第3章　流程控制

程序的基本结构包括顺序、分支和循环。本章介绍如何利用 Java 语言将人的思维用计算机程序来体现，从而解决现实问题。

学习目标

- 认识语句，理解不同类型语句的作用；
- 理解程序控制的概念；
- 掌握条件语句，能够使用 if、switch 来控制程序的不同执行路径；
- 掌握 for、while 和 do 句型结构控制程序的循环执行；
- 理解分支和循环的影响语句范围；
- 理解并能应用 break 和 continue 调整程序中的流程控制结构；
- 理解变量的作用域；
- 理解并使用断言对程序进行调试；
- 能够编写具有一定功能的程序。

3.1　句、块和空白

在第 1 章讨论程序结构的时候，提到构成 Java 程序的最小单位是类，而类则是由属性和方法构成。方法代表了某个具体的功能，一个方法就是为完成某个功能的若干条语句的集合，如 System. out. println( ) 中的 println( ) 就是 JDK 提供的一个供输出信息的方法。

3.1.1　语句

语句就和自然语言一样，告诉计算机要做什么。在 Java 编程语言中，语句是一条由分号“;”终止的代码，它是一个完整的可执行单元。例如，下面是一条赋值语句，将等号右边算术表达式的结果赋给左边的变量 total。

```
total=a+b+c+d+e+f;
```

有时候，由于编辑区宽度的限制以及为了提高程序的可阅读性，一条语句可能会分行书写，这并不影响语句的完整性，例如下面的语句和上述语句的功能完全相同。

```
total=a+b+c+
      d+e+f;
```

具体来说，在 Java 中，主要有以下的语句类型。

1. 声明语句

在第 2 章中，介绍了有关变量的声明，附加上一个表示结束的分号之后，就成了声明语

句，如：

```
int a=0;
```

变量声明加了一个分号就构成了一条声明语句，程序执行到此时，将会在内存中分配 4 个字节用于存储赋给变量 a 的值。

```
Student stu;
```

这是声明了一个引用类型的变量，其引用的对象类型是 Student。

2. 表达式语句

赋值表达式、自增表达式、方法调用、对象创建都可以和分号一起构成表达式语句，如：

```
System.out.println("Welcome");
```

这是调用了 out 对象的方法 println()，这是一种方法调用的形式，通常用于没有返回值的方法调用。

```
a=Math.abs(-3.1);
```

这是调用了 Math 类的求绝对值方法 abs()，这也是一种方法调用的形式，通常用于有返回值的方法调用，以便能够得到方法执行后的结果。

```
value=100;
```

赋值表达式加上分号，就构成了赋值语句。

```
a++;
```

自增表达式加上分号，就构成了自增语句。

3. 空语句

空语句就是仅包含一个分号的语句，没有任何实际作用，但它是一条可执行语句。

4. 控制语句

控制语句主要负责语句的执行顺序和方向，例如循环、分支、跳转等，在随后的流程控制中将详细介绍。

3.1.2 语句块

一个语句块(block)是以“{”和“}”为边界的 0 到多条语句集合，有时也称作复合语句。复合语句的执行可被视为是一条语句。

语句块可被嵌套。HelloWorldApp 类包含了一个 main 方法，这个方法就是一个语句块，它是一个独立单元。其他一些语句块的例子如下：

```
//a block 语句
{
    x=y+1;
    y=x+1;
}
//类声明所包含的语句块
```

```
public class MyDate {
    int day;
    int month;
    int year;
}
//一个嵌套语句块的例子
while ( i<large ) {                    //循环语句块开始,用"{"表示
    a=a+i;
    if ( a==max ) {                    //判断语句块开始,用"{"表示
        b=b+a;
        a=0;
    }                                  //判断语句块结束,用"}"表示
}                                      //循环语句块结束,用"}"表示
```

3.1.3 空白

在源代码元素之间允许空白,空白的数量不限。空白(包括空格、tabs 和新行)可以改善阅读源代码时的视觉感受,特别是利用空白明确显示出不同的嵌套结构(即常说的"缩进"风格),便于阅读和理解。如下面的程序片段所示:

```
public class ComputeArea {
    public static void main(String[] args) {
        int r=10;
        …
    }
}
```

程序中方法声明、方法中的语句等都和包含它们的上层结构向后退了几个空格,以保证这种逻辑上的嵌套关系呈现更好的视觉效果,便于理解。

最后关于语句需要强调的是,语句必须有一个存在范围。例如,循环语句必须由循环控制语句完全控制,一个方法中的语句必须放置在方法体的一对括号内,而不能出现在括号外。

3.2 顺序结构

一个程序(方法)基本的执行过程就是从前到后进行的。构成 Java 程序的是类,类的最小功能单元是方法,而方法是有一条或多条语句构成的,这些语句间最简单的结构关系是顺序结构,即语句是按照它们在方法中出现的先后顺序逐一执行的。例如,程序 3-1 描述了用程序实现交换两个变量值的过程。

根据变量的定义,在任何一个时刻,每个变量只能保存一个值,当把一个新的值赋给变量时,变量原来表示的值就没有了,因此,两个变量交换无法做到直接相互交换,不过可以借助一个临时的中间变量来实现,具体步骤如图 3-1 所示。

```
    /**
     * 程序 3-1: 交换两个变量的值
     */
(01)  public class Swap {
(02)    public static void main(String[] args) {
(03)        int a=10,b=20;
(04)        int t=0;
(05)        t=a;
(06)        a=b;
(07)        b=t;
(08)        System.out.println("a="+a);
(09)        System.out.println("b="+b);
(10)    }
(11)  }
```

第 3 行：程序用声明语句定义了两个变量 a 和 b；

第 4 行：声明了一个变量 t，用作临时保存；

第 5 行：因为无法直接把 b 的值赋给 a，因为这样 a 的值就被覆盖了，所以在将 b 的值赋给 a 之前，先将 a 的值临时保存到变量 t；

第 6 行：因为执行第 5 行语句时，将 a 的值已经保存到 t 中，所以，可以将 b 的值赋给 a，执行后，a 和 b 代表相同的值；

第 7 行：因为执行第 6 行语句时，b 的值已经赋给 a，所以将 t 保存的原来 a 的值再赋给 b，执行后，b 就获得了原来 a 的值，这样就实现了两个变量相互交换各自的值。

通过图 3-1 可以看出，这个交换程序按照语句出现的先后顺序逐一执行，自然地完成了两个变量的交互。

注意：图 3-1 中的图被称为“程序流程图”，其中的方框代表一条或一组语句；带箭头的直线表示程序的执行流向，流线的标准流向是从左到右和从上到下，沿标准流向的流线可不用箭头指示流向，但沿非标准流向的流线应用箭头指示流向；最后的椭圆形表示程序执行的结束。

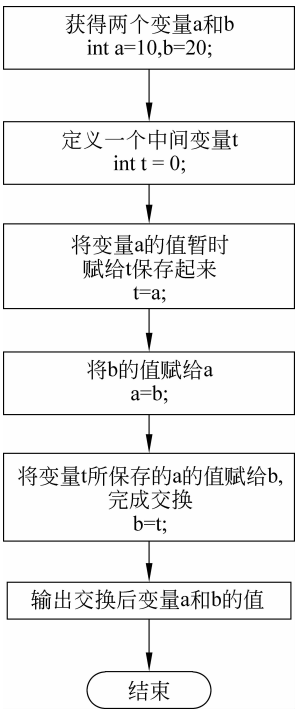

图 3-1　交换两个变量

3.3　选择结构

程序控制可以定义为对程序语句的执行顺序进行的规定。程序执行结构并不总是由顺序结构构成的，就如同早上出门时总是根据是否下雨来决定带不带雨伞一样，Java 也提供了使用分支语句在两种或更多的情况中做出选择，根据结果执行不同的程序语句。

3.3.1　if…else 语句

if…else 语句的基本句法如下：

```
if(布尔表达式){
```

```
    语句或块;
}
else{
    语句或块;
}
```

(1) 在 Java 编程语言中,if()用的是一个布尔表达式,而不是数字值,这一点与 C/C++ 不同,当布尔表达式为真时,执行其后的语句或语句块,为假时,根据是否有匹配的 else 决定如何执行。

(2) 另外,分支中的 else 部分是选择性的,当测试条件为假时如不需做任何事,else 部分可被省略,另外,else 部分不能独立存在,必须和 if 匹配。

(3) if 语句和 else 控制的语句可以是一条语句或多条语句,如果是多条语句,则这些语句必须放在随后紧跟的大括号定义的复合语句内。

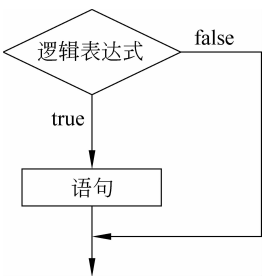

图 3-2 单分支选择语句

(4) if 语句有 3 种形式:单分支、双分支和多分支选择语句。

1. 单分支语句

单分支语句是最简单的一种情况,如图 3-2 所示,其一般形式为

```
if (布尔表达式)
    语句;
```

这种形式描述了当表达式成立时,执行随后的一条语句,如果需要执行的是多条语句,则这些语句应当放在大括号中,构成复合语句,类似下面这样。

```
if (布尔表达式) {
    语句 1;
      …
    语句 n;
}
```

无论哪种情况,当表达式不成立时,则不再执行分支控制的语句,而执行分支语句后续的语句,继续程序的执行。

程序 3-2 描述了一个求绝对值的例子。

```
    //程序 3-2: 单分支结构
    public class Abs {
      public static void main(String[] args) {
(01)        int a=-10;
(02)        if(a<0)
(03)            a=-a;
(04)        System.out.println("a="+a);
        }
    }
```

由于变量 a 的值是负值,程序 3-2 的执行过程是 1→2→3→4,其中 a=-a 是 if 语句的控制范围,假如 a 的值是大于 0 的,则 if 的逻辑表达式为 false,条件不满足则不会执行这条

语句，则程序 3-2 的执行过程是 1→2→4。

注意：为了更好地表现分支语句的影响范围，即使只影响一条语句，最好的编码风格是也要用大括号把它们括起来，不仅改善了程序的可阅读性，而且会避免程序出现修改上的错误。

2. 双分支语句

双分支语句体现了非此即彼的情况，如图 3-3 所示，如果逻辑表达式为真，则执行语句块 1，否则执行语句块 2（注：语句块可以只有一条语句）。

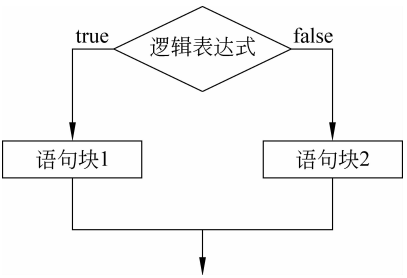

图 3-3　双分支 if 语句

双分支语句的一般形式如下：

```
if (逻辑表达式)
    语句 1;
else
    语句 2;
```

这种形式的分支，如果表达式结果为真，则执行语句 1，否则执行语句 2，随后执行分支后面的其他语句。

如果分支需要影响的语句有多条，需要将多条语句用大括号括起来，构成复合语句，如下所示：

```
if (逻辑表达式) {
    语句块 1;
} else {
    语句块 2;
}
```

程序 3-3 描述了求两个整数之间最大值的算法。

```
    //程序 3-3: 双分支结构
    public class Max {
        public static void main(String[] args) {
(01)        int a=10, b=20;          //准备两个变量,进行比较大小
(02)        int max=0;               //定义一个变量,保存两个变量的最大值
(03)        if (a>b) {
(04)            max=a;
            }else {
(05)            max=b;
            }
(06)        System.out.println("max="+max);
        }
    }
```

由于程序 3-3 中 a 小于 b，所以其执行路径是 1→2→3→5→6，如果修改成 a 大于 b，则程序执行路径是 1→2→3→4→6。

3. 多分支语句

采用 if…else if 形式以实现在多种情况下选择一种情况执行，它的执行逻辑如图 3-4 所示。一般的语法形式为：

```
if (逻辑表达式 1) {
    语句块 1;
} else if (逻辑表达式 2) {
    语句块 2;
} else if (逻辑表达式 3) {
    语句块 3;
}
…
else{
    语句块 n;
}
```

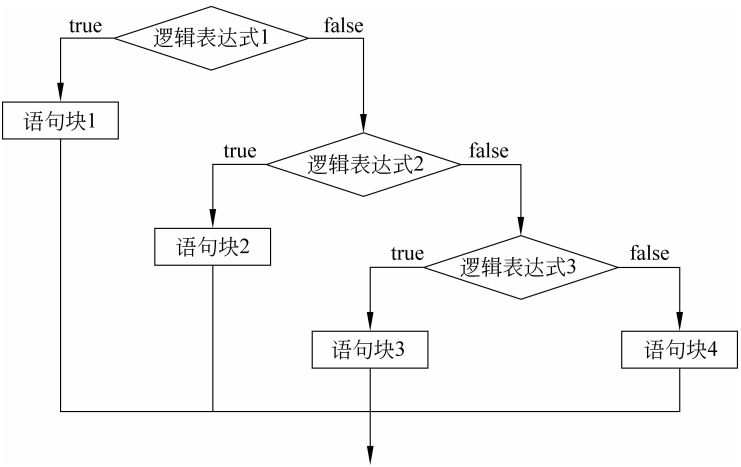

图 3-4　if…else if 多分支语句

程序碰到多分支语句时，将依次执行判断，如果碰到第一个表达式为真的情况，则执行对应的语句，然后跳至整个多分支语句之外继续执行后续程序；如果没有一个表达式为真，但存在一个 else 分支，则执行此分支对应的语句，否则，什么都不执行，直接跳至多分支语句后继续执行。

程序 3-4 描述一个应用多分支语句将百分制成绩转换为五级制的算法。

```
    //程序 3-4: 多分支结构
    public class Score {
      public static void main(String[] args) {
(01)    int score=85;
(02)    String level=null;
(03)    if (score >=80 && score<90) {
(04)        level="良好";
(05)    } else if (score >=70 && score<80) {
(06)        level="中";
(07)    } else if (score >=90) {
(08)        level="优秀";
(09)    } else if (score >=60 && score<70){
```

```
(10)        level="及格";
(11)    }else{
(12)        level="不及格";
(13)    }
(014)    System.out.println(score+"转换后的成绩是"+level);
       }
   }
```

在 score 等于 85 分的情况下，程序的执行流程是 1→2→3→4→14，如果是 90 分，则程序执行流程是 1→3→5→7→8→14。

使用这种多分支语句需要注意，每个 else if 分支都是在前面判断不满足的情况下再次判断的。

3.3.2 switch 语句

switch 语句是另外一种多分支情况选择语句，允许程序员在更多情况下选择执行不同的程序逻辑，switch 语句的语法如下：

```
switch (expr){
    case value1:
       statements;
       break;
    case value2:
       statements;
       break;
    default:
       statements;
       break;
}
```

(1) 在 switch (expr) 语句中，expr 一般来说与 int 类型赋值兼容的，允许的类型包括 byte、short 或 char 以及枚举类型；不允许使用结果类型为浮点或 long 等表达式，而且表达式必须放在小括号中。从 Java SE7 版本开始，expr 也可以是 String，即 switch 支持 String 类型的匹配。

(2) switch 的判断语句必须放在大括号中。

(3) case 后的 value 的类型必须和 expr 的类型应保持一致或兼容，例如都是字符型，而且所有 case 子句中的值应互不相等。

(4) switch 语句在执行时，从第一个 case 开始匹配是否相等，如相等则执行 case 语句后的语句，如果没有一个匹配成功，也没有 default 子句，则什么也不做，直接跳出 switch 语句，转而执行 switch 语句后的第一条语句。

(5) expr 的值如不能与任何 case 值相匹配时，可选默认符(default)指出了应该执行的程序代码。default 也是一种情况，作用和其他 case 是一样的。default 并不是必须存在的，可以省略，位置也可以放在 switch 结构中的任何顺序上，但默认位置放在最后。

(6) break 的作用是终止一个 case 语句的执行，转而执行 switch 语句后的第一条语句。

如果没有用 break 语句明确结束，并且还有其他 case 语句未执行，则程序将继续执行下一个 case，且不检查 case 后的值是否和 switch 的表达式结果是否匹配。也就是说，当一个粗心的程序员忘了用 break 语句结束一种 case 下的语句执行时，程序就不由自主地顺序执行下一个紧接的 case，直到整个 switch 结束或碰到一个 break 语句。

程序 3-5 用 switch 语句改写了程序 3-4，可以看出，利用 switch 可以使得程序在描述多分支条件下的决策时显得逻辑更清晰。

```
    //程序 3-5: switch 多分支结构
    public class Switch {
        public static void main(String[] args) {
(01)        int score=80;
(02)        String level=null;
(03)        char c=score >=80 && score<90 ? 'B'
                : score >=70 && score<80 ? 'C' : score >=90 ? 'A'
                : score >=60 && score<70 ? 'D' : 'E';
(04)        switch (c) {
(05)            case 'B':
(06)                level="良好";
(07)                break;
(08)            case 'C':
(09)                level="中等";
(10)                break;
(11)            case 'A':
(12)                level="优秀";
(13)                break;
(14)            case 'D':
(15)                level="合格";
(16)                break;
(17)            case 'E':
(18)                level="不合格";
(19)                break;
            }
(20)        System.out.println(score+"转换后的成绩是"+level);
        }
    }
```

由于 switch 表达式类型的限制，程序 3-5 只能利用条件表达式将成绩所在的区间转换为字符才能利用 switch 语句，如第 3 行语句。

当成绩 score 为 80 分时，其 switch 语句执行路径为 4→5→6→7→20，当执行到第 7 行，程序碰到了 break 语句，则退出 switch 语句，执行其后的第 20 行语句。当成绩 score 为 90 分时，其执行路径为 4→5→8→11→12→13→20。

可以安排不同的 case 执行相同的语句，当然这需要这几个 case 顺序排在一起，而且程序员要故意忘记写每个 case 所需的 break 语句，如下面的例子所示：

```
    char answer='N';
```

```
(01)  switch (answer) {
(02)      case 'Y':
(03)      case 'y':
(04)          System.out.println("Yes is selected");
(05)          break;
(06)      case 'N':
(07)      case 'n':
(08)          System.out.println("No is selected");
(09)          break;
  }
```

当 answer 变量的值为 N 时，执行路径 1→2→3→6→8→9，此时忽略了程序第 7 条语句的判断，直接执行了第 8 条语句。

3.4 循环结构

循环语句使语句或块的执行得以重复进行，例如，将一批学生的成绩从百分制转换为五分制，或者找出 100～200 之间所有能被 3 整除的数等。循环中要重复执行的语句成为循环体。每循环一次也称为一次迭代。正因为循环具有这样的特点，因此，在设计循环程序时，必须保证循环能够在有限次的循环后结束，避免出现死循环的情况。

Java 编程语言支持 3 种循环构造类型：for，while 和 do 循环。for 和 while 循环是在执行循环体之前测试循环条件是否满足，而 do 循环是在执行完循环体之后测试循环条件。这就意味着 for 和 while 循环可能连一次循环体都未执行，而 do 循环将至少执行一次循环体。

3.4.1 for 循环

for 循环的语法如下：

```
for (init_expr; boolean testexpr; alter_expr){
    loop body;
}
```

例如下面的代码段循环输出了 0～9 的数值。

```
for (int i=0; i<10; i++) {
    System.out.println("i="+i);
}
```

for 循环的执行流程参见图 3-5。

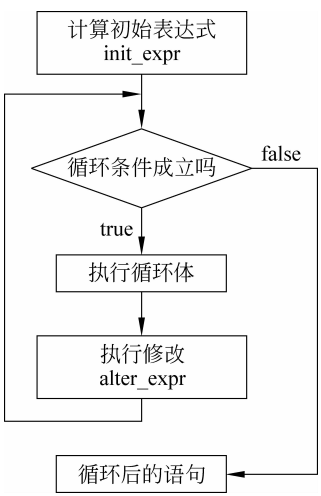

图 3-5 for 循环执行流程

(1) for 循环语句的控制出现在 for 之后的小括号中，包含了 3 个用分号隔开的部分。

(2) 第一部分 init_expr(初始化表达式)，在开始循环之前执行，只有这一次被执行的机会，通常用作循环变量的初始化。

(3) 第二部分 boolean testexpr(循环条件检查)，每次循

环之前，都要执行这个检查，表达式为 true，则进行本次循环，否则结束循环转到循环后的第一条语句继续执行程序。

(4) 第三部分 alter_expr(修改表达式)，每次循环过程执行之后，都要执行这个修改，然后再执行第二部分的循环条件检查，判断是否要进行下次循环。

下面的例子 3-6 就是一个最普通的 3 个部分都有的 for 循环的应用，程序输出了 100～200 之间所有能被 3 整除的整数。

```
//程序 3-6: 一个简单的循环应用
public class ForDemo {
    public static void main(String[] args) {
    for(int i=100;i<200;i++){
        if(i%3==0){
            System.out.print(i+"\t");
        }
      }
      System.out.println("程序结束!");
    }
}
```

由于问题要求输出 100～200 间能被 3 整除的数，用循环是最简单的方法，第 1 行使用了 for 语句，初始表达式定义了一个循环变量 i，初值为 100，实现问题从 100 开始的要求，循环的条件是 i 不能超过 200，每次循环后，利用 i＋＋对变量 i 实现加 1 操作，开始下一轮循环。

判断一个数是否能被 3 整除，程序采用了求余运算，第 2 行中使用了 i%3＝＝0 表达式，i%3 表示用循环变量 i 对 3 求余，如果余数为 0，则表示变量 i 可以被 3 整除。

这个程序使用了循环语句嵌套分支语句的结构来实现问题的要求，分支语句在这里充当了循环语句的循环体。

下面描述了部分循环的过程。

第 1 次循环：i 等于 100，i%3!＝0，不输出，i＋＋，i 的值变为 101；

第 2 次循环：i 等于 101，i%3!＝0，不输出，i＋＋，i 的值变为 102；

第 3 次循环：i 等于 102，i%3＝＝0，输出 i，i＋＋，i 的值变为 103；

…

第 100 次循环：i 等于 199，i%3!＝0，不输出，i＋＋，i 的值变为 200；

第 101 次循环：i 等于 200，i＞＝200，不满足循环条件，循环结束，执行循环后的输出语句。

在程序的第 1 个输出语句中，使用了 i＋ "\t"这样的表达式，主要是为了输出的美观，一个转义字符“\t”表示光标定位到下一个制表位，这样可以使每个输出定位到固定的位置。

这 3 个部分的表达式可以灵活使用，例如 3 个部分可以全不要，如：for(;;)一样，但是控制循环的 3 个部分的工作并不是没有了，而是安排在了程序的其他部分，例如初始化的部分也许在 for 语句之前就已经准备好了，而循环条件的变化也可以放在循环体内，第 3 部分的工作也只能放在循环体内来控制。例如程序 3-7 描述了输出 50 个从 100 开始能被 3 整除的数，由于并不知道循环结束到哪个数值，所以程序修改如下。

```
//程序 3-7: 输出 50 个能被 3 整除的数
public class ForDemo50 {
    public static void main(String[] args) {
        int count=0;
        for (int i=100;; i++) {
            if (i%3==0) {
                count++;
                System.out.print(i+"\t");
            }
            if (count==50) {
                break;          //退出循环
            }
        }
        System.out.println("程序结束!");
    }
}
```

程序 3-7 没有明确的循环条件,但是在循环体中,利用了另一个计数变量 count,每判断出一个被 3 整除的数,count 就加 1,程序利用分支语句判断 count 是否达到 50,如果达到,就执行 break 语句退出循环。

3.4.2 while 循环

while 循环称为“当型循环”,也就是当循环条件成立时才执行循环体。while 循环的语法是:

```
while (布尔表达式){
    循环体;
}
```

布尔表达式是循环的条件,每次循环开始时均要检查此布尔表达式是否为 true,如果为 true,则执行循环体内的语句,为 false 则结束循环,继续执行 while 循环后的语句。其执行过程如图 3-6 所示。

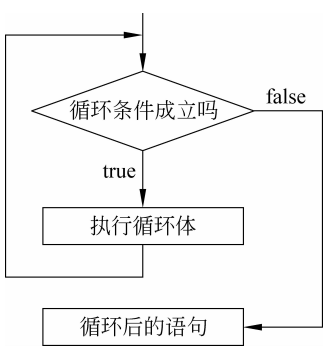

图 3-6 while 循环执行流程

和 for 循环相比,控制循环次数的布尔表达式的改变需要在循环体中进行处理以保证循环能够在有限次循环内结束。

程序 3-8 用 while 语句改写了程序 3-7。

```
//程序 3-8: 用 while 输出 50 个能被 3 整除的数
public class WhileDemo {
    public static void main(String[] args) {
    int count=0;
    int i=100;
    while (count<50) {
        if (i%3==0) {
            count++;
```

```
            System.out.print(i+"\t");
        }
        i++;          //设置每次循环变量 i 增加 1
    }
    System.out.println("程序结束!");
    }
}
```

程序 3-8 中 while 语句的循环条件是变量 count 的值小于 50，while 语句没有 for 语句那样的对循环变量进行修改的固定地方，所以需要在循环体内处理，在 if 分支语句块内，第 5 行语句 count＋＋实现了循环变量的修改，每获得一个能被 3 整除的数，则变量 count 加 1，同时每循环一次，变量 i 的值加 1，以达到下一次对新的数值进行判定的目的。

3.4.3 do 循环

do 循环有时也被称为“直到型循环”，这种循环是先执行循环体，然后再判断循环条件是否成立，它的执行流程如图 3-7 所示。语法形式如下：

```
do {
      循环体;
}while (布尔表达式);
```

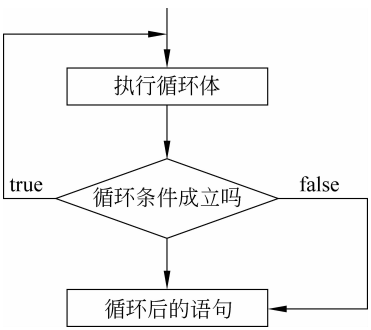

图 3-7　do 循环执行流程

do 循环的循环条件在循环体之后，这意味着循环体至少要执行一次，这是和其他两种循环最大的差别。

do 循环每次执行完循环体后，判定循环条件的结果是否为 true，来决定是否继续执行循环。

循环条件 while(布尔表达式)后跟语句结束符号。

程序 3-9 用 do 循环改写了程序 3-8。

```
//程序 3-9: 用 do 语句输出 50 个能被 3 整除的数
public class DoDemo {
    public static void main(String[] args) {
        int count=0;
        int i=100;
        do{
            if (i%3==0) {
                count++;
                System.out.print(i+"\t");
            }
            i++;          //设置每次循环变量 i 增加 1
        }while (count<50);
        System.out.println("程序结束!");
    }
}
```

由于该问题需要找出 50 个目标数值，因此，可以采用 do 循环解决，将循环条件放在了

循环体之后。当然,为了保证程序正常的运行,循环变量需要在循环体中加以控制。

本质上,3 种类型的循环并没有差别,可以相互替换。作为一种编程惯例,for 循环一般用于对一个值或(如集合式数组等)进行迭代语句,而 while 和 do 循环则用于满足某种特殊循环条件的情况。

3.4.4 跳转

1. break

break 语句被用来从 switch、for 等语句的块中退出,执行后续的语句。

程序 3-7 中已经演示了利用 break 语句在已输出 50 个数值时,不再循环,利用 break 语句强制退出当前循环,执行当前循环体之后的语句。

2. continue

continue 语句被用来略过循环体后续的所有语句,回到循环开始的地方,开始下一次循环。

下面的程序 3-10 是一个随机生成字母的程序,程序忽略非元音字母 AEIOU,只显示生成的每一个元音字母,当碰到生成的字母为 Q 时或者出现的元音字母达到 10 个时,结束程序。程序综合利用 break 和 continue 对程序的执行进行控制。

```
    //程序 3-10: break 和 continue 语句实例
    public class VowelCharGenerate {
        public static void main(String[] args) {
(01)        char ch='\0';
(02)        int count=0;                    //记录生成的元音字母个数
(03)        do{
(04)            double x=Math.random() * 100000;
(05)            int y=(int)x;
(06)            char z=(char)(y%26+65);  //确保生成的字符 ASCII 码在 65 和 90 之间
(07)            if(z=='Q'){              //如果产生到字母是 Q,则直接结束循环
(08)                break;               //退出当前层的循环
(09)            }
(10)            if(z!='A'&&z!='E'&&z!='I'&&z!='O'&&z!='U'){
(11)                //如果产生的字母不是元音字母,则结束本次循环,从新开始下一次的循环
(12)                continue;
(13)            }
(14)            System.out.printf(z+"\t");
(15)            count++;
(16)        }while(count<10);
        }
    }
```

理解这个程序首先需要了解如何随机产生字母,程序中 4、5、6 这 3 行语句用来产生字母。

第 4 行语句中的 Math.random()是调用了 Math 类的产生随机数的方法,这个 random 方法随机产生一个 0.0 到 1.0(不含)之间的随机数,为了保证获得一个较大的随机数,这里

将生成的随机数放大了 100000 倍。

第 5 行做了取整处理，转换为一个整数，主要为了满足第 6 行的求余计算的需要。

第 6 行中 y%26 得到的是 0～25 的整数，加上 65，则保证得到的是 65～90 的整数，而 65 是字母 A 的 ASCII 码值，90 是字母 Z 的 ASCII 码值。因为 int 和 char 类型之间可以相互转换，到此，程序就实现了随机生成大写字母的功能。

第 7～9 行程序利用分支语句判断生成的字母是否是 Q，如果是则用 break 语句直接退出所在的循环，执行循环语句后的第一条语句。

第 10～12 行程序利用分支语句判断生成的字母是否不是元音字母，如果不是则利用 continue 语句直接省略循环体后续的语句，进行下次循环条件的判定。

在排除了字母 Q 和非元音字母两种情况外，剩余的字母一定是元音字母了，所以循环体的第 14 行语句将该字母输出，第 15 行用变量 count 记录了生成的元音字母个数。

3.5 嵌套的结构

顺序、分支和循环构成了程序的基础，有时候，单一使用一种结构远远解决不了所有的问题，一般而言，3 种结构往往是混在一起使用。例如，循环体内有分支结构，如 3-10 的程序，分支语句也可以互相嵌套，分支语句内也可以嵌入循环，循环语句也可以嵌套循环语句等，从而实现复杂的程序结构。这种在一种结构中包含了另一种完整的结构，称为结构的嵌套。

程序 3-11 利用双重循环的嵌套结构实现了九九乘法表的输出。

```
//程序 3-11: 双重循环结构
public class DoubleLoop {
    public static void main(String[] args) {
        for(int row=1;row<=9;row++){          //row 控制输出行数
            for(int col=1;col<=row;col++){    //每行输出的个数等于所在的行数
                System.out.print(row+"*"+col+"="+row*col+"\t");
            }
            System.out.println();
        }
    }
}
```

双重循环的执行过程是外循环每执行一次，内循环要完全执行一遍。例如当 row 为 1 时，内循环执行 1 次，当 row 为 2 时，内循环共执行 2 次，以此类推，当 row 为 9 时，内循环共执行 9 次。

程序 3-11 的外循环利用循环变量 row 控制了乘法表的输出行数，内循环利用循环变量 col 控制每行的列数，分析乘法表可以看出第 1 行 1 列，第 2 行 2 列，第 9 行 9 列，因为有这样的规律，所以内循环的循环次数就是所在的行数，所以内循环的循环条件设为 col＜＝row。

程序的执行结果如图 3-8 所示。

```
1*1=1
2*1=2 2*2=4
3*1=3 3*2=6   3*3=9
4*1=4 4*2=8   4*3=12  4*4=16
5*1=5 5*2=10  5*3=15  5*4=20  5*5=25
6*1=6 6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
7*1=7 7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49
8*1=8 8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9 9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
```

图 3-8　利用双重循环实现九九乘法表的输出

3.6　变量的作用域

变量的作用域是指可以使用此变量的名称来引用它的有效程序区域。根据变量声明的位置不同，可以分为类变量、实例变量、方法参数、方法局部变量、异常参数等。到目前为止，程序涉及的变量都是方法内声明的局部变量，其作用域范围局限于方法的方法体内。更准确地讲，变量的作用域从它被声明时开始直到遇到声明变量的代码段的结束符“}”为止。只能在变量的作用域内访问它。如果在作用域之外访问变量，编译器将产生一个错误。

例如，程序 3-12 就是一个错误的程序：

```
    //程序 3-12: 变量的作用域范围
    public class VariableScope {
        public static void main(String[] args) {
(01)        int i=10;
(02)        {
(03            int k=10;
(04)           System.out.println("i="+i);
(05)           System.out.println("k="+k);
(06)        }
(07)        System.out.println("i="+i);
(08)        System.out.println("k="+k);
(09)    }
    }
```

按照作用域的规则，第 1 行语句声明的变量 i 其作用域范围从第 1 行开始，因为 i 所在的代码段是整个方法体，故其作用域终止于第一个匹配与代码段开始的“}”，即第 9 行。

同样的道理，变量 k 存在于从第 2 行到第 6 行的复合语句块内，所以它的作用域从第 3 行声明语句开始，到第 6 行就结束了。因此在程序的第 8 行用输出语句输出变量 k 的值时，k 这个变量已经不存在了，因此，编译器将会报告“找不到符号”错误，如图 3-9 所示。

```
C:\java\project\mytext\src\chap3>javac VariableScope.java
VariableScope.java:12: 找不到符号
符号:  变量 k
位置:  类 chap3.VariableScope
                System.out.println("k="+k);
                                   ^
1 错误
```

图 3-9　变量作用域错误

当方法体内的程序结构出现嵌套现象时，如果在外层结构中声明一个变量后，内层结构内不能再声明同名的变量，如下面的代码段所示。

```
{
    int i=10;
    {
        int i=0;          //错误,因为重复声明变量 i
        int k=10;
        System.out.println("i="+i);
        System.out.println("k="+k);
    }
    System.out.println("i="+i);
}
```

3.7 程序设计应用

3.7.1 求解素数

素数是这样的整数，它除了能表示为它自己和 1 的乘积以外，不能表示为任何其他两个整数的乘积，验证一个正整数 n(n＞＝2)是否为素数，一个最直观的方法是看在 2～n/2 中能否找到一个整数 m 能将 n 整除。若 m 存在，则 n 不是素数；若找不到 m，则 n 为素数。这是一个循环验证算法。这个循环结构用下列表达式控制：

初值：m＝2；

循环条件：m＜＝n/2；

修正：m＋＋；

即这是一个 for 结构。它的作用是穷举 2～n/2 中的各 m 值。循环体是判断 n 是否可以被 m 整除。

根据上述分析，程序 3-13 描述了求解 50 以内素数的过程。

```
//程序 3-13: 求解 50 以内的全部素数
public class PrimeApp {
public static void main(String[] args) {
        int m, n;                  //变量 n 为要判断的数字
        System.out.println("50 以内的素数有: ");
        boolean prime=true;
        for (n=2; n<=50; n++) {
            prime=true;            //用 prime 判断当前的 n 是否为素数,每次循环默认为是
            for (m=2; m<=n/2; m++) {
                if (n%m==0) {
                    prime=false;  //修改 prime 的值,用 flase 表示不是素数,
                    break;        //如果能被整除则变量 n 肯定不是素数,跳出内层循环
                }
            }
```

```
            if (prime) {            //根据 prime 的真假,决定是否输出当前的 n
                System.out.print(n+" "+"\t");     //输出素数
            }
        }
    }
}
```

在程序 3-13 中,外层 for 语句的循环变量 n 代表要判断的数字,因为素数不包括 1,所以变量 n 的取值为 2～50 的数值,内层 for 语句用来判断变量 n 表示的数值是否为素数。

内层循环变量 m 由 2 开始,每次增加 1,直到 n/2 为止,外层循环变量 n 依次除以内层循环变量 m,一旦出现余数为 0 时,表示 n 可以被一个大于 1 小于它本身的数字整除,因此,该数字一定不是素数,不需要再继续进行判断,执行 break 语句跳出内循环,或者内循环从 2 到 n/2 没有发现能被 n 整除的数字,则内循环自然结束运行。

内循环运行结束后,外循环后续的语句利用 prime 这个逻辑型变量的真假决定是否输出当前的变量 n,所以,每次外循环开始新的循环时,将 prime 置为 true 非常重要。如果执行完内层循环,变量 n 没有被 2 到 n/2 之间的某个数整除,也就是说,变量 n 是素数,则 prime 的值一直保持为 true,执行外层循环中的条件语句输出变量 n 的值。程序 3-13 的运行输出结果如下:

```
50 以内的素数有:
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47
```

3.7.2 递归

递归是一种算法设计方法,也是计算机程序设计问题中可利用的最有效的解题方法之一。在程序设计中,递归的使用相当普遍,递归的解决方法比非递归的解决方法更精妙、更简洁。有些问题看起来很复杂,但是使用递归的方法来解决却非常简单。下面用两个程序实例,说明递归的概念。

程序 3-14 用于求给定整数 n 的阶乘(n>0)。

编写一个方法 static int fac(int n),其功能是返回参数 n 的阶乘。

当求 n 的阶乘时,通常是从 1 开始,由 1!求 2!,由 2!求 3!,直至求出 n!。按这种考虑可写出如下的程序代码。

```
//程序 3-14: 不用递归法求解阶乘
public class Recurrence {
static int fac(int n) {
        int i=1, mult=1;
        while (i<=n) {
            mult=mult * i;
            i++;
        }
        return mult;
    }
    public static void main(String[] args){
```

```
        int f=fac(5);
        System.out.println("5!="+f);
    }
}
```

上述程序的基本思想是不断由已知值推出新值,直至求得解,这就是递推的方法。

再从另一个方面考虑:如果已经求得了(n－1)!,则 n!可以由(n－1)!乘 n 求得,而(n－1)!可以由(n－2)!乘 n－1 求得,以此类推,直至 1!无须计算即可直接得到,按这种考虑可写成如下的程序代码:

```
//程序 3-15:用递归法求解阶乘
public class Recursion{
    static int fac (int n) {
        if (n==1)
            return 1;
        else
            return n * fac(n-1);
    }
    public static void main(String[] args){
        int f=fac(5);
        System.out.println("5!="+f);
    }
}
```

上述程序的基本思想是不断把问题分解成规模较小的同类问题,直至问题足够小能直接求得解为止,这就是递归的方法。

在数学及程序设计方法学中为递归下的定义是,若一个对象部分地包含它自己或用它自己给自己定义,则称这个对象是递归的。对于一个方法来说,若直接或间接地调用它自己,则该方法为递归方法。递归过程的特点是结构清晰、程序简练易读、正确性容易证明,因此是程序设计的有力工具。

从上述程序设计实例可以看出,递归算法适应于这样一类问题:在对这一类问题求解的过程中得到的是和原问题性质相同的子问题,这一类问题自然可以用一个递归方法进行描述。为这一类问题设计递归算法时,通常可以先写出问题求解的递归定义。递归定义由基本项和归纳项两部分组成。基本项描述递归过程的终结状态。所谓终结状态,是指不需要继续递归而可直接求解的状态。归纳项描述了如何实现从当前状态到终结状态的转化。

3.8 程序调试和排错

3.8.1 利用断言调试程序

JDK 1.4 中引入的一个新特性之一就是断言(assert),为程序的调试提供了强有力的支持。可以在任何时候启用和禁用断言验证,因此可以在测试时启用断言而在部署时禁用断言(具体参数参见相关资料)。同样,程序投入运行后,最终用户在遇到问题时可以重新起用

断言。

断言语句有两种形式：

形式 1：

```
assert  expression;
```

这是一种简单的断言，expression 是一个布尔类型的表达式，当程序运行到断言时，计算该表达式，假如结果为 false，那么抛出一个没有详细信息说明的错误 AssertError。

形式 2：

```
assert  expression1 : expression2;
```

其中 expression1 也是一个布尔类型的表达式，但当错误发生时，expression2 将被送往 AssertionError 的构造函数，这样将会获得更详细的错误信息。程序 3-16 就是第二种形式的断言例子。

```
//程序 3-16: 一个简单的断言应用
import java.util.Scanner;
public class AssertDemo {
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        int score=0;
        for(int i=1;i<=5;i++){
            System.out.print("请输入一个 0-100 以内的整数: ");
            score=sc.nextInt();
            assert score>=0&&score<=100:"输入了错误的成绩";
        }
    }
}
```

由于程序的正确运行依赖于正确的输入。图 3-10 所示的步骤演示了程序 3-16 不用断言运行的过程。

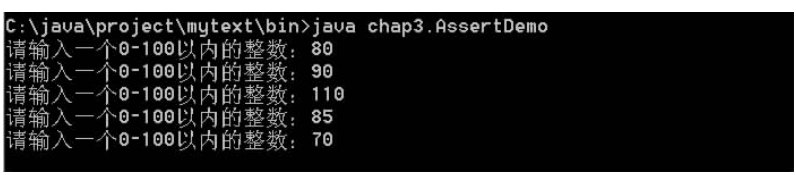

图 3-10　不用断言运行的过程

由于没有启用断言支持，因此当输入成绩为 110 时，断言并没有起到任何作用。下面，再重新运行一次，只不过这次在命令行中添加了支持断言的参数：-ea。

图 3-11 所示的过程描述了当程序 3-16 启用断言后，用命令 java -ea AssertDemo，运行时输出结果。

当运行命令参数添加了"-ea"后，当输入成绩 110 后，断言语句发挥作用，进行了一个条件表达式的判断，当发现结果为 false 时，抛出了一个异常，然后程序中止。