

Chp3 流程控制

本章导读

本章主要介绍 Java 中一些基本的流程控制语句。

在介绍流程控制语句之前,我们首先介绍一个预备知识:如何从命令行上读入一个数据。

0 读入数据

在本章的开始,简单为大家介绍一下 Java 中如何读入数据。Java1.5 中,有一个非常简单的用来读入数据的类: `java.util.Scanner`。

使用时的代码如下:

```
//引入 Scanner 类
import java.util.Scanner;

public class TestScanner{

    public static void main(String args[]){
        //下面这行代码创建了一个 Scanner 对象
        //可以理解为,这行代码为读入数据做准备
        Scanner sc = new Scanner(System.in);

        System.out.print("请输入一个字符串: ");
        //读入一行字符串,可以使用 sc.nextLine() 语句
        String str = sc.nextLine();
        System.out.println(str + " 收到了!");

        System.out.print("请输入一个整数: ");
        //读入整数时,使用 sc.nextInt() 语句
        int n = sc.nextInt();

        System.out.print("请输入一个小数: ");
        //读入浮点数,可以使用 sc.nextDouble() 语句
        double d = sc.nextDouble();

        System.out.println(n * d);

    }
}
```

运行时,可以根据提示,在控制台上进行输入。结果如下:

```

D:\Book\chp3>javac TestScanner.java

D:\Book\chp3>java TestScanner
请输入一个字符串: hello world
hello world 收到了!
请输入一个整数: 10
请输入一个小数: 2.6
26.0

D:\Book\chp3>_

```

根据提示，可以在命令行上输入字符串、整数和小数。通过 `java.util.Scanner` 类，我们就实现了在命令行上读入数据的功能。

1 分支结构

1.1 if 语句

我们之前写的所有代码，都是从头到尾顺序执行，也就是说，每句代码都会按照顺序被执行一遍。但是，光有顺序执行的功能是不行的，请看下面的代码：

```

import java.util.Scanner;
public class TestDivide{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int a = 10;
        int b = sc.nextInt();
        System.out.println(a/b);
    }
}

```

这段代码读取一个整数 `b` 的值，然后计算 `a/b`。在大部分情况下，代码执行都没有问题。例如，我们输入 2，结果如下：

```

D:\Book\chp3>javac TestDivide.java

D:\Book\chp3>java TestDivide
2
5

D:\Book\chp3>

```

然而当 `b` 的值为 0 时，由于除法的除数不能为 0，会产生一个错误。如下图所示：

```

D:\Book\chp3>javac TestDivide.java

D:\Book\chp3>java TestDivide
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestDivide.main<TestDivide.java:7>

D:\Book\chp3>

```

因此，仅仅顺序执行代码，已经无法满足要求了。我们需要对 `b` 的值进行判断：如果输

入的 `b` 的值不为 0，则可以执行 `a/b`；如果 `b` 的值为 0，则应当给出用户更加明确的提示，而不应该出现“Exception”之类的不太友好的错误信息。

1.1.1 if 语句的基本语法

为了能够完成上述的功能，我们引入了 `if` 语句。`if` 语句是最基本的分支结构之一，可以用来控制程序的执行。具体的说，`if` 语句表示能够对某些条件进行判断，根据是否满足特定的条件，让程序执行不同的代码。

最基础的 `if` 语句语法结构为：

```
if (布尔表达式) {  
    代码块 1  
}  
else {  
    代码块 2  
}
```

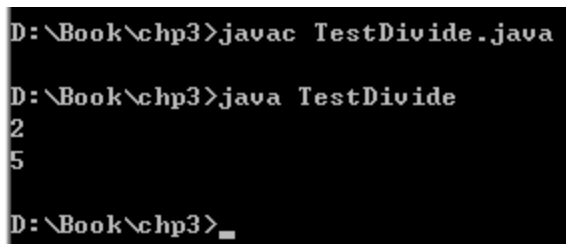
`if` 关键字后面跟一对圆括号，圆括号中是一个布尔表达式。所谓的布尔表达式，指的是值为 `boolean` 类型的表达式。例如，`a == 5`、`b >= 3`，以及 `(a > 4) && (b < 5)` 等，都是布尔表达式。

`if` 语句会对布尔表达式的值进行判断。当布尔表达式值为 `true`，执行代码块 1；反之当布尔表达式为 `false` 时，执行代码块 2。

有了 `if` 语句，我们就可以对前面的 `TestDivide.java` 文件进行改写。修改之后的代码如下：

```
import java.util.Scanner;  
  
public class TestDivide {  
    public static void main(String args[]) {  
        Scanner sc = new Scanner(System.in);  
        int a = 10;  
        int b = sc.nextInt();  
        if (b != 0) {  
            System.out.println(a/b);  
        } else {  
            System.out.println("b 不能为 0");  
        }  
    }  
}
```

上面的代码，对 `b` 是否为 0 进行了判断。假设我们首先输入了 2，此时，`b` 的值不为 0，所以 `b != 0` 这个表达式的值为 `true`。根据 `if` 语句的特点，会执行第一个代码块中的内容，输出 `a/b` 的值。如下图所示：



```
D:\Book\chp3>javac TestDivide.java  
  
D:\Book\chp3>java TestDivide  
2  
5  
  
D:\Book\chp3>_
```

当我们输入 0 时，此时 `b` 的值为 0，`b != 0` 这个表达式的值为 `false`。因此，会执行 `if` 语句第二个代码块中的内容，输出“b 不能为 0”。如下图所示：

```
D:\Book\chp3>java TestDivide
0
b不能为0
D:\Book\chp3>_
```

这样，使用 if 语句，我们就能够对用户输入的值进行判断。根据判断结果的不同，执行不同的代码。

需要注意的是，在代码块 1 和代码块 2 中，都可能包含不止一个 Java 语句。例如，下面的代码：

```
import java.util.Scanner;
public class TestBiggerThanTen{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        if (n > 10){
            System.out.println("statement 1");
            System.out.println("statement 2");
        }else{
            System.out.println("statement 3");
            System.out.println("statement 4");
        }
    }
}
```

上面这段代码读入一个整数，如果这个整数大于 10，则输出 statement 1 和 statement 2，如下图所示：

```
D:\Book\chp3>javac TestBiggerThanTen.java
D:\Book\chp3>java TestBiggerThanTen
12
statement 1
statement 2
D:\Book\chp3>_
```

当输入的整数小于或等于 10 时，则输出 statement3 和 statement4。如下图所示：

```
D:\Book\chp3>java TestBiggerThanTen
8
statement 3
statement 4
D:\Book\chp3>
```

上面我们介绍了 if 语句的基本形式。而除了基本形式之外，if 语句还有两种变化的形式。

第一，else 语句可以省略。当我们只需要当条件为真时执行某些操作，而条件为假时不需要任何操作时，就可以省略 else。例如，我们读入一个整数，然后取这个整数的绝对值。如果这个整数是正数或 0，则不需要对这个整数做任何的操作。只有在这个整数小于 0 时，才需要对其进行处理。

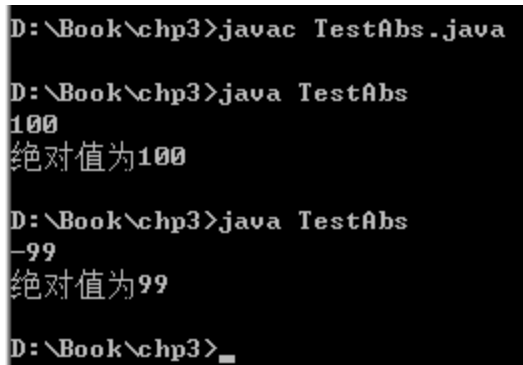
代码如下：

```
import java.util.Scanner;

public class TestAbs{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        if ( n < 0 ){
            n = -n;
        }
        System.out.println("绝对值为" + n);
    }
}
```

上面的代码，当 n 为负数时， n 的绝对值为 $-n$ 。而当 n 为正数和 0 时，由于绝对值就是 n 本身，因此不需要执行任何代码。所以，上面的程序中，`else` 部分可以省略。

运行结果如下：



```
D:\Book\chp3>javac TestAbs.java
D:\Book\chp3>java TestAbs
100
绝对值为100
D:\Book\chp3>java TestAbs
-99
绝对值为99
D:\Book\chp3>_
```

除了可以省略 `else` 部分之外，`if` 语句还有另外一种变化。这就是，当 `if` 或者 `else` 后面的代码块中只有一个语句时，花括号可以省略。例如我们第一个程序：

```
if (b != 0){
    System.out.println(a/b);
}else{
    System.out.println("b 不能为 0");
}
```

这个代码中，每个代码块中都只有一个语句，因此，花括号可以省略，写成下面的形式：

```
if (b != 0)
    System.out.println(a/b);
else
    System.out.println("b 不能为 0");
```

这两种形式的效果是一样的。但是，对于初学者来说，强烈建议大家无论在什么情况下，都保留花括号，这样能够避免初学者犯很多低级错误。

1.1.2 流程图

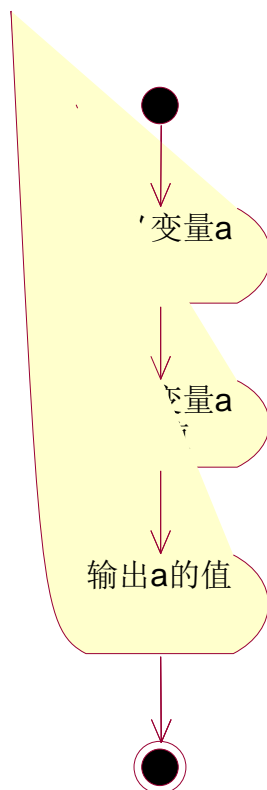
程序执行的流程有很多种。例如，有些代码是依次顺序执行的，有些代码要满足特定的

条件才能执行，有些代码会循环的执行。为了能够直观而清晰的描述程序执行的流程，我们可以绘制流程图。

流程图可以用来表示程序执行的流程。首先，最基本的流程：顺序执行的流程。例如下面的代码：

```
int a;  
a = sc.nextInt();  
System.out.println(a);
```

这段代码顺序执行了这些操作：定义变量、读取变量的值、输出变量。可以用下面的流程图来表示这段代码的执行：

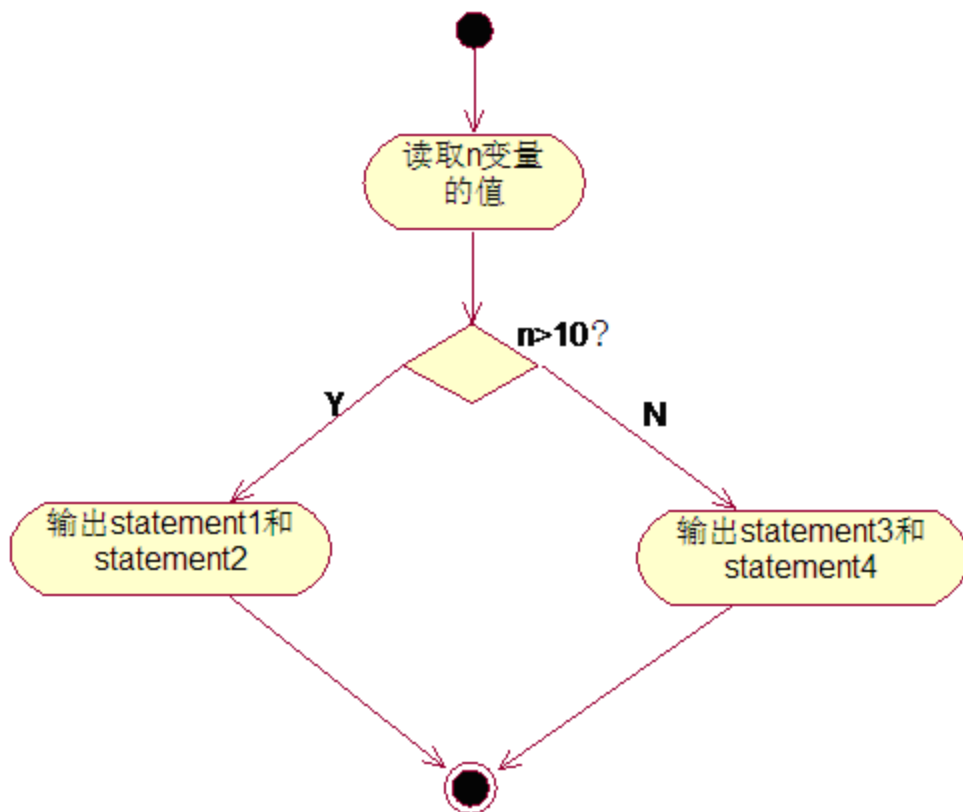


在上面的图形中，程序的开始用实心的黑色圆点表示，而程序的终止用黑色圆点加一个圆环来表示。另外，每一个框表示一个步骤，每个步骤包括一个或多个语句。通过箭头，表示从一个步骤跳到下一个步骤。

对于 if 语句来说，我们可以用一个菱形来表示判断。例如，下面的代码：

```
int n = sc.nextInt();  
if (n > 10) {  
    System.out.println("statement 1");  
    System.out.println("statement 2");  
} else {  
    System.out.println("statement 3");  
    System.out.println("statement 4");  
}
```

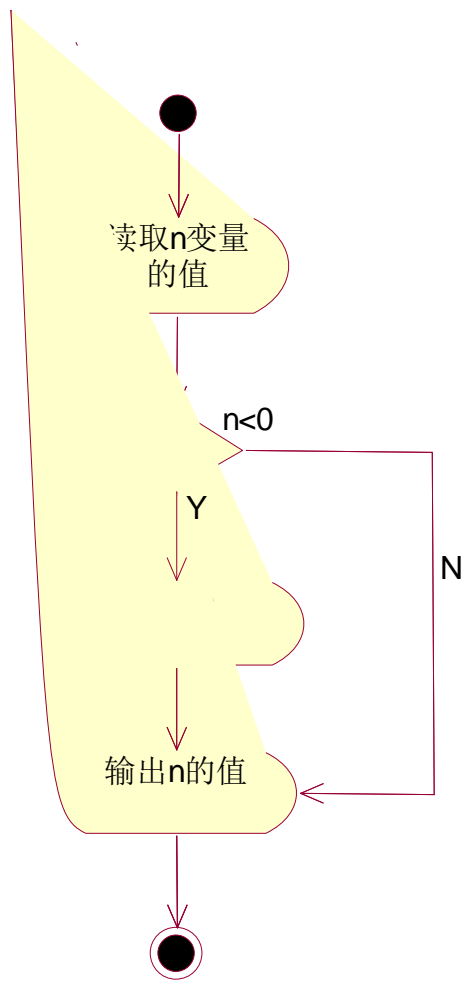
上面的代码，用流程图表示如下：



顺序执行的流程，在 if 语句的位置，被分成了两路：一路为 Y，表示的是当条件为真时的流程；另一路为 N，表示当条件为假时的流程。

if 语句通过对某些条件进行判断，能够把一条顺序执行的流程，分成两条支路。因此，if 语句也叫做分支结构。

而有些 if 语句会省略 else，例如前面提到的求绝对值的程序。用流程图表示如下：



1.1.3 用 if 语句做多重分支

通过上面介绍的内容,我们可以使用 if 语句对某些条件进行判断,根据判断的结果是 true 还是 false, 让程序执行不同的代码。

然而,有些时候,判断的结果并不仅仅是 true 或者 false。例如,我们读入一个 0~100 之间的整数,表示学生的成绩。我们要写一个程序,对成绩进行评级: 0~59 评为 E, 60~69 评为 D, 70~79 评为 C, 80~89 评为 B, 90~100 评为 A。我们要对表示成绩的这个整数区分出 5 种不同的情况。

这种类型的程序,我们需要进行多次判断。先定义一个 int 类型的变量 score, 并从命令行上读入用户的输入, 代码片段如下:

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
```

首先,应当判断这个整数是否是大于等于 0 并且小于 60。如果满足这个条件的话,则学生的成绩是 E 级, 输出 E;

如果不满足这个条件的话,则说明学生的成绩大于 59。而学生成绩如果大于 59 分,则又有很多种不同的情况, 需要继续判断。

代码片段如下:

```
if (score >= 0 && score < 60) { // 如果学生成绩大于等于 0 小于 60, 则输出 E
```

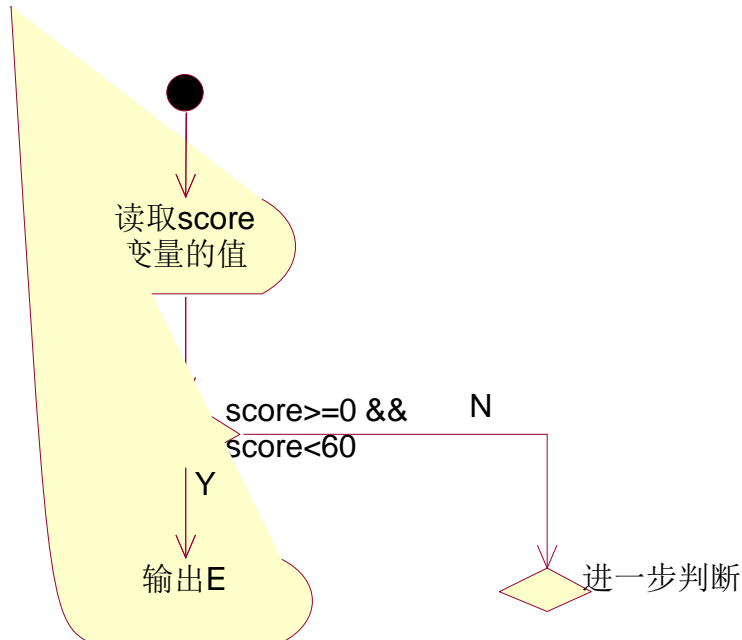


```

        System.out.println("E");
    }else{ //否则，需要进行进一步的判断
        ...
    }

```

流程图如下：



在 `else` 语句中，我们进行下一步的判断。接下来，我们判断 `score` 是否大于等于 60 小于 70。如果是，则可以确定学生成绩的等级是 D 级，可以利用 `System.out.println()` 方法输出 D；如果不是，则说明学生的成绩大于等于 70 分，还存在多种可能性，需进行更进一步的判断。

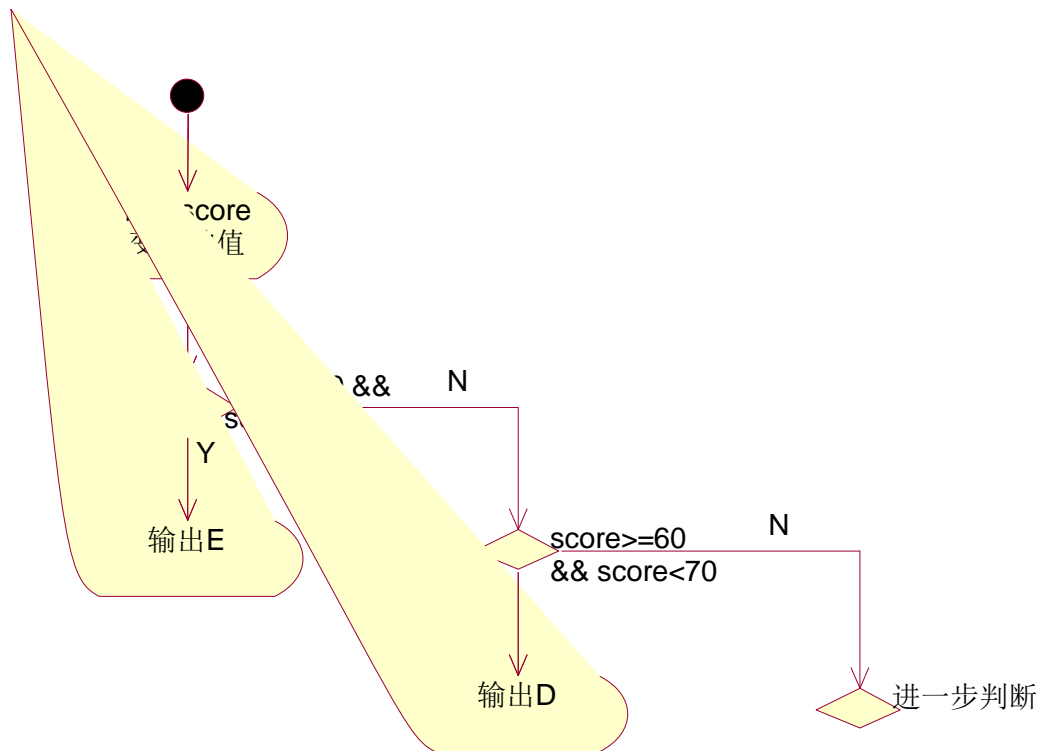
代码片段如下：

```

if (score >= 0 && score < 60) {
    System.out.println("E");
}else{
    if (score >= 60 && score < 70) {
        System.out.println("D");
    }else{
        ...
    }
}

```

流程图如下：



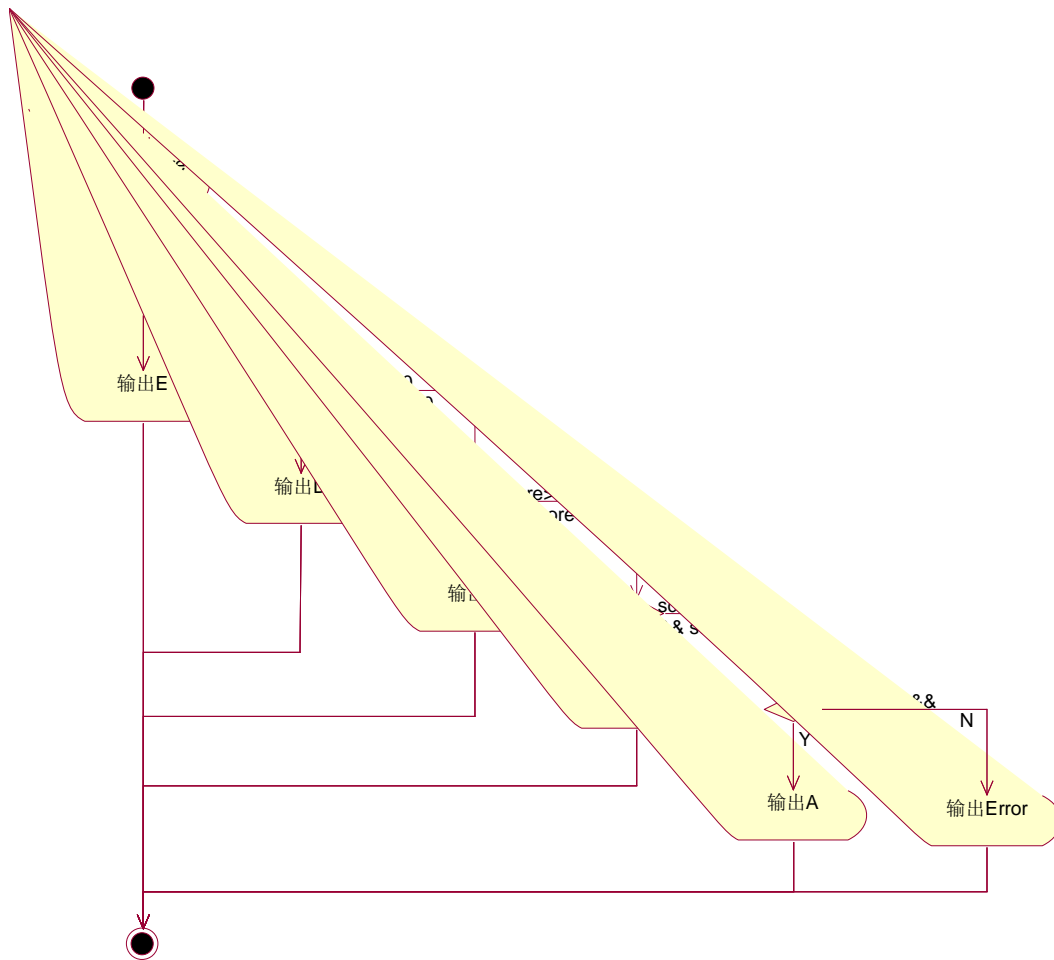
依此类推，完整的代码如下：

```

if (score>0 && score<60){
    System.out.println("E");
}else{
    if (score>=60 && score<70){
        System.out.println("D");
    }else{
        if (score>=70 && score<80){
            System.out.println("C");
        }else{
            if (score>=80 && score<90){
                System.out.println("B");
            }else{
                if (score>=90 && score <=100){
                    System.out.println("A");
                }else{
                    System.out.println("Error");
                }
            }
        }
    }
}
}

```

完整的流程图如下：



注意，最后一个 `else` 语句，只有当前面所有判断都为 `false` 的时候才执行。也就是说，只有当 `score` 小于 0 或者大于 100 时才会执行这个 `else` 语句。

接下来，我们调整一下代码结构。有如下代码片段：

```

if (score >= 0 && score < 60) {
    System.out.println("E");
} else {
    if (score >= 60 && score < 70) {
        System.out.println("D");
    } else {
        ...
    }
}
  
```

在这段代码中，外层的 `else` 语句，其代码块中，只有一个 `if...else` 语句。由于一个完整的 `if...else` 语句只能算一个语句，因此，我们可以把外层 `else` 代码块的花括号省略，并调整缩进，写成下面的形式：

```

if (score >= 0 && score < 60) {
    System.out.println("E");
} else if (score >= 60 && score < 70) {
  
```

```

        System.out.println("D");
    }else{
        if (score>=70 && score<80){
            System.out.println("C");
        }else{
            ...
        }
    }
}

```

现在，字体加粗部分 **else** 语句，其代码块的花括号已经被省略。而在带下划线的 **else** 语句的代码块中，也只有一个 **if...else** 语句。我们可以省略掉这个 **else** 的花括号，并调整缩进：

```

if (score>=0 && score <60){
    System.out.println("E");
}else if (score>=60 && score < 70){
    System.out.println("D");
}else if (score>=70 && score<80){
    System.out.println("C");
}else{
    ...
}

```

依次类推，最终代码的结构会被调整为：

```

if (score>0 && score<60){
    System.out.println("E");
}else if (score>=60 && score<70){
    System.out.println("D");
}else if (score>=70 && score<80){
    System.out.println("C");
}else if (score>=80 && score<90){
    System.out.println("B");
}else if (score>=90 && score <=100){
    System.out.println("A");
}else{
    System.out.println("Error");
}

```

经过调整之后，代码看起来简洁和清晰了很多。这种 **if...else if...else**，是一种常用的结构。这种结构的语法如下：

```

if (条件 1){
    条件 1 的代码块
}else if (条件 2){
    条件 2 的代码块
}else if (条件 3){
    条件 3 的代码块
}

```

```
...
else{
    所有条件都不满足时的代码块
}
```

如果有多个并列的条件，需要根据这些的条件执行不同的代码，这个时候，我们就可以使用 `if ...else if` 这个结构。

这个结构中，由一个 `if` 语句开头，判断第一个条件；中间有多个 `else if` 语句，每个 `else if` 语句中判断一个条件；最后有一个 `else` 语句，当所有条件都不满足时，执行这个 `else` 语句。

1.2 switch 语句

在生活中，我们可能都有过用手机开通业务的经验。在与服务台通话的过程中，可能会听到这样的部分：

“为手机充值请按 1，停开机请按 2，业务办理请按 3，人工服务请按 9，返回上一级菜单请按*……”

根据我们按键的不同，手机服务台会进入不同的流程。

这样，根据输入的值不同，进入多个不同的流程，这种程序结构称为多重分支。在 Java 中，我们可以用 `switch` 语句来完成多重分支。这种语句最基本的语法如下：

```
switch(表达式){
    case value1 :
        语句块 1;
    case value2 :
        语句块 2;
    case value3 :
        语句块 3;
    ...
    case valueN :
        语句块 N;
}
```

`switch` 语句会根据表达式值的不同，对每一个 `case` 所对应的 `value` 进行比较。如果有一个 `value` 与表达式的值相等，则流程跳转到该 `case` 语句的位置。

例如，我们从命令行上读入一个整数，这个整数的范围是 1~5，分别表示成绩的等级“A”~“E”。然后，根据这个整数的不同，输出该等级的成绩范围。例如，如果读入的值是 1，则表示“A”级，输出 90~100；如果读入的值是 3，则表示“C”级，输出 70~79。

利用 `switch` 语句，基本代码如下：

```
import java.util.Scanner;
public class TestSwitch{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        switch(n){
            case 1 :
                System.out.println("恭喜你！成绩不错！");
                System.out.println("90 ~ 100");
```

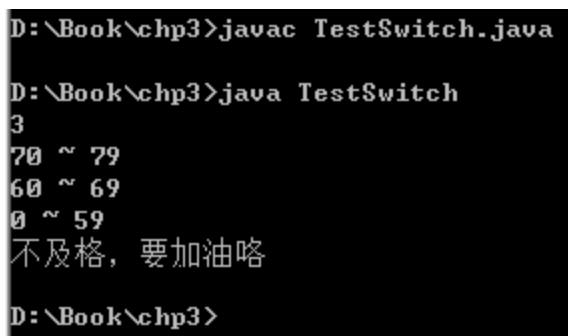
```

        case 2 :
            System.out.println("80 ~ 89");
        case 3 :
            System.out.println("70 ~ 79");
        case 4 :
            System.out.println("60 ~ 69");
        case 5 :
            System.out.println("0 ~ 59");
            System.out.println("不及格，要加油咯");
    }
}
}

```

在上面的代码中，switch 语句会根据 n 的值的不同，跳转到不同的 case 的位置。例如，当 n 的值为 1 时，会跳转到 case 1 的位置，而当 n 的值为 3 时，会跳转到 case 3 的位置。另外，在每一个 case 后面跟的语句块中，都可以包含多个语句。例如，在上面的代码中，在 case 1 和 case 5 的语句块中，包含两个输出语句。

但是，上述的代码并不能完成我们的需求。例如，如果输入 3，输出结果如下：



```

D:\Book\chp3>javac TestSwitch.java
D:\Book\chp3>java TestSwitch
3
70 ~ 79
60 ~ 69
0 ~ 59
不及格，要加油咯
D:\Book\chp3>

```

虽然，n 的值为 3 时，switch 语句会跳转到 case 3 的位置，输出 70 ~ 79，符合我们的设想。但是，在输出完 70 ~ 79 之后，switch 语句并没有结束，而是会从 case 3 的位置，继续往下执行，依次输出 60 ~ 69，0 ~ 59，以及“不及格，要加油咯”。

很显然，这样的执行方式不满足我们的要求。我们希望当执行完一个 case 的语句块之后，就应该跳出 switch 语句，而不是继续执行。为了实现这种要求，我们需要在每一个 case 语句后面，增加一个语句：break。break 语句能够跳出 switch 语句，不让程序继续向下执行。修改之后的 switch 代码如下：

```

switch(n) {
    case 1 :
        System.out.println("恭喜你！成绩不错！");
        System.out.println("90 ~ 100");
        break;
    case 2 :
        System.out.println("80 ~ 89");
        break;
    case 3 :
        System.out.println("70 ~ 79");

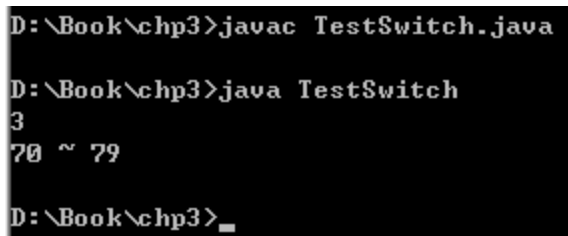
```

```

        break;
    case 4 :
        System.out.println("60 ~ 69");
        break;
    case 5 :
        System.out.println("0 ~ 59");
        System.out.println("不及格，要加油咯");
        break;
}

```

修改过的代码运行效果如下：



```

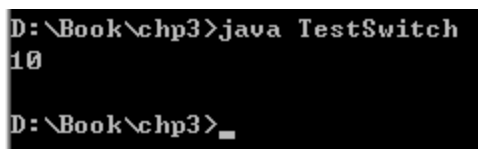
D:\Book\chp3>javac TestSwitch.java
D:\Book\chp3>java TestSwitch
3
70 ~ 79
D:\Book\chp3>_

```

现在，程序读入 3 之后，在 switch 语句中，会匹配到 case 3 的位置。然后，依次往下执行，执行 `System.out.println("70 ~ 79");` 语句，输出 “70~79”。之后，执行 break 语句，跳出 switch 语句块，因此就不会继续输出 60~69，0~59 等内容。

因此，从习惯上说，我们往往会在每一个 case 语句后面，都加上一个 break 语句，用来跳出 switch 代码块。此外，最后一个 case 语句的 break 语句可以省略，因为执行完最后一个 case 之后，后面没有其他的语句，程序会自动的跳出 switch 语句块。但是，为了让每个 case 的格式一致，不建议大家省略最后一个 break 语句。

如果输入的值超出 1~5 的范围，则没有一个 case 能够匹配这种情况。这样会直接跳出 switch 语句，没有任何的输出。例如，如果我们输入 10，则运行时结果如下：



```

D:\Book\chp3>java TestSwitch
10
D:\Book\chp3>_

```

为了在用户输入错误的情况下，给用户一个错误提示，我们需要在所有 case 都匹配不上的时候，输出一个“输入错误”的提示。

在 switch 语句中，还有一个 default 关键词。语法如下：

```

switch(value){
    case value1: XXX;
    case value2: XXX;
    ...
    default : 语句块;
}

```

有了 default 语句之后，在执行 switch 语句时，如果没有一个 case 的 value 值能够匹配上，就会执行 default 的语句块。

我们修改上面的程序，加入 default 语句。修改后的代码如下：

```

switch(n){
    case 1 :

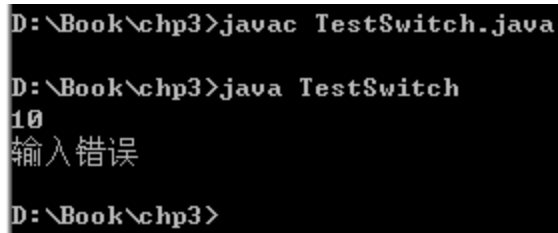
```

```

        System.out.println("恭喜你! 成绩不错! ");
        System.out.println("90 ~ 100");
        break;
    case 2 :
        System.out.println("80 ~ 89");
        break;
    case 3 :
        System.out.println("70 ~ 79");
        break;
    case 4 :
        System.out.println("60 ~ 69");
        break;
    case 5 :
        System.out.println("0 ~ 59");
        System.out.println("不及格, 要加油咯");
        break;
    default :
        System.out.println("输入错误");
        break;
}

```

修改之后的代码执行时结果如下:



```

D:\Book\chp3>javac TestSwitch.java

D:\Book\chp3>java TestSwitch
10
输入错误

D:\Book\chp3>

```

要注意的是, default 语句的执行, 与 default 的位置无关。例如, 我们调整 default 语句的位置, 把代码改成如下形式:

```

switch(n){
    case 1 :
        System.out.println("恭喜你! 成绩不错! ");
        System.out.println("90 ~ 100");
        break;
    case 2 :
        System.out.println("80 ~ 89");
        break;
    default :
        System.out.println("输入错误");
        break;
    case 3 :
        System.out.println("70 ~ 79");
        break;
    case 4 :

```

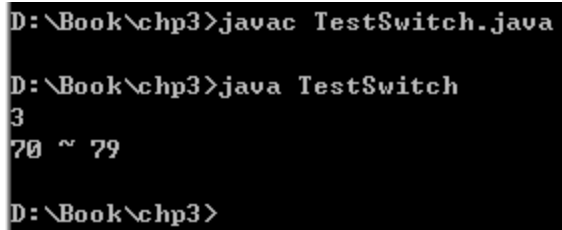


```

        System.out.println("60 ~ 69");
        break;
    case 5 :
        System.out.println("0 ~ 59");
        System.out.println("不及格，要加油咯");
        break;
}

```

我们把 default 语句放到 case 3 的前面，这个时候，如果输入 3，结果会是什么呢？编译运行结果如下：



```

D:\Book\chp3>javac TestSwitch.java

D:\Book\chp3>java TestSwitch
3
70 ~ 79

D:\Book\chp3>

```

可以看到，运行结果依然是输出“70~79”，并没有受到 default 语句的影响。因为 switch 语句在执行时，会首先比对所有的 case 语句，不管这些 case 语句在 default 语句之前还是在其之后。只有所有 case 都匹配不上时，才会执行 default 的代码块。例如上面的例子中，读入 3 之后，首先会比对 case 1、case 2，在代码遇到 default 之后，并不会马上执行，而是先比对 default 语句之后的 case 3。结果，由于读入的值为 3，能够跟 case 3 匹配上，因此输出“70~79”。

但是，从习惯上来说，我们往往会把 default 语句放在 switch 语句的末尾，这样能够提高代码的可读性。一般而言，switch 语句往往会写成如下形式：

```

switch(表达式) {
    case value1 :
        语句块 1;
        break;
    case value2 :
        语句块 2;
        break;
    case value3 :
        语句块 3;
        break;
    ...
    case valueN :
        语句块 N;
        break;
    default :
        default 语句块;
        break;
}

```

但是，switch 语句也有它的局限性。

首先，虽然 switch 语句能够做多重分支，但是不能做范围上的判断。例如，switch 可以

表示读入的值为 1 时如何，为 2 时如何；但是却不能表示读入的值在 50~100 的范围之内应当如何。也就是说，我们用 case 1, case 2 这样的语句来表示匹配 1、2 这两个值，但是 switch 语句不支持如 case 50...100 这种语法，来表示范围。

如果想要对值的不同范围进行判断，则可以使用 if ... else if 的语法。我们之前写的，根据学生成绩判断学生成绩等级的例子，就是一个非常典型的对不同范围的值进行不同的操作。这种逻辑无法使用 switch 语句完成，只能用 if ... else if 这样的语法。

此外，switch 语句只能够判断四种类型的值：byte、short、int 或 char。也就是说，在 switch 语句的圆括号中，表达式的值只能是 byte、short、int 和 char 类型。

2 循环控制

循环是计算机程序当中最重要的一个特性之一，毕竟，当初人们发明计算机，就是想和一些重复而机械的劳动交给计算机来完成，而把一些比较需要创造力的部分交给人来完成。

比如，我们在第一章中学会了怎样输出一个“HelloWorld”。那如果我们需要输出十个“HelloWorld”怎么办？这时候，也许还能够写 10 个输出语句。但是如果我们需要输出更多个 Hello World，比如 100 万个。显然，写 100 万个输出语句是不现实的。计算机更加擅长这种重复而机械的劳动，我们应当把这种反复的输出让计算机来完成。这就需要在编程时，掌握循环语句的使用。

循环语句的意义在于，可以让程序中的某些代码被反复的执行多次。

首先，我们来对循环的概念进行分析。对于任何一个循环，我们都应该从四个方面来思考它：1、初始化；2、循环条件；3、循环体；4、迭代操作。

初始化，指的是在循环开始之前，所需要做的准备工作。比如，生活中，“包饺子”这件事情，就是一个重复的劳动，我们可以把“包饺子”当做是循环。但是，要真正开始包饺子，必须做好充分的准备工作，例如和面，擀皮，和馅……等等。

循环条件，往往是一个布尔表达式，当这个表达式为真时，循环继续执行；而当这个表达式为假时，循环退出。也就是说，循环条件指的是，控制循环是否能够继续的条件。例如，在包饺子这个循环中，循环条件就是：剩余的饺子皮>0 && 剩余的饺子馅>0。

循环体，指的是需要反复执行的重复性的操作。例如，饺子的循环体，就是重复性的包饺子劳动，把饺子馅放在皮中间，把饺子捏成元宝状。完成这些操作之后，一个饺子就包完了，然后开始包下一个饺子。

迭代操作，往往是体现每次执行循环体之间所产生的变化，也可以理解为每次执行循环体时所产生的不同。举例来说，在包饺子的过程中，每包一个饺子，会产生什么变化呢？产生的变化就是，剩余的饺子皮少了一张，剩余的饺子馅少了一块。

迭代操作往往是和循环条件联系在一起的，在迭代操作时，往往会修改循环条件中使用的变量值。例如，在我们的包饺子循环中，迭代操作就操作了剩余的饺子皮和饺子馅的数量，而循环条件正是对这两个量进行判断。

要注意的是，并不是每一种循环都具有循环的四个要素。有些循环里，循环条件和迭代操作是同一个语句；有些循环里循环体和迭代操作是一样的，等等。但是，在写循环的时候，我们应当对循环的四个要素进行通盘考虑，避免遗漏。

介绍完循环的概念之后，下面要介绍的是 Java 中循环的语法。

在 Java 中有三种循环：for 循环、while 循环和 do...while 循环。下面我们依次来给大家阐述。

2.1 for 循环

for 循环包含循环的四个要素，语法如下：

```
for(初始化;循环条件;迭代操作){  
    循环体;  
}
```

举例来说，我们可以写一个 for 循环用来输出 10 个 HelloWorld。我们用一个变量 i，来统计总共循环了多少次，我们把这个 i 变量称为计数器。这样，循环的初始化，就应该把计数器清零。每当进行了一次循环之后，就把计数器+1，因此迭代操作，就是 i++。而循环条件，则是判断循环的次数：如果循环已经进行了 10 次，则结束循环；如果循环没有满 10 次，则循环继续。我们在判断循环的时候，只要把计数器 i 和 10 进行比较，就能判断是否执行循环。

由上面的分析，可以写出 for 循环的代码如下：

```
01: int i ;  
02: for(i = 0; i<10; i++){  
03:     System.out.println(i + " Hello World");  
04: }
```

首先，在 01 行，程序定义了一个变量 i，这个变量用作计数器。

之后，在 02 行开始进入 for 循环。

在 for 循环中，首先执行 i=0 的初始化操作，把计数器清零。要注意的是，初始化操作只执行一遍；

然后，进行循环条件的判断。此时 i 值为 0，i<10 判断为真，循环继续；

之后，执行循环体，输出 0 Hello World；

当循环体一次执行结束，执行迭代操作 i++，此时 i 值为 1，表示循环已经完成了一次；

然后，再进行循环条件的判断，此时 i 为 1，i<10 判断为真，循环继续；

执行循环体，输出 1 Hello World；

执行迭代操作 i++，此时，i 的值为 2；

进行循环条件的判断。由于此时 i 的值为 2，i<10 判断为真，循环继续

.....

依次类推，直到最后一次，当 i=9 时，输出 9 HelloWorld；

执行迭代操作，i++。此时，i 的值为 10，表示已经循环了 10 次。

再进行循环条件的判断。由于此时 i 的值为 10，因此 i<10 判断为假，循环退出。

至此，i 从 0 变化到 10，其中当 i 从 0 变化到 9 时各输出一个 HelloWorld，循环体总共执行了 10 次，而循环条件的判断则进行了 11 次。

完整的代码如下：

```
public class TestFor{  
    public static void main(String args[]){  
        int i ;  
        for(i = 0; i<10; i++){  
            System.out.println(i + " Hello World");  
        }  
    }  
}
```

运行结果如下：

```

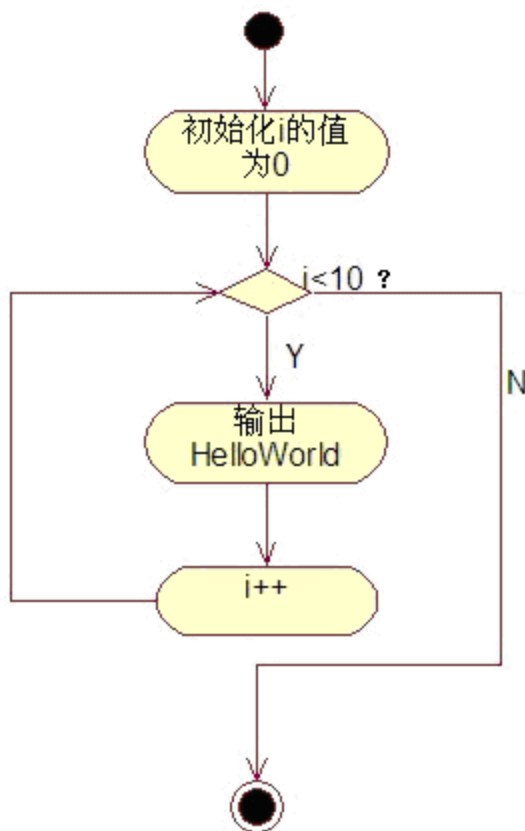
D:\Book\chp3>javac TestFor.java

D:\Book\chp3>java TestFor
0 Hello World
1 Hello World
2 Hello World
3 Hello World
4 Hello World
5 Hello World
6 Hello World
7 Hello World
8 Hello World
9 Hello World

D:\Book\chp3>_

```

for 循环的流程图如下：



在这个程序中，我们定义 i 变量的主要目的，就是用来控制循环的次数。用来控制循环的变量，被称为循环变量。

一般，程序员会把循环变量命名为 i 、 j 、 k ，因此，这三个名字，建议大家除了作为循环变量之外，尽量不要乱用。当一个有经验的程序员看到这三个字母时，自然而然的会想到循环变量，这个时候如果这些变量被拿来做其他的用途，可能会对程序员造成误导。

此外，往往循环变量仅仅是用来控制循环，而一旦脱离循环之后就没有什么价值，因此，我们能够在初始化的时候再定义这个变量。这样一来，这个变量的作用范围就被局限在循环的内部。例如：

```

//注意，i 变量在初始化时被定义
for(int i = 0; i < 10; i++){

```

```

        System.out.println(i + " Hello World");
    }
    // ! System.out.println(i); 编译错误! 找不到符号 i

```

在 for 循环的初始化部分定义变量，也是一种非常常见的写法。

在处理循环变量时，还要注意一点：在写 for 循环时一定要注意循环变量的范围。例如，请快速回答下面的问题：

1、int i=1; i<10; i++ 循环几次？

答案：i 的变化范围 1~9，循环 9 次

2、int i = 0; i<=10 循环几次？

答案：i 的变化范围 0~10，循环 11 次

3、int i = 1; i<=10; i+=2 循环几次？

答案，i 的变化范围 1~9，取值分别为 1、3、5、7、9，循环 5 次。

在写 for 循环的时候，一定要仔细分析循环变量的变化，明确循环总共执行了多少次。

另外，还要注意，循环变量最好不要在循环体中进行赋值。请看下面的这个程序：

//注意，i 变量在初始化被定义

```

for(int i = 0; i<10; i++){
    System.out.println(i + " Hello World");
    i = 4;
}

```

这个程序输出几个 Hello World？

答案是：无数个！

当第一次循环时，输出 0 Hello World，之后 i 被赋值为 4，迭代操作之后 i 的值为 5；当第二次循环时，输出 5 Hello World，之后 i 又被赋值为 4！这样，i 的值永远在 4~5 之间变化，从而永远无法退出循环！

这个错误是典型的死循环。所谓的死循环，指的是永远无法退出的循环。

在上述代码中，产生死循环的原因，是因为在循环体中对 i 进行了赋值，从而使得循环条件永远为真，循环永远无法退出。我们应当尽量避免在 for 循环的循环体中对循环变量进行赋值操作。

虽然应该避免在 for 循环的循环体中对循环变量赋值，但是完全可以在循环体中读取循环变量的值。例如下面这个练习：

求出 $sum = 1 + 2 + 3 + \dots + 50$

对于这个这个练习，我们可以用下面的步骤来进行计算。

首先，让 sum 的值为 0；

其次，让 sum 的值为原有值+1，sum 的结果为 1；

然后，让 sum 的值在原有的基础上+2，sum 的结果为 1+2=3；

再然后，让 sum 的值在原有的基础上+3，sum 的结果为 3+3=6

.....

最后，让 sum 的值在原有的基础上+50。

我们可以看到，在这个计算过程中，每一步都是让 `sum` 的值，在原有的基础上加上一个值。假设我们用一个变量 `i` 来表示这个值，则每一次进行的操作就是：

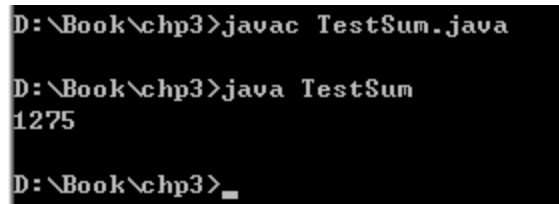
```
sum = sum + i;
```

其中，`i` 变量的变化范围是从 1~50。为此，我们可以设计一个 `for` 循环，在 `for` 循环中，让循环变量 `i` 从 1 循环到 50，表示这 50 个加数。循环体就是 `sum = sum + i`。

示例代码如下：

```
public class TestSum{
    public static void main(String args[]){
        int sum = 0;
        for (int i = 1; i<=50; i++){
            sum = sum + i;
        }
        System.out.println(sum);
    }
}
```

运行结果如下：



```
D:\Book\chp3>javac TestSum.java
D:\Book\chp3>java TestSum
1275
D:\Book\chp3>_
```

特别要注意一下，应当把 `sum` 变量的定义写在 `for` 循环外面。原因也很简单，`for` 循环是用来计算的，我们输出 `sum` 变量计算结果，应当在 `for` 循环结束之后。由于我们要在 `for` 循环之外继续使用 `sum` 变量，因此必须在 `for` 循环之外定义 `sum` 变量。

2.2 while 循环和 do...while 循环

`while` 循环和 `do...while` 循环比较类似，我们先来看这两种循环的语法。

`while` 循环：

```
while(循环条件){
    循环体
}
```

`do...while` 循环：

```
do{
    循环体
}while(循环条件);
```

对于这两种循环而言，初始化部分并不是语法的一部分。但是作为一个良好的编程习惯，强烈建议读者在写循环的时候，在循环的上面加上初始化的代码。

另外。对于这两种循环而言，循环条件的含义是一样的：这是一个布尔表达式，当表达式为 `true` 时，循环继续；当表达式为 `false` 时，循环退出。

最后，对于两种循环而言，循环体的含义也类似，都表示反复执行的那部分操作。要注意的是，在 `while` 循环和 `do...while` 循环中，没有单独的地方写迭代操作。如果需要进行迭代操作的话，应当把迭代操作写在循环体中。

例如，我们分别使用 `while` 循环和 `do...while` 循环来完成输出 10 个 `HelloWorld` 的程序。

//使用 `while` 循环输出 10 个 `Hello World`

```
public class TestWhile{
    public static void main(String args[]){
        int i = 0;
        while(i <10){
            System.out.println(i + " Hello World");
            i++;
        }
    }
}
```

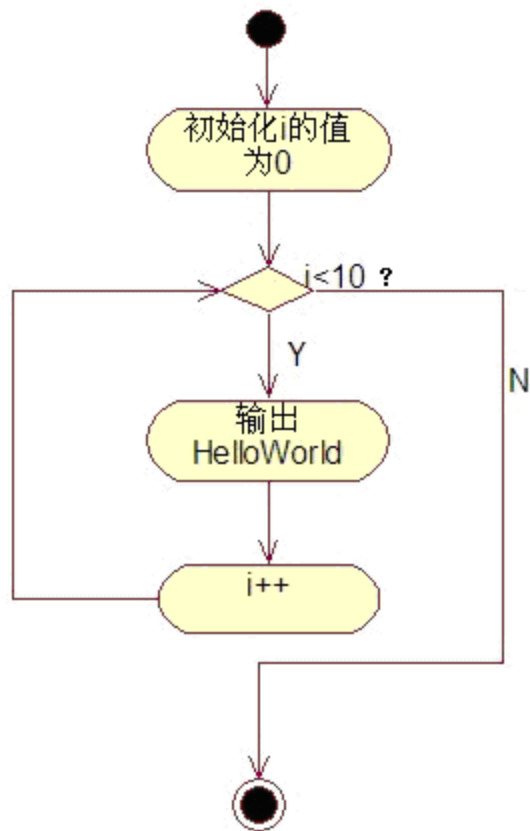
//使用 `do...while` 循环输出 10 个 `Hello World`

```
public class TestDoWhile{
    public static void main(String args[]){
        int i = 0;
        do{
            System.out.println(i + " Hello World");
            i++;
        }while(i <10);
    }
}
```

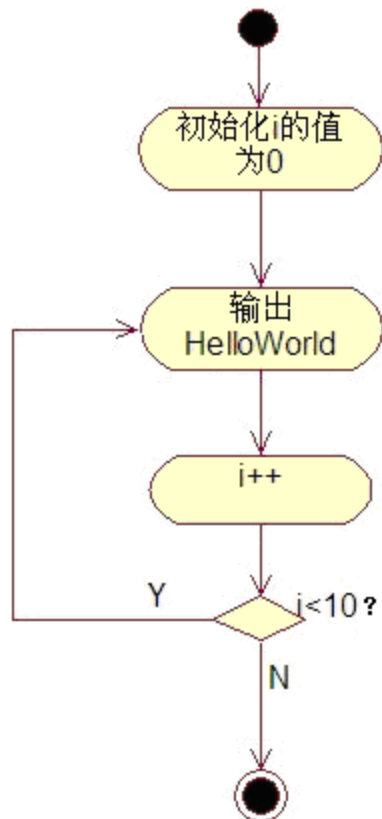
输出结果与 `for` 循环输出的结果类似，在此不再赘述。

这两段代码的执行流程类似，只有一个细小的差别：对于 `while` 循环而言，是先进进行判断，后执行循环体；而对于 `do...while` 循环而言，是先执行循环体，后执行判断。

`while` 循环的流程图如下：



可以看到，我们写的 while 循环程序的流程图，和 for 循环的流程图一样。
do...while 循环的流程图如下：



可以看出，与 while 循环不同，do...while 循环的判断，是在循环体之后执行的。这个

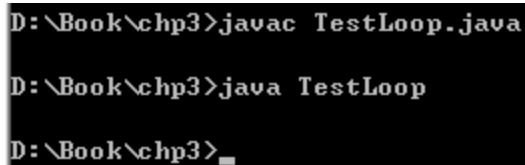
差别对于上面的例子而言，是循环条件判断次数的差别。

while 循环的循环条件，在 *i* 为 0~10 的过程中，共判断了 11 次；而 **do...while** 循环的循环条件，在 *i* 为 0 时没有判断。因为是先执行循环体，后执行判断，因此当 *i* 等于 0 时，会先执行循环体，在循环体中，执行了 *i++*。因此，**do...while** 循环执行第一次判断时，*i* 的值为 1。在整个循环过程中，循环条件的判断，当 *i* 的值为 1~10 的变化过程中，总共判断了 10 次。

由于 **while** 循环因为在第一次执行循环体之前，就要进行判断，因此循环体有可能一次都不执行；而 **do...while** 循环由于要执行一次迭代以后才进行判断，因此循环体至少会执行一次。例如，有如下代码：

```
public class TestLoop{
    public static void main(String args[]){
        int i = 100;
        while (i < 10){
            System.out.println(i + " Hello World");
            i++;
        }
    }
}
```

这段代码，在进入循环体之前，会先进行判断。此时，由于 *i* 的值为 100，循环条件 *i*<10 为假，因此，循环体一次都不执行，程序没有任何输出。运行结果如下：

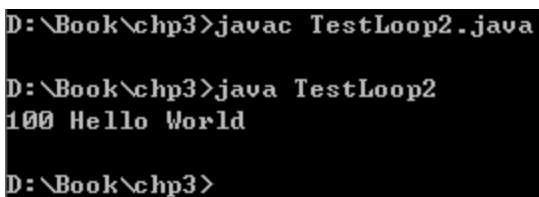


```
D:\Book\chp3>javac TestLoop.java
D:\Book\chp3>java TestLoop
D:\Book\chp3>_
```

而如果把上述代码改成 **do...while** 循环的写法，如下：

```
public class TestLoop2{
    public static void main(String args[]){
        int i = 100;
        do {
            System.out.println(i + " Hello World");
            i++;
        } while (i < 10);
    }
}
```

刚开始 *i*=100 时，没有进行判断，就进入了循环体。在循环体中，输出 100 Hello World，然后执行 *i++*。此时，*i* 的值为 101，循环条件 *i*< 10 结果为假，于是退出循环。运行结果如下：



```
D:\Book\chp3>javac TestLoop2.java
D:\Book\chp3>java TestLoop2
100 Hello World
D:\Book\chp3>
```

因此，我们可以看出，**while** 循环有可能一次都不执行，而与之对应的，**do...while** 循环

至少执行一次。

关于三种循环的基本语法，就介绍完了。我们可以看到，同样是输出 10 个 **Hello World**，用这三种循环都可以完成。既然三种方式都可以完成循环的操作，那在编程时，我们应当选择哪一种循环呢？以下是一些提示：

1) 能够确定次数的循环，应当用 **for** 循环。相反，如果循环的次数不能确定，则应当使用 **while** 循环或者 **do...while** 循环。例如，如果要输出 100 个 **Hello World**，这个循环我们能够确定其要循环 100 次，因此应当使用 **for** 循环的方式。而相反，假设我们要读取文件的所有内容，一次读取一个字节。由于我们不知道文件的长度究竟有多少，因此，不知道需要循环多少次。此时，就可以使用 **while** 循环，模拟代码如下：

```
while(文件中还有数据){  
    读取一个字节;  
}
```

2) 如果要对循环变量进行赋值操作，则应当使用 **while** 循环，而避免使用 **for** 循环。之前介绍过，应当尽量避免在 **for** 循环的循环体中对循环变量进行赋值操作。如果确实需要在循环体中修改循环变量的值，则应当使用 **while** 循环。

3) 如果循环体至少需要执行一次，则应当使用 **do...while** 循环。

2.3 break 和 continue

除了基本的循环之外，Java 还提供了循环中的 **break** 和 **continue** 语句。这两个语句能够帮助程序员对循环进行更加灵活的控制。

break 语句表示跳出当前的循环。例如：

```
01: public class TestBreak{  
02:     public static void main(String args[]){  
03:         for(int i = 0; i<=5; i++){  
04:             if (i == 3) break;  
05:             System.out.println("i="+i);  
06:         }  
07:     }  
08: }
```

程序的第 04 行，进行了一个判断，当 *i* 为 3 时执行 **break** 语句。在执行 **for** 循环过程中，*i* 的值为 0~2 时，判断为假，因此会执行循环体后面的输出语句，分别输出

```
i=0  
i=1  
i=2
```

而当 *i* 为 3 时，循环执行 **break** 语句。此时，会跳出 **for** 循环。由于 **for** 循环后面没有其他的代码，因此程序结束。程序运行结果如下：

```
D:\Book\chp3>javac TestBreak.java

D:\Book\chp3>java TestBreak
i=0
i=1
i=2

D:\Book\chp3>
```

`continue` 语句表示跳出本“次”循环。所谓的本次循环，是指的，`continue` 语句会跳到循环体的末尾，然后执行迭代操作，之后，再进行循环条件的判断。也就是说，使用 `continue` 语句不会跳出整个循环，只是跳过这一轮的循环。例如下面的代码：

```
public class TestContinue{
    public static void main(String args[]){
        for(int i = 0; i<=5; i++){
            if (i == 3) continue;
            System.out.println("i="+i);
        }
    }
}
```

前面，程序正常输出 `i=0, i=1, i=2`。当 `i` 为 3 时，执行 `continue` 语句。此时，代码会跳到 `for` 循环循环体的末尾，跳过输出语句。然后，执行迭代操作 `i++`，`i` 的值为 4，程序继续运行。因此，在最后的結果中，除了 `i=3` 被跳过之外，其他的部分都正常输出。

运行结果如下：

```
D:\Book\chp3>javac TestContinue.java

D:\Book\chp3>java TestContinue
i=0
i=1
i=2
i=4
i=5

D:\Book\chp3>
```

2.4 多重循环

考虑下面这个练习：从命令行上读入一个正整数，根据这个正整数，输出下面的图形：

例如，当 `n=3` 时，输出：

```
*
**
***
```

`n=4` 时，输出

```
*
**
***
****
```

这个练习如何完成呢？思路如下：

对于任何一个正整数 n ，都必须循环 n 次，这样才能输出 n 行。因此，必须写一个 `for` 循环，在循环中定义一个变量 i ，让 i 变量从 1 循环到 n ；而在循环中每一轮循环都输出第 i 行。这样，经过 n 次循环，最终会输出 n 行。循环如下：

```
for(int i = 1; i<=n; i++){
    //循环每次迭代输出第 i 行
}
```

下面考虑循环体。我们可以发现规律：对于第 1 行，需要输出 1 个*号，然后换行；第 2 行，需要输出 2 个*号，然后换行……以此类推，第 i 次执行循环体时，需要输出 i 个*号，以及一个换行符。

为了输出 i 个*号，可以考虑写一个循环。这个循环的循环体每次输出一个*号。这样，通过控制循环的次数，就可以控制输出的*号的个数。

由于要输出 i 个*号，因此，需要循环 i 次。我们可以定义一个变量 j ，让 j 从 1 循环到 i ，代码如下：

```
for(int j = 1; j<=i; j++){
    System.out.print("*");
}
```

把两部分结合起来，代码如下：

```
01:for(int i = 1; i<=n; i++){
02:    for(int j = 1; j<=i; j++){
03:        System.out.print("*");
04:    }
05:    System.out.println();
06: }
```

上面的代码，在外层循环的基础上，又嵌套了一个内层循环。这种结构被称为循环的嵌套。如果嵌套只有两层，则被称为二重循环。如果有多层循环之间嵌套，则被称为多重循环。

假设 n 的值为 3，输出一个三行的三角形。在执行上面的代码过程中，首先进入 01 行。此时， i 的值为 1， $i<=n$ 的判断为真，执行循环。由此，进入外层循环的循环体，范围是 2~5 行。

当代码执行到 02 行的时候，进入了内层循环。此时，在内层循环中定义了一个变量 j ，其作用范围是内层循环的内部，因此作用范围是 2~4 行。此时， j 的值为初始化时给出的值 1，而 i 的值也为 1，这样， $j<=i$ 的值为真，执行内层循环的循环体。

执行 03 行，输出一个 “*” 之后，程序运行到 04 行。此时，内层循环的循环体结束，因此，要执行内层循环的迭代操作： $j++$ 。此时， j 的值为 2。

然后，再进行内层循环的循环条件判断。此时 j 的值为 2， i 的值为 1， $j<=i$ 的值为假，因此内层循环退出。

内层循环退出之后，程序从第 4 行继续往下执行，执行到第 5 行并输出一个换行符。需要注意的是，此时已经不在 2~4 行的范围之内，已经在 j 变量的作用范围之外。也可以理解为，这个时候， j 变量不存在了。

然后，程序进入第 6 行，这意味着外层循环的循环体执行完了一遍。这个时候，需要执行外层循环的迭代操作： $i++$ 。此时， i 的值为 2。

接下来，进行外层循环的循环条件判断。此时 $i<=n$ 的值为真，外层循环继续。然后，循环进入第 2 行。

此时，在第 2 行中，再一次进入了内层循环。在进入这个内层循环的时候，又重新定义了一个变量 j 。要注意，这个变量 j 与第一次进入内层循环时定义的变量，不是同一个。再

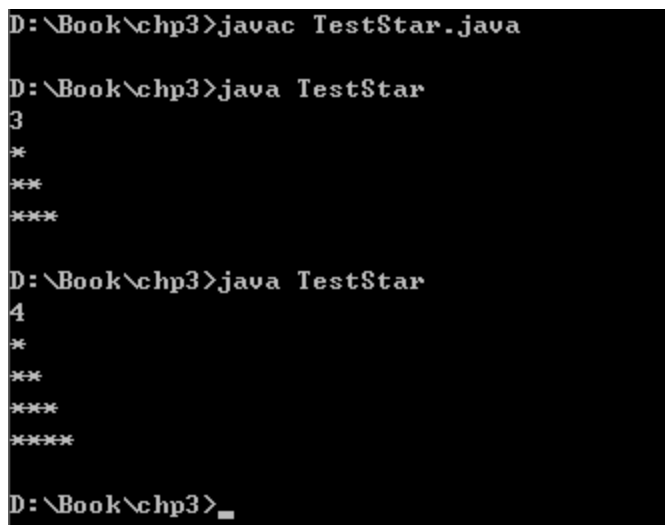
次执行内层循环。此时，由于 *i* 的值是 2，因此内层循环执行两遍，输出两个 “*” 号。

第三次进入内层循环的情况类似，在此不再赘述。

完整代码如下：

```
import java.util.Scanner;
public class TestStar{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        for(int i = 1; i<=n; i++){
            for(int j = 1; j<=i; j++){
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

运行结果如下：



```
D:\Book\chp3>javac TestStar.java
D:\Book\chp3>java TestStar
3
*
**
***
D:\Book\chp3>java TestStar
4
*
**
***
****
D:\Book\chp3>_
```

2.5 多重循环下的 break 和 continue

有了多重循环之后，break 和 continue 语句就显得更加复杂。我们首先来看一个二重循环的代码：

```
public class TestBreakContinue{
    public static void main(String args[]){
        for(int i = 1; i<=3; i++){
            for(int j = 1; j<=4; j++){
                System.out.println("i=" + i + " j= " + j);
            }
        }
    }
}
```

```
}
```

这段代码的输出结果如下：

```
D:\Book\chp3>javac TestBreakContinue.java
D:\Book\chp3>java TestBreakContinue
i=1 j= 1
i=1 j= 2
i=1 j= 3
i=1 j= 4
i=2 j= 1
i=2 j= 2
i=2 j= 3
i=2 j= 4
i=3 j= 1
i=3 j= 2
i=3 j= 3
i=3 j= 4
D:\Book\chp3>_
```

这是一个典型的多重循环。接下来，我们在内层的循环中，增加一个 `break` 语句。修改后的代码片段如下：

```
01: for(int i = 1; i<=3; i++){
02:     for(int j = 1; j<=4; j++){
03:         if (j == 3) break;
04:         System.out.println("i=" + i + " j= " + j);
05:     }
06: }
```

我们来分析一下程序执行的过程。首先，进入 01 行之后，初始化 `i` 变量，并把其值设置为 1。然后，执行外层循环的循环体，进入 02 行，执行内层循环。此时的 `i` 变量值为 1。当 `j` 的值为 1~2 时，输出：

```
i=1 j=1
```

```
i=1 j=2
```

然后，当 `j` 为 3 时，判断的结果为真，执行 `break` 语句，跳出循环。要注意的是，跳出循环时，跳出的是内层循环，因此，程序跳出内层循环的范围 2~5 行，跳转到第 6 行。

第 6 行是外层循环中，循环体的末尾。因此，此处会执行外层循环的迭代操作：`i++`，然后进行外层循环的条件判断。由于此时 `i` 的值为 2，因此 `i<=3` 的值为真，循环继续。从而，程序再次进入内层循环。

由上面的分析我们得知，默认情况下，`break` 语句只能跳出一层循环。如果 `break` 语句在内层循环中，则只能跳出内层循环，而无法直接跳出外层循环。

程序运行的结果如下：

```

D:\Book\chp3>javac TestBreakContinue.java

D:\Book\chp3>java TestBreakContinue
i=1 j= 1
i=1 j= 2
i=2 j= 1
i=2 j= 2
i=3 j= 1
i=3 j= 2

D:\Book\chp3>_

```

与 break 语句类似，continue 语句在默认情况下，也只能对内层循环执行 continue。例如下面的例子：

```

public class TestBreakContinue{
    public static void main(String args[]){
        for(int i = 1; i<=3; i++){
            for(int j = 1; j<=4; j++){
                if (j==3) continue;
                System.out.println("i=" + i + " j= " + j);
            }
        }
    }
}

```

在这段代码中，内层循环的 continue，只能对内层循环起作用。因此，当 i 为 1 时，程序会输出

i=1 j=1

i=1 j=2

i=1 j=4

运行结果如下：

```

D:\Book\chp3>javac TestBreakContinue.java

D:\Book\chp3>java TestBreakContinue
i=1 j= 1
i=1 j= 2
i=2 j= 1
i=2 j= 2
i=3 j= 1
i=3 j= 2

D:\Book\chp3>

```

那有没有办法，能够让 break 语句一下跳出多层循环呢？能不能有办法让 continue 语句对外层循环起作用呢？

接下来，我们介绍一下带标签的 break 和 continue。首先以 break 语句为例，完成能够跳出外层循环的代码。

首先，为了在 **break** 语句中区分内层和外层循环，我们首先应该给这两层循环分别起个名字。这个名字，就是循环的标签。请注意：标签名也必须符合 Java 标识符语法，不能使用非法的字符，关键字，或以数字开头。

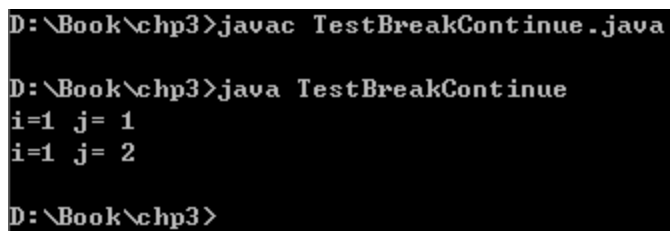
我们可以在循环之前加一个标签，这个标签用来区分循环。代码如下：

```
outer:for(int i = 1; i<=3; i++){
    inner:for(int j = 1; j<=4; j++){
        if (j == 3) break;
        System.out.println("i=" + i + " j= " + j);
    }
}
```

通过上面的操作，我们就把外层循环加上标签 **outer**，把内层循环加上标签 **inner**。然后，在内层循环的 **break** 语句处，为了说明要跳出的是外层循环，可以为 **break** 语句明确指明要跳出的是 **outer** 循环。代码如下：

```
outer:for(int i = 1; i<=3; i++){
    inner:for(int j = 1; j<=4; j++){
        if (j == 3) break outer;
        System.out.println("i=" + i + " j= " + j);
    }
}
```

这样，我们就能够明确指明要跳出 **outer** 循环。执行的结果如下：



```
D:\Book\chp3>javac TestBreakContinue.java
D:\Book\chp3>java TestBreakContinue
i=1 j= 1
i=1 j= 2
D:\Book\chp3>
```

我们可以看到，当 **i** 为 1，**j** 为 3 的时候，执行 **break outer**。这样，就跳出了外层循环。结果，就只输出了两行，程序就结束了。

同样的，**continue** 也有类似的使用方式。我们把上述的代码修改如下：

```
outer:for(int i = 1; i<=3; i++){
    inner:for(int j = 1; j<=4; j++){
        if (j == 3) continue outer;
        System.out.println("i=" + i + " j= " + j);
    }
}
```

这样，执行 **continue** 的时候，会让 **continue** 语句对 **outer** 标签起作用。因此，程序会跳转到外层循环的最末尾，然后执行外层循环的迭代操作。运行结果如下：


```
D:\Book\chp3>javac TestBreakContinue.java

D:\Book\chp3>java TestBreakContinue
i=1 j= 1
i=1 j= 2
i=2 j= 1
i=2 j= 2
i=3 j= 1
i=3 j= 2
D:\Book\chp3>_
```