时，因为解决大问题的方法和解决小问题的方法往往是同一个方法，所以就产生了函数调用它自身的情况。

适宜于用递归算法求解的问题的充分必要条件是：

(1) 可以通过递归调用来缩小问题规模，简化后的新问题与原问题有着相同的解决形式。

(2) 某一有限步的子问题有直接的解存在，即递归必须有简洁的退出条件。

当一个问题存在上述两个基本要素时，该问题的递归算法的设计方法是：

(1) 把对原问题的求解设计成包含对子问题的求解形式。

(2) 设计递归出口。

递归算法解题通常显得很简洁，结构清晰，可读性强；但递归算法的运行效率较低，无论是耗费的计算时间，还是占用的存储空间，都比非递归算法要多。

3.3.2 阶乘的递归计算

通常求 n! 可以描述为：

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

也可以写为：

$$n! = n\times(n-1)\times\cdots\times3\times2\times1 = n\times(n-1)!$$

这样，一个整数的阶乘就被描述成为一个规模较小的阶乘与一个数的积。用函数形式描述，可以得到以下递归模型。

$$fact(n)=\begin{cases}\text{非法} & (n<0) \quad (1)\\ 1 & (n=0) \quad (2)\\ n\times fact(n-1) & (n>0) \quad (3)\end{cases}$$

其中，式(1)和式(2)给出了递归的终止条件，称为递归出口，式(1)或式(2)皆可作为递归出口；式(3)给出了 fact(n)和 fact(n－1)之间的关系，称为递归体。

一般地，一个递归模型由递归出口和递归体两部分组成，前者确定递归何时结束，后者确定递归求解时的递归关系。递归模型必须有明显的结束条件，即至少有一个递归出口，这样就不会产生无限递归的情况了。

3.3.3 用递归实现阶乘计算器

用递归实现阶乘计算器，主要是将 fact()方法改为用递归实现。

【代码 3-6】 用递归实现的 fact()方法。

```
1  public class FactorialCalculator {
2      /** 主方法 */
3      public static void main(String[] args) {
4          //创建 FactorialCalculator 类对象
5          FactorialCalculator fc =new FactorialCalculator(5);
6          //调用对象的 fact()方法计算阶乘
7          System.out.println(fc.getN() +"!=" +fc.fact(fc.getN()));
8      }
9
10     private int n;
11
12     /** 带参构造器 */
13     public FactorialCalculator(int n) {
14         this.n =n;
15     }
16
17     /** 用递归法计算阶乘 */
18     public long fact(int n) {
19         long factorial =1;
20
21         if (n <0) {
22           //抛出不合理参数异常
23           throw new IllegalArgumentException("必须为非负整数!");
24         } else if (n ==0) {
25           return 1;
26         } else {
27           factorial =n * fact(n -1);
28           return factorial;
29         }
30     }
31
32     public int getN() {
33         return n;
34     }
35
```

```
    public void setN(int n) {
        this.n =n;
    }
}
```

程序运行结果：

```
5!=120
```

说明：

(1) 递归由两个过程组成：递推和回归。递推就是把复杂问题的求解推到比原问题简单一些的问题的求解；当获得最简单的情况后，逐步返回，依次得到复杂的解，就是回归。将求 n! 转换成求(n－1)!，而(n－1)! 又可以转换成(n－2)!，(n－1)! ＝(n－1)×(n－2)!，…，重复这个过程，直至 0! ＝1。0! ＝1 称为结束条件。这个过程称为“递推”。当“递推”到结束条件时，就可以开始“回归”，通过 0! 求出 1!，通过 1! 求出 2!，…，通过(n－2)! 求出(n－1)!，最后通过(n－1)! 求出 n!。这个过程称为“回归”。“回归”就是一个值“回代”的过程。图 3.7 体现了求 fact(4)的递归计算过程。

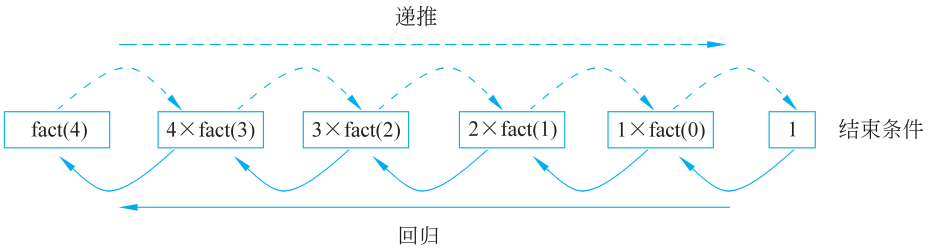

图 3.7　求 fact(4)的递归计算过程

(2) 递归过程不应无限制地进行下去，当调用有限次后，就应当到达递归调用的终点得到一个确定值(例如图 3.7 中的 fact(0)＝1)，然后进行回代。在这样的递归过程中，程序员要根据数学模型写清楚调用结束的条件。

习题

1. 编写程序。输入两个正整数 m 和 n，使用递归求其最大公约数。

2. 编写程序。一个人赶着鸭子去每个村庄卖，每经过一个村子卖去所赶鸭子的一半又一只。这样他经过了七个村子后还剩两只鸭子，问他出发时共赶了多少只鸭子？经过每个村子卖出多少只鸭子？要求使用递归实现。

3.4　内 容 扩 展


3.4 内容扩展

3.4.1　用静态成员变量记录素数的个数

若想记录素数序列产生器对象所生成的素数总个数，可用静态成员变量来存储不同素数序列产生器对象生成的素数个数总和。

1. 静态成员变量的性质

不用 static 修饰的成员变量称为实例变量，而用 static 修饰的成员变量称为静态成员变

量(简称静态变量、静态域、静态属性、静态字段等)。静态成员变量具有如下一些重要特性。

(1) 具有类共享性。静态成员变量不用作区分一个类的不同对象,而是为该类的所有对象共享,所以也称为类属变量(简称类变量)。当要使用的变量与对象无关又不是一个方法中的局部变量时,就需要定义一个静态成员变量。static 成员的这一特性使得可以使用一个静态变量 count 作为素数序列产生器对象之间的共享变量,存储素数生成的个数。

同样,类的 static 方法(静态方法)也称为类方法,即当一个方法与生成的对象无关时可以将其定义为静态方法,最典型的静态方法是 main()。

(2) 静态成员变量可以被任何(静态或非静态)方法直接使用,可以由类名直接调用,也可以用对象名调用。例如,System.in、System.out 和 System.err 表明 in、out 和 err 是 System 的静态成员。这是与实例变量的不同之处,实例变量只能由类的实例调用。但是,静态方法只能对静态变量进行操作。

(3) 静态成员变量不用区分不同对象,所以不通过构造器初始化,而是在类声明中直接显式初始化。若不直接显式对其进行初始化,编译器将对其进行默认初始化。如果是对象引用,则默认初始化为 null;如果是基本类型,则初始化为基本类型相应的默认值。

(4) 静态成员变量每个类只存储一份,为该类所有对象共享;而实例变量每个对象都会存储一份,为该类每个对象私有。

(5) 类中的静态成员变量,在该类被加载到内存时,就分配了相应的内存空间,并且所有对象的这个静态变量都分配给相同的一处内存,以达到共享的目的;而实例变量只有在创建对象时才会分配内存空间,并且分配到不同的内存空间。

2. 带有静态成员变量的 PrimeGenerator 类定义

【代码 3-7】 带有静态成员变量的 PrimeGenerator 类定义。

```
1  public class PrimeGenerator {
2      /** 主方法 */
3      public static void main(String[] args) {
4          PrimeGenerator ps1 =new PrimeGenerator(2, 20);
5          ps1.getPrimeSequence();
6          //用类名 PrimeGenerator 调用其静态变量 count
7          System.out.println("\n已生成素数个数: "+PrimeGenerator.count);
8          PrimeGenerator ps2 =new PrimeGenerator(40, 70);
9          ps2.getPrimeSequence();
10         System.out.println("\n已生成素数个数: "+PrimeGenerator.count);
11     }
12
13     /** 存储素数产生的个数 */
14     static int count=0;
15
16     /** 区间下限 */
17     private int lowerNaturalNumber;
18     /** 区间上限 */
19     private int upperNaturalNumber;
20
21     /** 带参构造器 */
22     public PrimeGenerator(int lowerNaturalNumber, int upperNaturalNumber) {
23         this.lowerNaturalNumber =lowerNaturalNumber;
24         this.upperNaturalNumber =upperNaturalNumber;
25     }
26
27     private void getPrimeSequence() {
```

```
28          System.out.print(lowerNaturalNumber +"到" +upperNaturalNumber +
29                                                  "之间的素数序列为: ");
30          //循环控制
31          for (int m =lowerNaturalNumber; m <=upperNaturalNumber; m++) {
32            if (isPrime(m)) {
33                //记录素数个数
34                count++;
35                System.out.print(m +",");
36            }
37          }
38      }
39
40      /** 判定素数方法 */
41      private boolean isPrime(int number) {
42          //其他代码
43      }
44  }
```

程序运行结果：

```
2到20之间的素数序列为: 2,3,5,7,11,13,17,19,
已生成素数个数: 8
40到70之间的素数序列为: 41,43,47,53,59,61,67,
已生成素数个数: 15
```

说明：

(1) lowerNaturalNumber、upperNaturalNumber 属于实例变量。创建类的对象时，不同对象的实例变量，有不同的副本，它们存储在相互独立的内存空间中。一个对象的数据成员的值的改变不会影响到另一个对象中同名的数据成员。

(2) count 属于类变量。一个类的类变量(静态变量)，所有由这个类生成的对象都共用这个类变量，类装载时就分配存储空间。静态变量将变量值存储在一个公共的内存地址。因为它是公共的地址，所以如果一个对象修改了静态变量的值，则所有对象中这个变量的值都会发生改变。若想让一个类的所有实例共享数据，就要使用静态变量。

图 3.8 反映了实例变量及静态变量的存储情况。从图中可以看出，属于实例的实例变量(lowerNaturalNumber 和 upperNaturalNumber)存储在互不相关的内存中，静态变量(count)是被同一个类的所有实例所共享的。

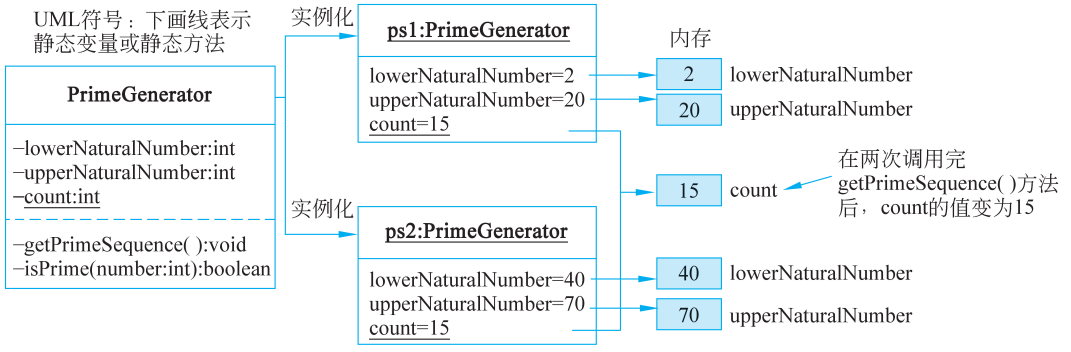

图 3.8　实例变量及静态变量存储情况

3.4.2 静态成员方法——类方法

1. 静态成员方法的性质

在 Java 类中不仅可以有静态成员变量——类属性，还可以有静态方法——类方法，就是用 static 修饰的方法，它们与类属性一样对于所有的类对象是公共的。没有被 static 修饰的方法就是非静态方法，也称为实例方法。

(1) 静态成员方法可以由类名直接调用，也可以用对象名调用。而实例方法必须用对象名访问。

(2) 静态方法只能直接访问静态成员(静态变量和静态方法)，而不能直接访问非静态成员(实例变量和实例方法)；而非静态方法既可以直接访问静态成员，也可以直接访问非静态成员。静态成员和实例成员的关系总结如表 3.3 所示。

表 3.3 静态成员和实例成员的关系

√：能访问 ×：不能访问	实例成员		静态成员	
	实例方法	实例属性	静态方法	静态属性
实例方法	√	√	√	√
静态方法	×	×	√	√

【代码 3-8】 静态成员和实例成员访问示例。

StaticInstMethodDemo.java

```
1  public class StaticInstMethodDemo {
2      /** 主方法 */
3      public static void main(String[] args) {
4          StaticInstMethodDemo simd =new StaticInstMethodDemo();
5          //用对象名调用实例方法
6          simd.print11();
7          //用对象名调用静态方法
8          simd.print21();
9          //用类名调用静态方法
10         StaticInstMethodDemo.print21();
11     }
12
13     /** 实例变量 */
14     int a =3;
15     /** 静态变量 */
16     static int b =5;
17
18     /** 实例方法 */
19     public void print11() {
20         //能访问实例变量 a
21         System.out.println("in print11 method: a=" +a);
22         //能访问静态变量 b
23         System.out.println("in print11 method: b=" +b);
24
25         //能访问实例方法 print12
26         print12();
27         //能访问静态方法 print22
28         print22();
29     }
30
31     /** 实例方法 */
32     public void print12() {
33         System.out.println("in print12 method: 实例方法");
```

```
34      }
35
36      /** 静态方法 */
37      public static void print21() {
38          //不能访问实例变量 a
39          //System.out.println("a="+a);
40          //能访问静态变量 b
41          System.out.println("in print21 method: b=" +b);
42
43          //不能访问实例方法 print12
44          //print12();
45          //能访问静态方法 print22
46          print22();
47      }
48
49      /** 静态方法 */
50      public static void print22() {
51          System.out.println("in print22 method: 静态方法");
52      }
53  }
```

运行结果：

```
in print11 method: a=3
in print11 method: b=5
in print12 method: 实例方法
in print22 method: 静态方法
in print21 method: b=5
in print22 method: 静态方法
in print21 method: b=5
in print22 method: 静态方法
```

说明：如果类的成员没有使用任何访问权限修饰符，那么该成员只能被同包的其他类成员访问，如 StaticInstMethodDemo 类的数据成员 a 和 b。

（3）静态方法中不能使用 this、super 关键字。

（4）类的静态方法和实例方法在内存中都只存储一份，同一类的多个对象在内存中共用这一份方法代码。不过，两者分配入口地址的时机不一样。对于静态方法，在该类被加载到内存时就分配了相应的入口地址。从而静态方法不仅可以被类创建的任何对象调用执行，也可以直接通过类名调用。静态方法的入口地址直到程序退出才被取消。对于实例方法，只有该类创建对象后才会分配入口地址，从而实例方法可以被类创建的任何对象调用。需要注意的是，当创建第一个对象时类中的实例方法会分配入口地址，当再次创建对象时就不再分配入口地址。也就是说，方法的入口地址被所有的对象共享，当所有的对象都不存在时，方法的入口地址才被取消。

main()方法就是一个静态方法，当所在的类加载时即分配了入口地址，因而使 JVM (Java Virtual Machine)无须创建对象即可直接调用它。

如何判断一个变量或方法应该是实例的还是静态的？如果一个变量或方法依赖于类的某个具体实例，那就应该将它定义为实例变量或实例方法。如果一个变量或方法不依赖于类的某个具体实例，就应该将它定义为静态变量或静态方法。

2. 将 isPrime()定义为静态方法

分析素数序列产生器的 isPrime()方法可以发现，在这个方法中不对任何实例变量进行操作，即它与类的实例无关，仅与类有关。或者说，isPrime()方法为类的所有实例共享。这

样的方法可以定义为静态方法。

【代码 3-9】 将 isPrime()定义为静态方法。

```
1  public class PrimeGenerator {
2      /** 主方法 */
3      public static void main(String[] args) {
4          PrimeGenerator ps1 =new PrimeGenerator();
5          //用对象名调用静态方法
6          System.out.println(ps1.isPrime(2));
7          //用类名调用静态方法
8          System.out.println(PrimeGenerator.isPrime(14));
9      }
10
11     /** 判定素数方法 */
12     private static boolean isPrime(int number) {
13         int m =2;
14
15         //1 及小于 1 的数都不是素数,2 是素数
16         if (number <=1) {
17           return false;
18         } else if (number ==2) {
19           return true;
20         }
21
22         do {
23           //若能找到用来整除 number 的数 m,则 number 不是素数
24           if (number %m ==0) {
25               return false;
26           }
27           //取下一个数
28           ++m;
29         } while (m <number);
30
31         return true;
32     }
33 }
```

3.4.3 变量的作用域和生命期

1. 变量的访问属性

Java 语言要求所有程序元素都放在有关类中。在本章中,getPrimeGenerator()和 isPrime()都是类 PrimeGenerator 的成员方法。细心的读者可能已经发现,在这两个方法中各有一个变量 m。那么这两个变量会产生冲突吗?答案是不会,因为它们各自有自己的作用域(scope)和生命期。

变量的访问属性主要涉及 4 个方面,即生命期(也称存储期)、访问权限、作用域和可见性。这好比要访问一个人,首先要确定叫这个名字的人是否在世,如果他还没有出生或者已经死亡,即他不在生存期内,那是绝对无法访问的;其次,要看这个人是否在要访问的范围内,例如,活动的权限范围就在某个城市,那么要访问的这个人虽然活着,但不属于这个城市,也不可访问;第三,要看有没有权限见这个人;第四,要看这个人名有没有被覆盖,例如,有一位名字为王明的县领导,还有一个普通家庭中也有一个叫王明的人,显然,在家里说:“王明吃饭”,不会是叫县领导王明吃饭。这就是家里的“王明”,覆盖了县里的领导“王明”。

2. 变量的作用域

变量的作用域是指变量名在程序正文中有效的区域。“有效”指的是在这个区域内该变量名对于编译器是有意义的。因此，变量的作用域由变量的声明语句所在的位置决定，即在哪个范围域中声明的变量，其作用域就是那个区域。根据定义变量位置的不同，可以将变量分成两大类：成员变量(实例变量和类属变量)和局部变量。下面分为实例变量、类属变量和局部变量三种情形进行讨论。

1) 实例变量的作用域

实例变量声明在类定义中，所以实例变量的作用域在类的每个实例——对象中，即一个类实例的所有成员方法都可以引用它。

2) 类属变量的作用域

类属变量的作用域是一个类代码区域以及该类的所有实例中。

3) 局部变量的作用域

局部变量是声明在某个代码块中的变量，可以分为如下 3 种情形讨论。

(1) 声明在一个代码块(即用花括号括起来的一组代码，包括方法体中声明的变量)中的变量，其作用域就在这个代码区间内，在这个区间外部的任何引用都会导致编译错误或不正确的结果。例如在本章中，getPrimeGenerator()方法体中定义的 m 只能在 getPrimeGenerator()方法体中被引用，在 isPrime()方法体中定义的 m 只能在 isPrime()方法体中被访问。两个 m 各自独立，在各自的作用域内被引用，不会产生混淆。如果在 getPrimeGenerator()方法体中企图引用在 isPrime()方法体中定义的 m，将导致错误。

(2) 方法参数也是一个局部变量，其作用域是整个方法体。

(3) 异常处理参数也是一个局部变量，它们一般声明在一个 catch 后面的圆括号中作为这个 catch 的参数，作用域在其后面的代码块中。

3. Java 数据实体的生命期

这里将在 Java 程序运行中占有一块独立的存储空间的数据称为数据实体。所谓数据实体的生命期，是指该数据实体从获得分配的存储空间到该空间被回收之间的时间区间。

1) 变量的生命期与对象的生命期

如前所述，Java 数据类型可以分为基本类型和引用类型两大类。相应的数据对象可以分别称为变量和对象。变量的生命期是由编译器自动分配与回收的，例如：

- 类属变量的生命期是与类相同，即从类被装载到类被撤销。
- 实例变量的生命期是与对象相同，即从对象被创建到对象被撤销。
- 局部变量的生命期是与所在的程序块有关，即从声明开始到所在的块结束。

而 Java 对象是用 new 操作创建的，它不会因定义的代码区间结束而自动撤销。但是在任何一个程序中，任何一个对象都有自己的使命，它的使命一旦完成，存在就没有必要，却占据着系统的内存资源，使这些内存资源无法被回收利用，这种现象称为“内存泄漏”。这样，老的对象占据资源，又为了执行新的使命需要生成新的对象。这个过程不断进行，内存泄漏加剧，可利用内存资源不断减少，有可能导致 JVM(Java Virtual Machine)崩溃。

2) 类属变量、实例变量与局部变量的比较

表 3.4 为类属变量、实例变量与局部变量的比较。

表 3.4　类属变量、实例变量与局部变量的比较

比较内容	类属变量	实例变量	局部变量
其他名称	类变量、静态成员变量、静态域(属性、字段、变量)	对象变量、实例域(属性、字段、状态)	方法变量
定义位置	定义在类中,方法之外	定义在类中,方法之外	定义在方法头、方法体、代码块等位置
存在特征	用 static 修饰的类属性	不用 static 修饰的类属性	不能用 static 修饰
与方法的关系	独立于方法	独立于方法	一般从属于某个方法(定义在方法外初始化块中的局部变量除外)
存储分配时间	虚拟机加载类时	创建一个类的实例时	定义时
存储区	堆区	堆区	栈区
存储数量	每个类只有一份存储	每个实例都有一份存储	在定义域内只有一份存储
默认生命期	从类加载到类销毁	从对象创建到对象被销毁	从声明到所在代码段执行结束
默认初始值	有	有	无
可用范围	为所有类的对象共享	只能为某个对象使用	所定义的代码段
调用与引用	可用类名、对象名调用; 可在类的任何方法中引用	可由对象、this 调用; 不可用类名调用; 不可在静态方法中引用	仅可在所定义的方法内被引用; 不可用类名、对象名、this 调用

注意: 由于局部变量没有默认值,因此,局部变量定义后,必须初始化(赋值)才能使用。

【代码 3-10】 变量作用域与生命期示例。

```
1   public class ScopeLifeDemo {
2       /** 主方法 */
3       public static void main(String[] args) {
4           ScopeLifeDemo sd1 =new ScopeLifeDemo();
5           sd1.test();
6           System.out.println("in main method: b=" +b);
7           //局部变量 b
8           int b =22;
9           //局部变量会隐藏同名的成员变量
10          System.out.println("in main method: b=" +b);
11          ScopeLifeDemo sd2;
12          {
13              //在代码块中创建的局部对象引用 sd3
14              ScopeLifeDemo sd3 =new ScopeLifeDemo();
15              sd3.setA(88);
16              sd3.test();
17              //sd2 和 sd3 指向同一个对象了
18              sd2 =sd3;
19          }
20          //错误: sd3 在代码块外部就不能用了,不过其对象在堆中还是存在的
21          //sd3.test();
22          //对象在堆中一直存在,因此 sd2 仍然能访问它指向的对象
23          sd2.setA(99);
24          sd2.test();
25      }
26
27      /** 实例变量 */
```

```
    int a;
    /** 类变量 */
    static int b = 11;

    /** 局部变量: 形式参数 a */
    public void setA(int a) {
        this.a = a;
    }

    public void test() {
        //方法中定义的局部变量 j
        int j = 3;
        if (j == 3) {
          //代码块中定义的局部变量 k
          int k = 5;
          System.out.println("in test method: k=" + k);
          //错误: 同一个作用域范围的包裹下局部变量不可以重名
          //int j=8;
        }
        //错误: 代码块中定义的局部变量不能被外部调用
        //System.out.println("k="+k);
        System.out.println("in test method: a=" + a + ",b=" + b + ",j=" + j);
        int i;
        //错误: i 没有赋值,不能使用
        //System.out.println("i=" + i);
    }
}
```

运行结果如下：

```
in test method: k=5
in test method: a=0,b=11,j=3
in main method: b=11
in main method: b=22
in test method: k=5
in test method: a=88,b=11,j=3
in test method: k=5
in test method: a=99,b=11,j=3
```

3.4.4 基本类型打包

1. 基本类型的包装类

基本类型不是类类型，为了将基本类型当作类类型处理，并连接相关方法，Java 提供了与基本类型对应的包装类，见表 3.5。其中，Byte、Double、Float、Integer、Long 和 Short 是 Number 类的子类。

表 3.5 基本类型的包装类

基本类型	char	byte	short	int	long	float	double	boolean
包装容器类	Character	Byte	Short	Integer	Long	Float	Double	Boolean

2. 基本类型与对应的包装类之间的转换以及自动装箱和拆箱

一般来说，可以使用如下转换方法。

(1) 基本类型转换为类对象通过相应包装类的构造器完成，例如：

```
Integer intObj = new Integer(8);
```