CHAPTER 3

第3章

程序的流程控制和数组

程序的流程就是程序执行的过程，程序语句执行的先后顺序；流程控制就是控制程序语句的执行，使得程序可以按照编程人员的设计思路执行。Java 程序的基本流程控制主要有 3 类：顺序结构、选择结构和循环结构。顺序结构中程序语句按照顺序执行，前面的代码先执行完，后面的代码再执行，没有专门的控制语句。本章重点关注选择结构、循环结构语句。此外，结合循环结构，讨论数组的定义、声明、创建、初始化和访问。

3.1 预 备 知 识

3.1.1 选择结构

选择结构就是要选择不同的分支语句来执行，这就需要条件的控制，程序的流程会根据条件的不同流向不同的分支。控制条件可以看作是分支的控制开关，条件满足，开关会自动打开，每个分支都有相应的控制条件。

1. if 语句

if 语句有以下三种基本格式：单分支 if 语句、双分支 if-else 语句和多分支 if-else-if 语句。

1）单分支 if 语句

```
if (条件表达式) {
     语句块;
}
```

当条件表达式返回结果为 true 时，执行语句块，否则什么都不执行。

2）双分支 if-else 语句

```
if (条件表达式) {
    语句块 A;
} else {
    语句块 B;
}
```

当条件表达式返回结果为 true 时，执行语句块 A，否则执行语句块 B。

3）多分支 if-else-if 语句

```
if (条件表达式 1) {
   语句块 A;
} else if (条件表达式 2) {
   语句块 B;
} else if (条件表达式 3) {
   语句块 C;
}…
```

当条件表达式 1 返回 true 时，执行语句块 A；否则当条件表达式 2 返回 true 时，执行语句块 B；否则当条件表达式 3 返回 true 时，执行语句块 C……但是只有第一次条件表达式为 true 时对应的语句块会被执行。当控制条件很多时，使用这种方式就不方便了，需要改造成另一种条件控制语句，就是 switch-case。

2. switch 语句

switch 语句是根据表达式的值，选择执行多个语句块中的一个。此时表达式的值不能是 boolean 类型，而只能是 byte、char、short 或 int 类型。

```
switch (表达式) {
    case 常量表达式 1: 语句块 A; break;
    case 常量表达式 2: 语句块 B; break;
    case 常量表达式 3: 语句块 C; break;
    …
    default : 语句块 N;
}
```

当表达式的值满足 case 后面相应常量表达式的值时，会执行其后面对应的语句块。在使用 switch-case 时需要注意以下几点：

- 表达式的值只可以转换成 int 类型的类型或枚举常量，常用的是 byte、char、short 和 int 类型，不可以是 String 类型，这和 C＃不同；
- 语句块后面如果不使用 break，程序将会从此语句块开始往后执行，直到遇见 break 或 switch 结束，而不管 case 后面的常量表达式是不是匹配，所以一般语句块后面都需要加上 break；
- default 的作用是默认分支，当以上常量表达式都不匹配时，执行其后的语句块，也可以不使用，其位置可以放在前面，也可以放在后面。

3. 选择结构的嵌套

上面介绍的几种条件语句都可以相互嵌套使用，选择结构的嵌套是指在条件控制语句的语句块中又包含了条件控制语句，常见的嵌套形式如下：

```
if (表达式 1) {
    if (表达式 11){
        语句块 A;
    }else {
```

```
            语句块 B;
        }
    }else {
        if (表达式 21) {
            语句块 C;
        }else {
            语句块 D;
        }
    }
```

选择结构的嵌套和多分支 if-else-if 不同，选择嵌套的条件是不同级别的，当外层条件满足后才能进入内层，继续判断内层的条件，如果满足，则执行相应的语句块，语句块中可以继续嵌套 if 控制语句。

if 语句之间，if 语句和 switch 语句，以及 switch 语句和 switch 语句之间都可以相互嵌套，以便完成复杂的程序流程控制。如下面的 switch 语句和 if 语句的嵌套形式：

```
switch(表达式){
    case 常量表达式:  if(条件表达式){
                        语句块 A;
                      }else{
                        语句块 B;
                      }
                      break;
    case 常量表达式:  语句块 C; break;
    …
}
```

选择结构的嵌套使得程序流程控制更加灵活多变，但是随之而来的问题也增加了，在使用选择结构的嵌套时需要注意以下几点：

- 养成良好的代码书写习惯，保持代码缩进，以提高代码的可阅读性；
- 在不需要嵌套也可以实现流程控制情况下，尽量不使用嵌套，如果使用一定要理清思路，不要出现逻辑漏洞；
- 嵌套时注意 if-else 的对应关系，else 总是与离它最近的 if 匹配，else 不能单独出现，建议使用{}将代码语句组成语句块。

3.1.2 循环结构

相比选择结构的选择执行分支代码块，循环结构的作用是重复执行一段代码块，直到满足循环终止条件。实现循环结构的循环控制语句有 3 种：for 语句、while 语句和 do-while 语句。另外还有一种增强型的循环，for/in 循环，是在 Java 5.0 提出的一个比较方便的特性，使得遍历数组和集合更加简单，

对于一个循环结构，一般应包括四个基本构成部分：初始化部分、循环条件判断部分、循环体部分和循环条件迭代部分(循环条件的增或减)。上面所提到的 3 种循环控制语句，都有这 4 个基本部分，只不过它们的组织方式不同，因此 3 种循环语句也是可以相互转换的。

1. for 循环

for 循环经常用在已知循环次数的情况下，当然这不是固定的，它是一种比较标准的循环控制语句，一般形式如下：

```
for (初始化部分; 循环条件判断部分; 循环条件迭代部分){
        循环体部分;
}
```

for 循环语句的执行顺序是：首先执行“初始化部分”；然后检验“循环条件判断部分”；如果判断的结果是 true，则执行“循环体部分”；接着执行“循环条件迭代部分”；随后再检验“循环条件判断部分”，如果判断的结果是 true，则继续执行“循环体部分”；直到判断结果是 false，结束 for 循环。

上面列出的是 for 循环的最基本形式，也是最常用的形式，但不是固定的，也可以有其他形式。比如，“初始化部分”也可以在写在 for 语句之前，即在循环开始之前就完成了控制条件的初始化工作；“循环条件判断部分”也可以不写，此时循环没有控制部分，将一直执行“循环体部分”，这就是死循环，虽然可以使用 if 和辅助流程控制语句实现跳出循环，但这就失去使用 for 循环的意义了；“循环条件迭代部分”可以写在“循环体部分”之后，也可以与“循环条件判断部分”合并。对于这些变化形式一般不建议使用，如果出现这种情况，可以使用其他循环语句。使用 for 循环需要注意以下几点：

- for 后的 3 个部分都可以省，但括号内的两个分号不能省；
- “初始化部分”和“循环条件迭代部分”可以是多条语句，用逗号分开；
- 循环体是多条语句时，需要用{}组成语句块。

2. while 循环

while 循环经常用在已知循环条件，但不知道需要循环多少次的情况下，这是它与 for 循环的区别所在。while 循环的一般形式如下：

```
初始化部分;
while (循环条件判断部分){
        循环体部分;
        循环条件迭代部分;
}
```

while 循环语句的执行顺序是，先检验“循环条件判断部分”，检验的结果是 true，则执行“循环体部分”和“循环条件迭代部分”；检验的结果是 false，跳出 while 循环。

同 for 循环语句一样，while 循环语句也有变形形式，比如，“初始化部分”和“循环条件判断部分”同时进行；“循环条件迭代部分”与“循环条件判断部分”同时进行等。使用 while 循环需要注意以下几点：

- “初始化部分”和“循环条件迭代部分”不是必需的；
- 当首次检验“循环条件判断部分”就返回 false 时，“循环体部分”将不被执行，这一点区别于 do-while 循环。

3. do-while 循环

do-while 循环的应用场景和 while 循环类似，多用在知道循环的条件，而不知道次数的情况下，但与 while 循环不同的是，do-while 循环不管循环的条件如何，总会执行一次循环体。一般形式如下：

```
初始化部分;
do{
        循环体部分;
        循环条件迭代部分;
}while(循环条件判断部分);
```

do-while 循环语句的执行顺序是，首先执行“循环体部分”和“循环条件迭代部分”，然后检验“循环条件判断部分”，如果检验的结果是 true，则继续执行“循环体部分”和“循环条件迭代部分”；否则跳出 do-while 循环。

do-while 循环和 while 循环比较类似，上面的形式也不是固定的，可以根据具体情况有所变化。使用 do-while 循环需要注意以下几点：

- “初始化部分”和“循环条件迭代部分”不是必需的；
- while(循环条件判断部分)需要以分号结尾，初学者很容易漏掉；
- do-while 循环一般用在“循环体部分”需要初始化的场景下。

4. 循环结构的嵌套

循环结构的嵌套，指的是在一个循环语句的循环体中又包含了另一个循环语句。嵌套的循环在执行过程中，外层循环执行一次，内层循环就要循环到其循环控制条件不满足为止，然后退到外层循环，外层循环条再执行一次，内层循环又循环到其循环控制条件不满足为止，退出到外层循环，如此反复，直到外层循环的循环控制条件也不满足为止。

同种循环语句间可以嵌套，不同的循环语句间也可以相互嵌套，如下面的 for 语句的嵌套、for 与 while 的嵌套：

```
for (初始化部分; 循环条件判断部分; 循环条件迭代部分){
    for (初始化部分; 循环条件判断部分; 循环条件迭代部分){
     循环体部分;
    }
}

for (初始化部分; 循环条件判断部分; 循环条件迭代部分){
     while (循环条件判断部分){
      循环体部分;
    }
}
```

循环结构的嵌套可以处理复杂的流程控制，但是也会让程序变得复杂化，在使用循环嵌套时需要注意以下几点：

- 保持代码的缩进，区分内外层循环；
- 循环嵌套的层次不宜过深，一般最多 2～3 层，过多会影响程序的执行效率；

- 内层循环声明的变量会覆盖外层的同名变量，一般情况各层循环应该使用独立变量作为循环控制条件。

3.1.3　辅助流程控制语句

辅助流程控制语句指的是 break、continue 及 return，使用它们可以改变原有的程序流程。

1. break 语句

前面已经提到 break，可以跳出 switch，除此之外 break 还常用来跳出 for 循环语句，while 循环语句和 do-while 循环语句，去执行紧跟在循环后面的其他语句。另外 break 还可以配合标签使用，而直接跳出标签所指示的代码块。

2. continue 语句

continue 语句常用在 for 循环语句，while 循环语句和 do-while 循环语句中，用以结束本次循环，进入下一次循环，这点是与 break 不同的，它不是跳出循环。continue 语句也可以配合标签使用，用以重新从标签指示的代码块执行，只是标签只能设在 for 之前。

3. return 语句

return 语句的作用是从当前位置返回，其后的代码不再执行。return 后面可以加表达式，也可以不加。return 语句一般放在代码块的最后。

3.1.4　流程控制语句的嵌套

在程序设计过程中，如果是简单的流程控制，用单个控制语句就能够完成；如果是复杂的流程的控制，就需要多个控制语句相互配合，也就是相互嵌套，共同完成。在解决实际问题的过程中，大多数流程控制都是需要多个控制语句，因此熟练掌握与应用流程控制语句的嵌套非常重要，流程控制语句的嵌套也是难点。

流程控制语句的嵌套可以分为选择结构控制语句的嵌套，循环结构控制语句的嵌套和二者的混合嵌套。

3.1.5　数组

1. 数组的定义

数组是由相同类型的相关数据组成的数据结构，存储在连续的内存单元中。

2. 数组的声明

1）一维数组的声明

数组声明时，不需要指明数组的长度。例如：

```
int[] numbers;
```

2）二维数组的声明

二维数组实际上是数组的数组，即每一个元素是一个一维数组。例如：

```
int[][] numbers;
```

3. 数组的创建

当声明一个数组时，并没有为数组分配任何存储空间。只有当创建数组对象的时候才会分配存储空间。数组创建的操作，最直接的方式是使用 new 关键字来实现。

1）一维数组的声明

在上述声明一维数组的基础上，为其分配空间：

```
numbers = new int[3]; //表示该一维数组包含 3 个元素
```

2）二维数组的声明

在上述声明二维数组的基础上，为其分配空间：

```
numbers = new int[3][4];
```

表示该二维数组包含 3 个元素，每个元素（一维数组）各自包含 4 个元素。或者：

```
numbers = new int[3][];
numbers[0] = new int[5];
numbers[1] = new int[3];
numbers[2] = new int[2];
```

表示该二维数组包含 3 个元素，每个元素（一维数组）各自所包含的元素个数不同（分别为 5、3 和 2）。

4. 数组的初始化

在默认情况下，一旦数组被创建，数组中的每个成员都有一个初始值。如果数组是数值型的，成员则初始化为 0；如果数组是布尔型的，成员则初始化为 false；如果数组成员是引用类型的，成员则初始化为 null。

1）声明时初始化

一维数组：

```
int[] numbers = {1, 2, 3};
```

二维数组：

```
int[][] numbers = {{1,2,3},{4,5,6}};
```

2）创建时初始化

一维数组：

```
int[] numbers = new int[]{1, 2, 3};
```

二维数组：

```
int[][] numbers = new int[][]{{1,2,3},{4,5,6}};
```

以下是数组初始化的错误方式：

```
int[ ] numbers; numbers = {1, 2, 3};
int[ ] numbers = new int[3]{1, 2, 3};
int[][] numbers = new int[][]{1,2,3,4,5,6};
int[][] numbers = new int[2][]{{1,2,3},{4,5,6}};
```

5. 访问数组元素

数组元素可以通过数组的下标进行访问。数组下标的取值范围是从 0 到数组的长度减 1 之间。其中数组的长度可以通过“数组名.length”来获得。例如：

一维数组：

```
numbers[0] = 1; numbers[1] = 2; numbers[2] = 3;
```

二维数组：

```
numbers[0][0] = 1; numbers[0][1] = 2; numbers[0][2] = 3;
```

3.2　实验 1　计算航空旅客行李收费

【实验目的】

(1) 掌握选择结构及其嵌套的使用，能够解决实际问题。

(2) 学会使用命令行参数。

【实验内容】

假设某航空公司要求编写一种可以计算乘客行李收费的算法，具体要求如下：乘客可以免费携带重量不超过 30kg 的行李，当重量超过时，头等舱内的乘客的超重部分按 4 元/kg 收费，其他舱的乘客的超重部分按 6 元/kg 收费，如果乘客是外国人收费标准增加一倍，如果乘客是残疾人士收费标准降低一半。

【参考代码】

```
public class Charge {
    public static void main(String[ ] args) {
        int weight = 0;                          //行李的重量
        float charge = 0.0f;                     //收取的费用
        int passenger = 0;                       //机舱,0 是头等舱,1 是其他舱
        int nationality = 0;                     //国别,0 是国内,1 是国外
        int normal = 0;                          //乘客是否为正常人士,0 正常,1 残疾

        weight = Integer.parseInt(args[0]);      //命令行参数 String 类型转换为 int 型
        passenger = Integer.parseInt(args[1]);
        nationality = Integer.parseInt(args[2]);
        normal = Integer.parseInt(args[3]);
        if(weight <= 30){
            charge = 0.0f;                       //如果重量不足 30kg,免费托运
        }else{
            if(passenger == 0){
```

```
                if(nationality == 0){
                    if(normal == 0){          //超重,头等舱,国内,正常乘客的收费标准
                        charge = (weight - 30) * 4;
                    }else {                   //超重,头等舱,国内,残疾乘客的收费标准
                        charge = (weight - 30) * 2;
                    }
                }else{
                    if(normal == 0){          //超重,头等舱,国外,正常乘客的收费标准
                        charge = (weight - 30) * 8;
                    }else{                    //超重,头等舱,国外,残疾乘客的收费标准
                        charge = (weight - 30) * 4;
                    }
                }
            }else{
                if(nationality == 0){
                    if(normal == 0){          //超重,其他舱,国内,正常乘客的收费标准
                        charge = (weight - 30) * 6;
                    }else{                    //超重,其他舱,国内,残疾乘客的收费标准
                        charge = (weight - 30) * 3;
                    }
                }else{
                    if(normal == 0){          //超重,其他舱,国外,正常乘客的收费标准
                        charge = (weight - 30) * 12;
                    }else{                    //超重,其他舱,国外,残疾乘客的收费标准
                        charge = (weight - 30) * 6;
                    }
                }
            }
        }
        System.out.println("Charge is : " + charge);
    }
}
```

程序的运行结果如图 3.1 所示。

```
Charge is : 12.0
```

图 3.1　计算航空旅客行李收费

【程序解析】

该段程序中,关于行李的重量(weight)、机舱舱位(passenger)、国别(nationality)、乘客是否为正常人士(normal)几个变量,此处使用命令行参数(String[] args)对其赋值。在使用 Java 命令执行含有 main 方法的类时,其后可跟随若干参数,这些参数会以 String 类型自动以此赋值数组 args 的每一个元素。使用数组 args 的某个元素值时,可利用对应包装类的静态方法进行类型转换,如将 String 类型转换为 int 型(Integer.parseInt( ))。

该程序算法代码完全是由 if 语句的嵌套实现的,在真正应用环境编写时,这不算是最佳代码,还有其他方式实现,即使是 if 语句嵌套,也不是唯一的形式,这段代码的关键是要清楚选择语句在嵌套时的逻辑关系。当嵌套的层次很深时,如上面的代码所示,程序的执行

效率会有所下降，这就需要改变选择判断的形式；程序的可读性也会下降，这就需要借助其他工具，如判定表和判断树，关于判定表和判断树，请参考软件工程相关书籍。

3.3　实验 2　输出九九乘法表

【实验目的】

掌握循环结构及其嵌套的使用，能够解决实际问题。

【实验内容】

设计和实现九九乘法表。

【参考代码】

```
public class Test {
    public static void main(String[ ] args){
        for(int i = 1; i < 10; i++){
            for(int j = 1; j <= i; j++){
                System.out.print(i + "X" + j + "=" + i * j + " ");
            }
            System.out.println();
        }
    }
}
```

程序的运行结果如图 3.2 所示。

```
1X1=1
2X1=2 2X2=4
3X1=3 3X2=6 3X3=9
4X1=4 4X2=8 4X3=12 4X4=16
5X1=5 5X2=10 5X3=15 5X4=20 5X5=25
6X1=6 6X2=12 6X3=18 6X4=24 6X5=30 6X6=36
7X1=7 7X2=14 7X3=21 7X4=28 7X5=35 7X6=42 7X7=49
8X1=8 8X2=16 8X3=24 8X4=32 8X5=40 8X6=48 8X7=56 8X8=64
9X1=9 9X2=18 9X3=27 9X4=36 9X5=45 9X6=54 9X7=63 9X8=72 9X9=81
```

图 3.2　输出九九乘法表

【程序解析】

因为在九九乘法表中，循环的次数都是已知的，所以首先 for 循环语句。上面的代码是两层 for 循环的嵌套，外层控制行，从 1～9，内层控制列，从 1 到外层循环的控制变量 i，即 j 的初始化部分是“j＝1；”，循环控制条件的判断是“j＜＝i；”。“System. *out*. print(i ＋ "X" ＋ j ＋ "＝" ＋ i ＊ j ＋ " ")；”构建算式输出，但是没有换行，所以在内层循环完成后，需要再增加一个“System. out. println()；”，用来输出一个换行。

在循环结构中，循环控制语句的循环控制条件是关键，如果循环控制条件使用不当，会造成意想不到的问题，尤其是用循环在遍历数组和集合元素的时候，如果循环的控制语句不

正确，会造成溢出错误或是没有全部遍历。

下面的代码也是实现输出九九乘法表的功能，请读者对比前面的代码，比较两次的不同。

```
public class Test {
    public static void main(String[ ] args){
        for(int i = 9; i >= 0; i -- ){
            for(int j = 1; j <= i; j++){
                System.out.print(i + "X" + j + "=" + i * j + " ");
            }
            System.out.println();
        }
    }
}
```

这段代码的运行结果如图 3.3 所示。

```
9X1=9 9X2=18 9X3=27 9X4=36 9X5=45 9X6=54 9X7=63 9X8=72 9X9=81
8X1=8 8X2=16 8X3=24 8X4=32 8X5=40 8X6=48 8X7=56 8X8=64
7X1=7 7X2=14 7X3=21 7X4=28 7X5=35 7X6=42 7X7=49
6X1=6 6X2=12 6X3=18 6X4=24 6X5=30 6X6=36
5X1=5 5X2=10 5X3=15 5X4=20 5X5=25
4X1=4 4X2=8 4X3=12 4X4=16
3X1=3 3X2=6 3X3=9
2X1=2 2X2=4
1X1=1
```

图 3.3　重新输出九九乘法表

【程序解析】

两次九九乘法表的不同就在于外层循环的控制条件不同，for(i＝1;i＜10;i＋＋)是从 1～9 的循环，每次增 1，由于外层循环是控制输出行的，所以它所输出的九九乘法表是从 1～9 的；for(i＝9;i＞＝0;i－－)是从 9～1 的循环，每次减 1，所以它所输出的九九乘法表是从 9～1 的。

3.4　实验 3　输出 100 以内的质数

【实验目的】

(1) 掌握选择结构和循环结构以及二者的嵌套使用，能够解决实际问题。

(2) 了解和灵活使用 continue 和 break 等语句。

【实验内容】

质数又称为素数，指的是只有 1 和它本身这两个因数的自然数。实验要求输出 100 以内的所有质数，只需用循环依次检验每个数是否只有 1 和它本身两个因数即可，因为循环次数已知，所以采用 for 循环语句；检验一个数是否质数，只需在 2 到该数平方根之间寻找因

数即可，如果找到，则不是质数，否则是质数。

【参考代码】

```
public class Prime {
    public static void main(String[ ] args) {
        int flag = 1;                                    //计数器
        for(int i = 2;i <= 100;i++) {
            // Math.sqrt(i)的返回类型是 double,需要转换成 int 类型
            for(int j = 2;j <= (int)Math.sqrt(i) + 1; j++) {
                if(i == 2) {                             //2 被默认是质数
                        System.out.print(j + " is a prime\t");
                        continue;
                }
                if(i % j == 0) break;                    //余数为 0 说明不是质数,跳出循环
                if(j >(int)Math.sqrt(i)){
                        if(flag++ % 5 == 0) System.out.println();  //输出 5 个换行
                        System.out.print(i + " is a prime\t");
                }
            }
        }
        System.out.print("\nSummation :" + flag);
    }
}
```

程序运行的结果如图 3.4 所示。

```
2 is a prime    3 is a prime    5 is a prime    7 is a prime    11 is a prime
13 is a prime   17 is a prime   19 is a prime   23 is a prime   29 is a prime
31 is a prime   37 is a prime   41 is a prime   43 is a prime   47 is a prime
53 is a prime   59 is a prime   61 is a prime   67 is a prime   71 is a prime
73 is a prime   79 is a prime   83 is a prime   89 is a prime   97 is a prime
Summation :25
```

图 3.4　输出 100 以内的质数

【程序解析】

上面的程序采用两次循环实现，外层循环用来从 2～100 遍历，内层循环用来寻找因数。外层循环从 2 开始(1 不是质数)，每循环一次，内层循环都要去检验一次，检验从 2 开始，到被检验数的平方根加 1 为止即可，没有必要检验到自身。从 2 到 Math. sqrt(i)＋1 之间如果存在 i 的因数，即 i％j＝＝0 值为 true，则 i 不是质数，内存循环不必继续，直接 break 即可；如果内层循环从 2 一直循环到 Math. sqrt(i)＋1，即 j＞ Math. sqrt(i)＋1，内层循环结束，都没有找到一个因数，则 i 是质数，直接输出。其中 Math. sqrt()是 Math 类的一个静态方法，用于求一个数的平方根，返回 double 类型的值，在程序中需要强制转换成 int 类型。局部变量 flag 是计数器，用来保存质数的个数，“if(flag＋＋％5＝＝0)System. out. println();”用于输出换行符，每 5 个质数输出在一行。

continue 语句会结束其后的代码，返回标签指定的代码块，与 break 语句不同，请读者

仔细分析。对程序的流程控制需要注意以下几点：

- 使用流程控制语句嵌套时，要理清逻辑关系，当程序的流程比较复杂时可以借助流程图，先把程序的流程确定，再翻译成代码；
- 使用循环语句时，要确定循环语句的控制条件以及循环的起始和结束，即循环的边界问题；
- 当多层循环嵌套时，尽量将长循环放在内层，短循环放在外层。

上面的代码可以实现求质数，但不是唯一形式，读者可以参考该程序写出自己的求解质数的程序。下面的这段代码也实现了求质数的功能，请读者自己解析：

```
public class Prime {
    public static void main(String[ ] args) {
        boolean flag;
        for(int i = 2; i <= 100; i++) {                    //外层循环
            flag = true;
            for(int j = 2; j <= i / 2; j++) {              //内层循环
                if(i % j == 0) {
                    flag = false;
                    break;
                }
            }
            if(flag) {
                System.out.print(i + " ");
            }
        }
    }
}
```

3.5 实验 4 求解猴子吃桃问题

【实验目的】

(1) 使用循环结构解决实际问题。

(2) 练习使用逆向思维和递归思想来分析解决问题。

【实验内容】

编写程序求解猴子吃桃问题。有一只猴子摘了很多桃子，当天便吃了一半，感觉不过瘾，便多吃了一个。第二天将剩下的桃子吃了一半，并又多吃了一个。以后每天都吃前一天剩下的一半并多一个，到第十天再吃时就只剩下了一个桃子。求解猴子刚开始摘了多少桃子。

仔细分析该问题中的规律：假设每天有桃子 num 个，当然 num 会每天都变化一次，可以确定的是第十天 num＝1，从而可以求出第九天的 num 是第十天的 num 加 1 然后乘以 2，即 num＝(num＋1)＊2；一共是 10 天，循环的次数已知，故采用 for 循环语句。

【参考代码】

```
public class Monkey {
    public static void main(String[ ] args) {
```

```
        int num = 1;                                    //第十天桃子的个数
        for(int i = 10; i >= 1; i--) {
            System.out.println("Day " + i + " : " + num);
            num = (num + 1) * 2;                        //求前一天的桃子
        }
    }
}
```

程序的运行结果如图 3.5 所示。

```
Day 10 : 1
Day 9 : 4
Day 8 : 10
Day 7 : 22
Day 6 : 46
Day 5 : 94
Day 4 : 190
Day 3 : 382
Day 2 : 766
Day 1 : 1534
```

图 3.5　求解猴子吃桃问题

【程序解析】

程序代码很简单，关键是要想清楚其中的逻辑关系，前一天剩余桃子的个数是第二天剩余桃子的个数加 1 再乘以 2，所以可以从最后一天剩余桃子数量为 1 开始，循环计算 num＝(num＋1)＊2，直到第一天即循环到第 10 次即可。上面的 for 循环是从 i＝10 开始循环的，循环的条件是 i＞＝1，其中的关键语句是 num＝(num＋1)＊2，将当天的桃子个数加 1 再乘以 2 后赋给前一天，直到 i＝1 为止。

解决一个实际问题并没有严格固定的套路，只要将自己的想法翻译成代码即可，在翻译的过程中按照语法的要求，注意合理的流程控制即可。对于本实验也可以采用递归实现，参考程序如下：

```
public class Monkey {
    public static void main(String  args[ ]){
        int day = 10, sum = 1;                          //day 天数, sum 当天的桃子个数
        peach(day, sum);
    }
    public static void peach(int d, int s){
        if(d == 1){                                     //递归到第一天时, 返回
            System.out.println("Day " + d + " : " + s);
            return;
        }
        else{
            System.out.println("Day " + d + " : " + s);
            d = d - 1;                                  //天数减 1
            s = (s + 1) * 2;                            //桃子个数
```

```
            peach(d,s);                                    //递归调用
        }
    }
}
```

3.6 实验5 数组排序

【实验目的】

(1) 理解和掌握数组的声明、创建、初始化和访问等。

(2) 熟练掌握和运用数组来解决实际问题。

【实验内容】

通过键盘任意输入6个数字,对这些数字使用冒泡法进行排序,排序后将6个数字由小到大的顺序排列显示出来。

冒泡排序的基本概念是:依次比较相邻的两个数,将小数放在前面,大数放在后面。即在第一趟:首先比较第一个和第二个数,将小数放前,大数放后。然后比较第二个数和第三个数,将小数放前,大数放后,依此类推,直至比较最后两个数,将小数放前,大数放后。至此第一趟结束,将最大的数放到了最后。在第二趟:仍从第一对数开始比较(因为可能由于第二个数和第三个数的交换,使得第一个数不再小于第二个数),将小数放前,大数放后,一直比较到倒数第二个数(倒数第一的位置上已经是最大的),第二趟结束,在倒数第二的位置上得到一个新的最大数(其实在整个数列中是第二大的数)。如此下去,重复以上过程,直至最终完成排序。由于在排序过程中总是小数往前放,大数往后放,相当于气泡往上升,所以称作冒泡排序。

【参考代码】

```
import java.util.Scanner;

public class BubbleSort {

    private int[ ] numbers = new int[6];

    /*
     * 输入数组元素
     */
    public void input() {
        Scanner scanner = new Scanner(System.in);
        for(int i = 0; i < numbers.length; i++) {
            numbers[i] = scanner.nextInt();
        }
    }

    /*
     * 输出数组元素
     */
    public void print() {
        for(int i = 0; i < numbers.length; i++) {
```

```
            System.out.print(numbers[i] + "\t");
        }
        System.out.println();
    }

    /*
     * 排序数组元素
     */
    public void sort() {
        int temp;
        for (int i = 1; i <= numbers.length - 1; i++) {      //比较 n-1 趟
            for (int j = 1; j <= numbers.length - i; j++) { //每趟比较 n-i 次
                if (numbers[j - 1] > numbers[j]) {
                    temp = numbers[j - 1];
                    numbers[j - 1] = numbers[j];
                    numbers[j] = temp;
                }
            }
        }
    }
    public static void main(String args[ ]){
        BubbleSort bubbleSort = new BubbleSort();
        bubbleSort.input();
        System.out.println("按照输入顺序显示");
        bubbleSort.print();
        bubbleSort.sort();
        System.out.println("按照排序后顺序显示");
        bubbleSort.print();
    }
}
```

程序的运行结果如图 3.6 所示。

```
5
1
7
3
9
4
按照输入顺序显示
5   1   7   3   9   4
按照排序后顺序显示
1   3   4   5   7   9
```

图 3.6　数组排序

【程序解析】

这里使用 java.util 包中的 Scanner 类实现接收键盘输入的数据。按照冒泡排序思想，

程序排序顺序如下所示：

```
原序列:          5   1   7   3   9   4
第 1 趟:
   第 1 次比较:  1   5   7   3   9   4
   第 2 次比较:  1   5   7   3   9   4
   第 3 次比较:  1   5   3   7   9   4
   第 4 次比较:  1   5   3   7   9   4
   第 5 次比较:  1   5   3   7   4   9
第 2 趟:
   第 1 次比较:  1   5   3   7   4   9
   第 2 次比较:  1   3   5   7   4   9
   第 3 次比较:  1   3   5   7   4   9
   第 4 次比较:  1   3   5   4   7   9
第 3 趟:
   第 1 次比较:  1   3   5   4   7   9
   第 2 次比较:  1   3   5   4   7   9
   第 3 次比较:  1   3   4   5   7   9
第 4 趟:
   第 1 次比较:  1   3   4   5   7   9
   第 2 次比较:  1   3   4   5   7   9
第 5 趟:
   第 1 次比较:  1   3   4   5   7   9
```

3.7 课 外 练 习

1. 编写一个程序，实现命令行输入一个月份，可以判断出是什么季节，如输入 May，判断的结果是“May is the summer!”。

2. 编写一个程序，实现判断 2009 年是否闰年，输出判断结果。

3. 编写一个程序，求出 1!+2!+…+10! 的值。

4. 编写一个猜数字游戏，程序运行时会自动生成一个幸运数字，用户从控制台输入猜测的数字。如果输入的数字大于幸运数字，给出“您输入的数字太大”提示，如果输入的数字小于幸运数字，给出“您输入的数字太小”提示，用户输入了正确的幸运数字，给出“恭喜您!”提示，用户输入“quit”退出游戏。

提示：该程序循环的次数预先不知，此时可以使用 while 循环语句来实现。循环语句中嵌套了 if 判断语句，当 if 判断满足条件时，使用流程控制的辅助语句“break;”来跳出循环，否则一直循环下去。Math.random()可以产生一个从 0.0～1.0 的 double 类型的数据，Math.random() * 20 则会产生一个从 0.0～20.0 的数据。