

第3章 控制语句

语句是程序完成一次操作的基本单元。程序由一条条的语句构成，这些语句依照一定的顺序执行。语句可以顺序执行，可以循环执行也可以跳转执行。

第2章学习了变量，变量的声明就是一条语句，这条语句通知计算机声明了一个变量，计算机得到命令后，执行该语句。本章学习C#中的语句及不同语句的执行方式，包括语句的含义、构成和语句的不同类型等。

本章学习目标：

- 理解语句的含义。
- 掌握基本语句格式。
- 掌握选择语句的结构和用法。
- 掌握循环语句的结构和用法。
- 理解嵌套的用法。
- 掌握跳转语句的用法。
- 掌握异常处理语句的用法。

3.1 语句概述

语句是日常生活中不可缺少的，人们通过语句相互交流，以达到目的。程序中的语句是人与计算机的交互，人们通过语句向计算机发出命令或数据信息，以实现某种功能。

计算机语句与人类语句不同，计算机语句命令性强，一条语句就是一条命令，用来指示计算机运行。语句是程序的构成组件，计算机的所有操作都是根据语句命令来执行的。

目前常用的高级编程语言，如C#和Java，其语句分类和语法格式相差不大，有过其他高级语言编程基础的读者在学习本章内容时，只需要了解各语言间的差别即可。

3.1.1 语句分类

语句是程序的基本组成，语句又分为多种，包括基本语句、空语句、声明语句、选择语句、循环语句和跳转语句等。

程序由一条条的语句构成，默认情况下，这些语句是顺序执行的。但顺序执行的语句使用范围有限，满足不了程序需求，因此C#将语句分为多种。除了顺序执行的语句外，C#中的程序执行语句分为以下几种。

- 选择语句包括：if、else、switch、case。
- 循环（迭代）语句包括：do、for、foreach、in、while。

- 跳转语句包括: break、continue、default、goto、return、yield。
- 异常处理语句包括: throw、try catch、try finally、try catch finally。
- 检查和未检查语句包括: checked、unchecked。
- Fixed 语句包括: fixed。
- Lock 语句包括: lock。

选择语句可根据不同条件选择需要执行的下一条语句。循环语句可重复执行相同语句。跳转语句常与选择语句和循环语句结合使用,用于中断目前执行顺序,并执行指定位置的语句。异常处理语句用于异常的处理,程序运行中常会出现意想不到的错误或漏洞,为了防止这些异常影响系统,可使用异常处理语句来处理。检查和未检查语句用于指定 C#语句的执行的上下文,C# 语句既可以在已检查的上下文中执行,也可以在未检查的上下文中执行。fixed 语句禁止垃圾回收器重定位可移动的变量。lock 关键字将语句块标记为临界区。

除了执行顺序上的分类,C#程序语句在功能上还有其他几种类型:空语句、声明语句、赋值语句和返回值语句等。

3.1.2 基本语句

没有特别说明的语句都按顺序执行,无论语句如何执行,语句结构和语法是固定的。

语句是程序指令,一条语句相当于一条命令,命令语句以分号结尾。命令语句可大可小,长语句可以写在多个代码行上,两行之间不需要连接符,用分号结尾;而一个单纯的分号即可构成一个短语句。

分号是语句不可缺少的结尾;语句与分号之间不能有空格,语句与语句之间用分号隔开,语句之间可以有空格和换行。例如,声明一个整型变量 num,语句如下所示:

```
int num;
```

简单的两个单词、一个空格和一个分号,就构成了一条声明语句。这条语句用来通知计算机准备一个位置给 int 型的变量 num。

最简单的语句是空语句,只有一个分号,不执行任何操作。如下所示:

```
int num;
num=3;
;
```

执行一个空语句就是将控制转到该语句的结束点。这样,如果空语句是可到达的,则空语句的结束点也是可到达的。

3.1.3 语句块

程序中的语句单独为命令,但一个功能常常需要多条语句顺序执行才能实现。C#中允许将多条语句放在一起,作为语句块存在。

语句块是语句的集合，将多条语句写在一个{}内，作为一个整体参与程序执行。

例如，定义一个变量 price 描述单价，定义一个变量 num 描述数量，则描述总价的变量 total 的值为 price 与 num 的乘积。语句如下：

```
{
    int price = 12;
    int num = 10;
    int total;
    total = price * num;
}
```

上述语句中的单条语句都是命令，但都是计算过程的一部分，分开来没有意义，多条语句描述了总价的计算过程，因此可将这些语句作为一个语句块存在。

语句块后不用加分号，常与选择语句关键字或循环语句关键字结合，用于表示参与选择或循环的语句。

3.2 选择语句

选择语句并不是顺序执行的，前面已经提到。如同人们生活中的不同选择，程序中也存在着选择。

如登录系统时的验证，当用户名、密码正确时便可进入系统，但只要密码有误，就不能进入系统。这是一种选择，在不同的状态下系统接下来执行不同的操作。

C#提供了多种选择语句类型，以满足不同的程序需求，如下所示。

- **if 语句** 当满足条件时执行。
- **if else 语句** 当满足条件时执行 if 后的语句，否则执行 else 后的语句。
- **if else if else if 语句** 当满足条件时执行 if 后的语句，否则满足第 2 个条件执行 else if 后的语句，否则满足第 3 个条件执行 else if 后的语句。
- **switch case 语句** 不同条件下执行不同语句。

3.2.1 if 语句

if 语句是选择语句中最简单的一种，表示当指定条件满足时，执行 if 后的语句。执行流程如图 3-1 所示。

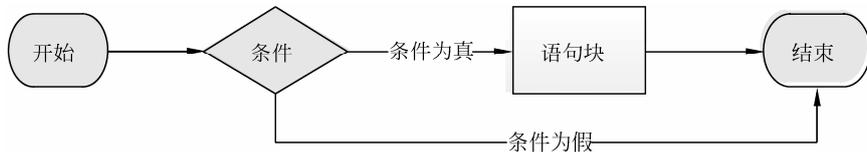


图 3-1 if 语句流程图

if 语句执行时，首先判断条件表达式是否为真：条件为真执行 if 语句下的语句块，结束条件语句；条件为假直接结束条件语句块，执行 if 语句块后面的语句。语法如下：

```
if(条件表达式)
{条件成立时执行的语句}
```

当条件表达式成立时，执行{}内的语句，否则不执行。if 括号内和括号后不使用分号；{}符号内的语句是基本语句，必须以分号结尾；{}符号后不需要使用分号，其用法如练习 3-1 所示。

【练习 3-1】

程序中定义变量 `rpas` 表示用户输入的密码，密码的正确数据为“123456”，若密码正确则输出“登录成功！”字样，使用语句如下：

```
static void Main(string[] args)
{
    Console.WriteLine("请输入密码：");
    string rpas = Console.ReadLine();
    if(rpas=="123456")
    {
        Console.WriteLine("登录成功！");
    }
    Console.WriteLine("登录结束");
}
```

运行上述代码，输入正确的密码“123456”和错误的密码“123123”，当密码正确，输出“登录成功！”字样，如图 3-2 所示；而当密码有误，如图 3-3 所示，程序跳过了 if 语句块的执行，直接执行后面的语句，输出“登录结束”字样。

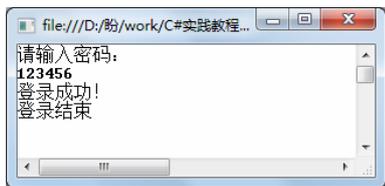


图 3-2 密码验证成功

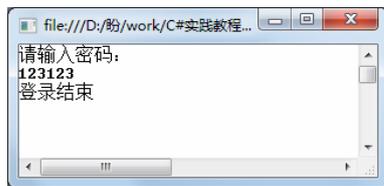


图 3-3 密码无效

3.2.2 if else 语句

if else 语句在 if 语句的基础上，添加了当条件不满足时进行的操作。执行流程如图 3-4 所示。

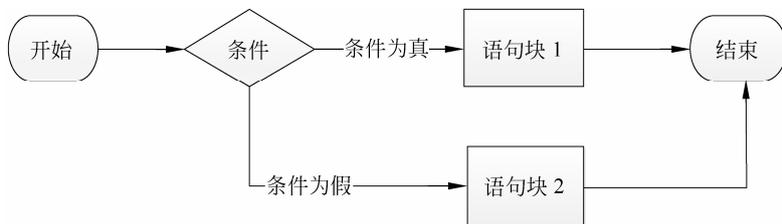


图 3-4 if else 语句流程图

条件的成立只有两种可能，即成立和不成立。if else 语句在条件表达式成立时与 if 语句一样执行 if 后的语句块 1，并结束条件语句；条件表达式不成立时执行 else 后的语句块 2，执行完成后结束条件语句。语法如下：

```
if(条件表达式)
{条件表达式成立时执行的语句}
else
{条件表达式不成立时执行的语句}
```

else 后的 {} 内同样是基本语句，以分号结尾，{} 符号后不需要使用分号，如练习 3-2 所示。

【练习 3-2】

同样以密码的验证为例，在练习 3-1 的基础上添加密码验证失败的提示，即密码有误时，输出“密码有误！”字样，代码如下：

```
static void Main(string[] args)
{
    Console.WriteLine("请输入密码:");
    string rpass = Console.ReadLine();
    if (rpass == "123456")
    {
        Console.WriteLine("登录成功!");
    }
    else
    {
        Console.WriteLine("密码有误!");
    }
    Console.WriteLine("登录结束");
}
```

同样是密码的登录，这次添加密码无效时需要执行的语句，则输入错误的密码时，其效果如图 3-5 所示。程序跳过了 if 语句块的执行，而执行 else 语句块中的语句，输出“密码有误！”字样。

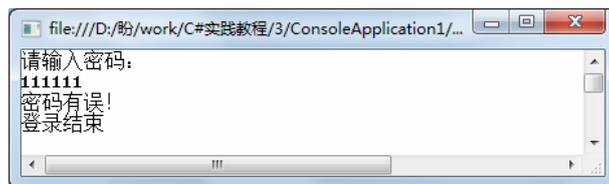


图 3-5 密码有误

3.2.3 if else if 语句

if else if 语句相对复杂，它提供了多个条件来筛选数据，将数据依次分类排除。程序流程如图 3-6 所示。

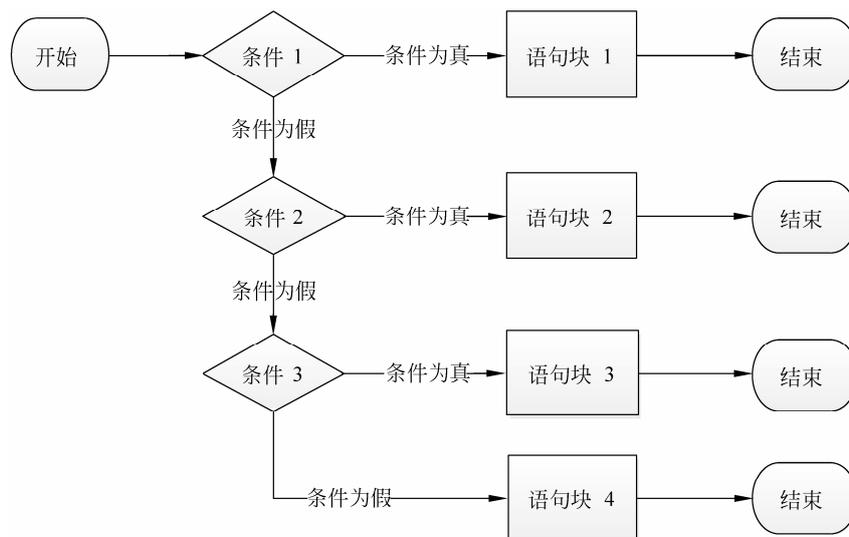


图 3-6 if else if 语句流程图

如图 3-6 所示，if else if 语句在程序进入语句时，首先判定第一个 if 下的条件 1。

- 条件 1 成立，执行语句块 1 并结束条件语句。
- 条件 1 不成立，判断条件 2，条件 2 成立，执行语句块 2 并结束条件语句。
- 条件 2 不成立，判断条件 3，条件 3 成立，执行语句块 3 并结束条件语句。
- 条件 3 不成立，执行语句块 4 并结束条件语句。

图中只有 3 个条件和一个 else 语句。在 if else if 语句中，条件可以是任意多个，但 else 语句小于等于 1 个。即 else 语句可以不要，也可以要，要的话只能有 1 个，因为条件只有成立和不成立两种结果。

if else if 语句基本语法如下：

```

if (条件表达式 1)
{语句块 1}
else if (条件表达式 2)
{语句块 2}
else if (条件表达式 3)
{语句块 3}
...
[else]
{}
  
```

表达式和语句块的语法同 if 语句和 if else 语句一样，有如下的实例。

【练习 3-3】

程序中定义变量 age 表示年龄，定义变量 title 表示称呼，我国有着根据不同年龄对一个人的称呼，如童年、少年、青年、中年、老年，根据年龄判断称呼，语句如下。

```

Console.WriteLine("请输入年龄：");
int age = Convert.ToInt32(Console.ReadLine());
  
```

```
string title="";
if (age < 6)
{
    title = "童年";
}
else if (age < 17)
{
    title = "少年";
}
else if (age < 40)
{
    title = "青年";
}
else if (age < 65)
{
    title = "中年";
}
else
{
    title = "老年";
}
Console.WriteLine("{0}岁的人们称为: {1}",age,title);
```

运行上述代码，分别使用年龄 15、23 和 45 来测试程序的运行结果，其效果分别如图 3-7、图 3-8 和图 3-9 所示，输出了对应的称谓。

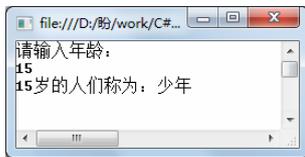


图 3-7 15 岁时的称呼

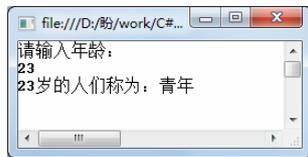


图 3-8 23 岁时的称呼

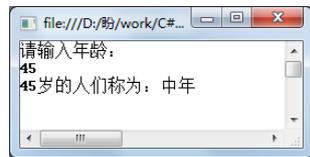


图 3-9 45 岁时的称呼

示例中第二个条件为 `age<17`，虽然年龄小于 17 的还有童年，但童年在第一个条件中已经排除。因此这里使用 `age<17` 与使用 `age>=6 && age<17` 效果是一样的。



提示 还有一些不需要使用最后的 `else` 语句的例子，此时，`if` 和 `else if` 将所有的可能性都包括了。

3.2.4 switch 语句

`switch` 语句的完整形式为 `switch case default`。`switch` 语句与 `if else if` 语句用法相似，但 `switch` 语句中使用的条件只能是确定的值，即条件表达式等于某个常量，不能使用范围。`switch case` 语句流程图如图 3-10 所示。

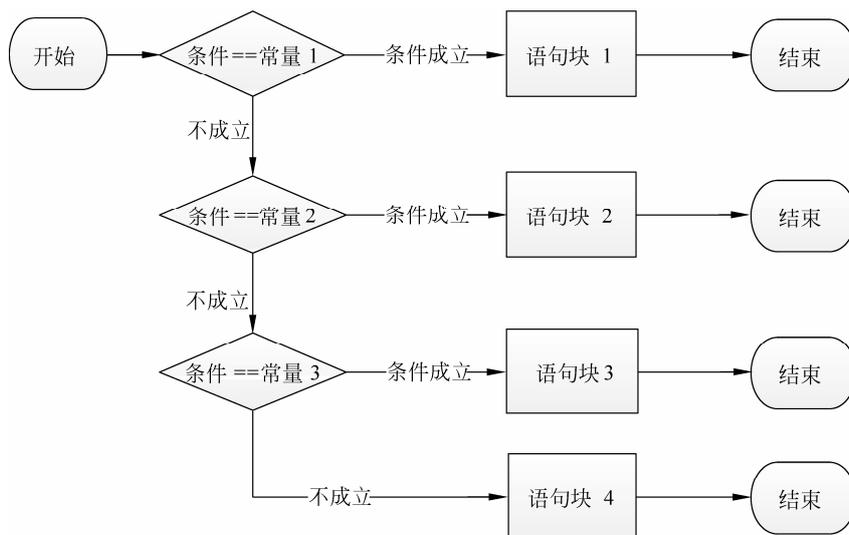


图 3-10 switch 语句流程图

如图 3-10 所示，switch 语句在程序进入语句时，首先判定常量 1 是否与条件相等。常量可以是具体数值，也可以是表达式。

- 条件与常量 1 相等，执行语句块 1 并结束条件语句。
- 条件与常量 2 相等，执行语句块 2 并结束条件语句。
- 条件与常量 3 相等，执行语句块 3 并结束条件语句。
- 条件与三个常量都不相等，执行语句块 4 并结束条件语句。

图中只有 3 个条件表达式和一个 default 语句。default 语句表示剩余的情况，与 else 类似。

与 if else if 语句一样，条件常量可以是任意多个，default 语句可以不要，也可以要，要的话只能有 1 个，因为条件只有成立和不成立两种结果。

switch 语句基本语法如下：

```

switch (条件表达式)
{
    case 常量 1:
        语句块 1
        break;
    case 常量 2:
        语句块 2
        break;
    case 常量 3:
        语句块 3
        break;
    ...
    [default]
}
  
```

switch 语句只使用一个 {} 包含整个模块; break 语句属于跳转语句, 用于跳出当前选择语句块。

switch 语句与 if 语句不同, 当条件符合并执行完当前 case 语句后, 不会默认跳出条件判断, 将会接着执行下一条 case 语句, 使用 break 语句后, 程序将跳出 switch 语句块, 执行后面的语句。

如当表达式等于常量 1, 执行了第一个 case 语句。若不使用 break, 将执行第二个 case 语句而无论表达式是否等于常量 2; 若使用了 break, 接下来将执行 switch {} 后的语句。对 switch 语句的使用, 如练习 3-4 所示。

【练习 3-4】

接收一个从 1 到 7 的任意数字, 输出对应的一周 7 天, 其中, 数字 1 对应星期一, 使用语句如下:

```
Console.WriteLine("请输入日期: ");
int age = Convert.ToInt32(Console.ReadLine());
string title = "";
switch (age)
{
    case 1:
        title = "星期一";
        break;
    case 2:
        title = "星期二";
        break;
    case 3:
        //部分代码省略
        break;
    case 7:
        title = "星期天";
        break;
}
Console.WriteLine("您选择了: {0}", title);
```

运行上述代码, 分别使用数字 2 和数字 7 来验证程序的运行结果, 如图 3-11 和图 3-12 所示。

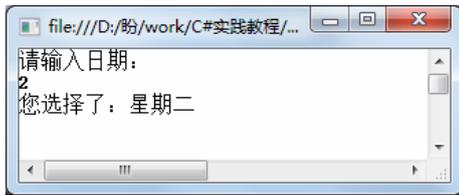


图 3-11 选择 2 的执行结果

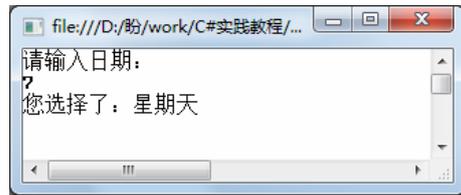


图 3-12 选择 7 的执行结果



case 语句的值是唯一的, 即任何两个 case 语句不能具有相同的值。

3.3 循环语句

循环语句用于重复执行特定语句块，直到循环终止条件成立或遇到跳转语句。程序中经常需要将一些语句重复执行，使用基本语句顺序执行将使开发人员重复工作，影响效率。

如 $1+2+3+\dots+100$ ，使用顺序语句需要将 100 个数相加；若加至 1000、10000 或更大的数，使得数据量加大，容易出错，不便管理。

循环语句简化了这个过程，将指定语句或语句块根据条件重复执行。循环语句分为 4 种：

- **for** for 循环重复执行一个语句或语句块，但在每次重复前验证循环条件是否成立。
- **do while** do while 循环同样重复执行一个语句或语句块，但在每次重复结束时验证循环条件是否成立。
- **while** while 语句指定在特定条件下重复执行一个语句或语句块。
- **foreach in** foreach in 语句为数组或对象集中的每个元素重复一个嵌入语句组。

66

3.3.1 for 语句

for 循环在重复执行的语句块之前加入了循环执行条件，循环条件通常用来限制循环次数，执行流程图如图 3-13 所示。

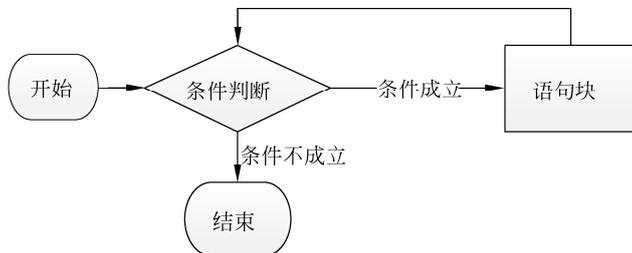


图 3-13 for 循环流程图

如图 3-13 所示，从开始进入判断循环条件是否成立：若成立，执行语句块，并重新判断循环条件是否成立；若不成立，结束这个循环。语法格式如下：

```
for(<初始化>; <条件表达式>; <增量>)
{语句块}
```

for 语句执行括号里面语句的顺序如下：

首先是初始化的语句，如 `int num=0`；若 for 循环之前已经初始化，可以省略初始化表达式，但不能省略分号。

接着是条件表达式，如 `num<5`；表达式决定了该循环将在何时终止。表达式可以省

略，但省略条件表达式，该循环将成为无限死循环。

最后是增量，通常是针对循环中初始化变量的增量，如 `num++`；增量与初始值和表达式共同决定了循环执行的次数。增量可以省略，但省略的增量将导致循环无法达到条件表达式的终值，因此需要在循环的语句块中修改变量值。

增量表达式后不需要分号，因 `for` 语句（）内的 3 个表达式均可以省略，表达式间的分号不能省略，因此有以下空循环语句：

```
for (;;)
{
}
```

循环条件中的变量也可用于实际意义，如以下示例。

【练习 3-5】

定义整型变量 `num`，计算 `1+2+3+4+...+1000` 的数值并赋值给 `num`，输出 `num` 的值。使用 `for` 循环语句如下：

```
int num = 0;
for (int x = 1; x <= 1000; x++)
{
    num = num + x;
}
Console.WriteLine(num);
```

执行结果为：500500。练习 3-5 中将 1 递加至 1000，相当于求等差数列的值，省略 `for` 循环中的初始值和增量，可以使用以下语句。

```
int num = 0;
int x = 1;
for (; x <= 100; )
{
    num = num + x;
    x++;
}
Console.WriteLine(num);
```

以上代码的运行结果与练习 3-5 是一样的。除了数字，条件表达式同样可用于字符，根据字符的 ASCII 值顺序进行，如练习 3-6 所示。

【练习 3-6】

定义字符串变量 `num` 和字符变量 `x`，将字符从 `a` 到 `z` 组合在一起赋值给字符串 `num`，并输出，使用语句如下：

```
string num="";
char x = 'a';
for (; x <= 'z'; )
{
    num = num + x;
    x++;
}
```

```

}
Console.WriteLine(num);

```

执行结果如下所示：

```

abcdefghijklmnopqrstuvwxyz

```



条件表达式必须是布尔值，而且不能是常量表达式，否则循环将会因无法执行或无法结束，而出现漏洞或失去意义。

3.3.2 do while 语句

do while 循环在重复执行的语句块之后加入了循环执行条件，与 for 循环执行顺序相反。除了条件判断顺序的不同，do while 语句虽然同样使用小括号“()”来放置条件表达式，但小括号里面只能有一条语句，不需要分号结尾。执行流程图如图 3-14 所示。

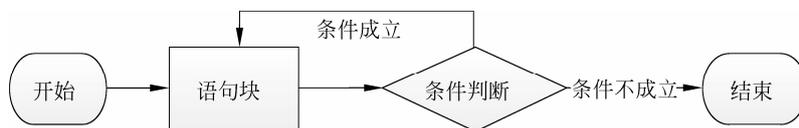


图 3-14 do while 流程图

如图 3-14 所示，程序在开始时首先执行循环中的语句块，在语句块执行结束再进行循环条件的判断。条件成立，重新执行语句块；条件不成立，结束循环。语法结构如下：

```

do
{ 语句块 }
while (条件表达式);

```

与 for 循环不一样的有：

- do 关键字与 while 关键字分别放在循环的开始和结束。
- 条件表达式放在循环最后。
- while 关键字后的括号（）内，表达式只有一个。
- 表达式的括号后需要加分号。

与 for 循环相比，do while 循环将初始化放在了循环之前，将条件变量的变化放在了循环语句块内。同样是 1 到 1000 递加，使用 do while 语句过程如下。

【练习 3-7】

定义整型变量 num，计算 1+2+3+4+……+1000 的数值并赋值给 num，输出 num 的值。使用 do while 语句如下：

```

int num = 0;
int x=1;
do
{

```

```

    num = num + x;
    x = x + 1;
}
while (x <=1000);
Console.WriteLine(num);

```

执行结果与练习 3-5 相同。for 主要控制循环的次数，而对于不确定次数的循环，使用 do while 比较合适，如练习 3-8 所示。

【练习 3-8】

定义整型变量 num，计算 143 除了 1 以外的最小约数，并赋值给 num，输出 num。求约数即相除余数为 0 的整数，只能从最小的数依次计算，则语句如下：

```

int num=2;
do
{
    num=num+1;
}
while (143%num!=0);
Console.WriteLine(num);

```

程序开始，余数都不为 0，直到余数为 0，循环条件不成立，就找到了 143 的最小约数。若使用 for 循环，语句如下：

```

int num = 2;
for (; 143%num!=0; )
{
    num++;
}
Console.WriteLine(num);

```

执行结果为：11。

3.3.3 while 语句

while 语句在条件表达式判定之后执行循环，与 for 循环的执行顺序一样。不同的是语句格式和适用范围。

while 的使用比较灵活，甚至在某些情况下能替代条件语句和跳转语句。while 循环流程图如图 3-15 所示。

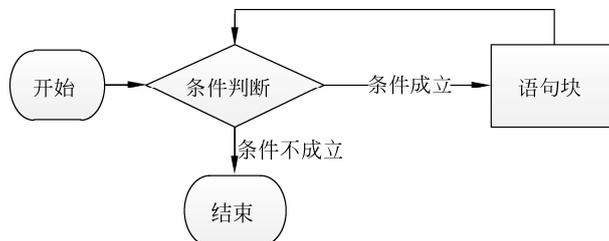


图 3-15 while 语句流程图

如图 3-15 所示，在执行至 `while` 语句时首先判断循环条件是否成立：若成立，执行语句块，并重新判断循环条件是否成立；若不成立，结束这个循环。语法格式如下：

```
while (条件表达式)
{语句块}
```

从 `while` 使用格式看出，`while` 的使用与 `for` 很接近，满足条件表达式即进行 `while` 语句块，否则结束循环。

`while` 后的括号（）只能使用一个条件表达式语句，若在循环中不改变条件表达式中的变量值，循环将无限进行下去，因此循环语句块中包含改变变量值的语句。如使用 `while` 计算 1 到 1000 这 1000 个数字的和，如练习 3-9 所示。

【练习 3-9】

定义整型变量 `num`，计算 $1+2+3+4+\dots+1000$ 的数值并赋值给 `num`，输出 `num` 的值。使用 `while` 循环语句如下：

```
int i=0,num=0;
while(i<=1000)
{
    num = num + i;
    i++;
}
Console.WriteLine(num);
```

运行上述代码，其结果与练习 3-5 结果一样。代码中，`i` 变量在循环中改变了数值，否则循环将永远进行下去，`Console.WriteLine(num);` 语句将无法被执行。

3.3.4 foreach in 语句

`foreach in` 语句主要用于数组、集合等元素成员的访问。单个变量的赋值是最为简单的，之前介绍过。但 C# 中有数组、集合之类数据类型，这些数据类型变量中，不止包含一个数据，而是一系列数据。对这些数据类型的访问，可以使用 `foreach in` 语句。

使用 `foreach in` 语句，可以在不确定元素成员个数的前提下，对元素成员执行遍历，如数组（在本书第 2 章有简单介绍）类型的元素。

数组成员的访问需要指定成员的索引，而若要遍历数组中所有成员，则需要知道数组成员的个数（也叫作数组长度），但使用 `foreach in` 语句可以在没有获取数组长度的情况下对元素成员进行遍历。

数组中的数可多可少，若一个个赋值会加大开发人员工作量，使用 `foreach in` 语句可以对数组及对象集合成员进行操作，如赋值、读取等。

数组中的成员数量各不相同，`foreach in` 循环不需要指定循环次数或条件。如针对有 N 个成员的数组，`foreach in` 循环流程图如图 3-16 所示。

图 3-16 是 `foreach in` 循环作用的形象图，并非标准流程图。`foreach` 语句为数组或对象集合中的每个元素重复一个嵌入语句组，循环访问集合以获取所需信息。



图 3-16 foreach in 循环流程图

嵌入语句为数组或集合中的每个元素顺序执行。当为集合中的所有元素完成操作后，控制传递给 foreach 语句块之后的下一个语句。语法格式如下：

```
foreach (变量声明 in 数组名或集合类名)
{
    语句块 // 使用声明的变量替代数组和集合类成员完成操作
}
```

foreach 语句声明的变量替代了数组成员，由于格式过于模糊、不易理解，通过练习来讲解 foreach 语句的使用，如练习 3-10 所示。

【练习 3-10】

定义整型数组 num 并赋值，定义整型变量替代 num 成员，输出数组成员，使用语句如下。

```
int[] num = new int[] { 0, 1, 2, 3, 5, 8, 13 };
foreach (int i in num)
{
    Console.WriteLine(i);
    Console.WriteLine(" ");
}
```

执行结果为：

```
0 1 2 3 5 8 13
```

代码中的 Console.WriteLine();表示输出()中的字符串，不进行换行。Console.WriteLine(" ");语句输出一个空格。

在 foreach 语句块内的任意位置都可以使用跳转语句跳出当前循环或整个 foreach 语句块。跳转语句在本章 3.5 节介绍。



foreach 循环不能应用于更改集合内容，以避免产生不可预知的副作用。

3.4 嵌套语句

嵌套语句用于在选择或循环语句块中加入选择或循环语句，将内部加入的选择或循环语句作为一个整体，有以下几种形式：

- 选择语句嵌套 在选择语句块中使用选择语句。

- 循环语句嵌套 在循环语句块中使用循环语句。
- 多重混合语句嵌套 在选择或循环语句块中使用多个选择或循环语句。

3.4.1 选择语句嵌套

选择语句以 if else 语句为例，在 if 或 else 后的语句块中，使用另一个条件语句，如获取一个年份是否是闰年，如练习 3-11 所示。

【练习 3-11】

定义整型变量 year 为年份，接收用户对年份的输入数据。根据 year 的值，判断该年份是否是闰年。闰年的判断条件有两个，如下所示。

- 年份是整百数的，先除去 100 后，能被 4 整除的为闰年。
- 年份不是整百数的，能被 4 整除的为闰年。

使用语句如下：

```
Console.WriteLine("请输入年份：");
int year = Convert.ToInt32(Console.ReadLine());
if (year % 100 == 0)
{
    year = year / 100;
    if (year % 4 == 0)
    { Console.WriteLine("年份 {0} 是 闰年", year); }
    else
    { Console.WriteLine("年份 {0} 不是 闰年", year); }
}
else
{
    if (year % 4 == 0)
    { Console.WriteLine("年份 {0} 是 闰年", year); }
    else
    { Console.WriteLine("年份 {0} 不是 闰年", year); }
}
```

练习 3-11 中，首先判断年份是否能被 100 整除，能的话将年份除以 100 再与 4 取余数；若年份不能被 100 整除，直接将年份与 4 取余数，并根据余数判断年份是否是闰年。

分别使用 2012 和 2013 来验证程序，其运行结果如图 3-17 和图 3-18 所示。



图 3-17 验证 2012 是否是闰年



图 3-18 验证 2013 是否是闰年

3.4.2 循环语句嵌套

在循环语句块使用循环语句是常用的，以 for 循环为例，若想输出一行数据或者一列数据，直接使用 for 循环即可，但若想输出几行几列的数据，只能在循环内部使用循环。

例如，2013 年 4 月 1 日为周一，按一行一周输出 4 月份日期，则每一行是一个循环，在一行结束后换行，进行下一个循环，如练习 3-12 所示。

【练习 3-12】

定义整型变量 day 表示日期，输出 4 月份日期。

使用嵌套语句如下：

```
int day;
for (day = 1; day < 31; )
{
    for (int i = 0; (i < 7)&&(day<31); i++)
    {
        Console.Write(day);
        Console.Write(" ");
        day++;
    }
    Console.WriteLine("\n");
}
```

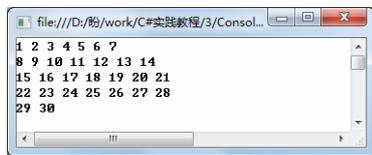


图 3-19 4 月份日期

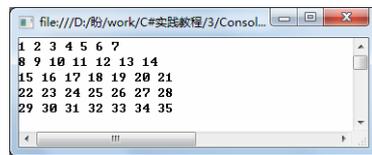


图 3-20 无意义日期

运行结果如图 3-19 所示。练习 3-12 中，由于 day 等于 30 时还会进行内部循环，在内部循环中 day 将大于 30 并进行下去，因此在内部循环中需要添加条件 (day<31)。否则执行结果如图 3-20 所示。

3.4.3 混合语句嵌套

嵌套不仅可以用于选择语句之间或循环语句之间，选择与循环之间的嵌套同样常用。复杂的功能尝试用多重的嵌套，一个循环内出现多个循环和选择语句。

当程序使用多重嵌套时，执行时将循环和选择语句块由内到外作为整体进行。如图 3-19 所示，由于日期数字有一位，有两位的，因此数据看起来比较乱。若在一位数字的前面加一个空格，即可使日期整齐、便于查看，如练习 3-13 所示。

【练习 3-13】

借助练习 3-12 的示例，在输出日期时判断该日期是否小于 10，若小于 10 则为一位

数字，需要在数字前输出空格，否则不输出空格，代码如下：

```
int day;
for (day = 1; day < 31; )
{
    for (int i = 0; (i < 7) && (day < 31); i++)
    {
        if (day < 10)
        {
            Console.Write(" {0}", day);
        }
        else
        {
            Console.Write(day);
        }
        Console.Write(" ");
        day++;
    }
    Console.WriteLine("\n");
}
```

练习 3-13 是多重嵌套的例子，在 for 循环内有 for 循环，内部的 for 循环内又有选择语句。分析这段代码时，将最里面的 if 语句块看作一个整体，接着将内部的 for 循环作为一个整体，执行外部循环。运行练习 3-13，其效果如图 3-21 所示。

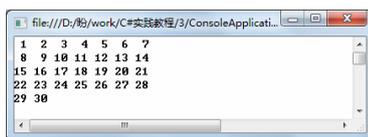


图 3-21 日期格式化

74

3.5 跳转语句

跳转语句用于中断当前执行顺序，从指定语句接着执行。在 switch 语句中曾使用跳转语句中的 break 语句，中断了当前的 switch 语句块，执行 switch 后的语句。

跳转语句同样分为多种，其关键字及其使用特点如下所示。

- ❑ **break 语句** break 语句用于终止它所在的循环或 switch 语句。
- ❑ **continue 语句** continue 语句将控制流传递给下一个循环。
- ❑ **return 语句** return 语句终止它出现在其中的方法的执行，并将控制返回给调用方法。
- ❑ **goto 语句** goto 语句将程序控制流传递给标记语句。

3.5.1 break 语句

3.2.4 节曾将 break 语句用于 switch 语句块，对 break 语句有了简单了解。break 有两种用法：

- ❑ 用在 switch 语句的 case 标签之后，结束 switch 语句块，执行 switch{} 后的语句。

□ 用在循环体，结束循环，执行循环{}后的语句。

循环有多种，任意一种循环都可以使用 `break` 跳出。接下来通过实例讲述 `break` 语句与循环语句的结合。

【练习 3-14】

851 在 100 以内有两个约数，23 和 37。找出 100 以内，851 除了 1 以外的最小约数，语句如下：

```
int num=0;
for (int i = 2; i < 101; i++)
{
    if (851 % i == 0)
    {
        num = i;
        break;
    }
}
Console.Write(num);
```

运行结果为：23。说明循环在 `i=23` 时就结束了，否则 `i` 应该为 37。`break` 语句直接跳出了 `for` 循环而不是 `if` 选择语句，也说明 `if` 选择语句中不需要使用 `break` 跳转。

3.5.2 continue 语句

`continue` 语句是跳转语句的一种，用在循环中可以加速循环，但不能结束循环。`continue` 语句与 `break` 不同之处在于：

- `continue` 语句不能用于选择语句。
- `continue` 语句在循环中不是跳出循环块，而是结束当前循环，进入下一个循环，忽略当前循环的剩余语句。

同样是找出约数，将练习 3-14 改为找出 100 以内除了 1 以外的所有约数，如练习 3-15 所示。

【练习 3-15】

找出 100 以内，除了 1 以外，851 的所有约数，语句如下。

```
for (int i = 2; i < 101; i++)
{
    if (851 % i == 0)
    {
        num = i;
        Console.WriteLine(num);
        continue;
    }
}
```

执行结果为：

23
37

练习 3-15 的例子中省略 `continue` 效果是一样的，因为循环语句中 `continue` 语句后没有语句，因此有没有 `continue` 都将执行下一个循环。`continue` 语句的使用效果如练习 3-16 所示。

【练习 3-16】

输出整型数 1 至 30，但取消整数 5 的倍数的输出，在执行到 5 的倍数时换行。使用语句如下所示：

```
for (int i = 1; i < 30; i++)
{
    if (i % 5 == 0)
    {
        Console.WriteLine("\n");
        continue;
    }
    if (i < 10)
    {
        Console.Write(" {0} ", i);
    }
    else
    {
        Console.Write(i);
        Console.Write(" ");
    }
}
```

执行结果如图 3-22 所示。从执行结果看得出，在执行至 `continue` 语句后，“`Console.WriteLine("{0} ", i);`”语句和“`Console.Write(i);`”语句都没有执行，数字 5 没有输出，而输出了换行。



图 3-22 30 以内的数字显示

3.5.3 return 语句

`return` 语句经常用在方法的结尾，表示方法的终止。方法是类的成员，将在第 5 章中介绍。

方法相当于其他编程语言中的函数，是描述某一功能的语句块。方法定义后并不是直接执行的，而是在其他地方使用语句调用的，如同变量在声明后在其他地方被调用。

方法可以有返回值，在调用时将返回值传递给调用语句，也可以没有返回值。返回值可以是常数，也可以是变量，由 `return` 语句定义返回值。

方法语句块中，在 `return` 语句后没有其他语句，但控制流并没有结束，而是找不到接下来要进行的语句。使用 `return` 语句将控制流传递给调用该方法的语句，同时将返回值传递给调用语句。

方法的定义需要有访问修饰符、返回数据类型、方法名称和参数列表，以及方法中的语句块。参数是方法中的变量，这些变量由调用它的语句来赋值，作为已经被初始化的数据来使用。

如定义一个方法 `power()` 将整型数字 `num` 求 `pow` 次方，该方法含有两个参数，一个是整型数字 `num`，另一个是该数需要获取的乘方数 `pow`，方法返回两个数的计算结果，代码如下：

```
public static int power(int num, int pow)
{
    int newnum=1;
    for (int i = 1; i <= pow; i++)
    {
        newnum = newnum * num;
    }
    return newnum;
}
```

上述代码中，通过 `for` 循环将变量 `num` 乘了 `pow` 次，并将结果赋给 `newnum`。最后一条语句使用 `return` 关键字返回运算结果 `newnum`。

方法的调用需要指定方法中的参数，如调用该方法，计算 12 的 10 次方，则调用语句如下所示：

```
power(12,10);
```

如在 `Main()` 函数中调用该方法，输出 12 的 10 次方计算结果，使用语句如下所示：

```
Console.WriteLine("数字 12 的 10 次方为{0}",power(12,10));
```

运行上述代码，其结果如下所示。

```
数字 12 的 10 次方为 61917364224
```

3.5.4 goto 语句

`goto` 语句是跳转语句中最灵活的，也是最不安全的。`goto` 语句可以跳转至程序的任意位置，但欠考虑的跳转将导致没有预测的漏洞。`goto` 语句也有限制：

- ❑ 可以从循环中跳出，但不能跳转到循环语句块中。
- ❑ 不能跳出类的范围。

使用 `goto` 语句首先要在程序中定义标签，标签是一个标记符，命名同变量名一样。标签后是将要跳转的目标语句，一条以内不用加 `{}`，超过一条则必须放在 `{}` 内，`{}` 后不用加分号。如下所示：

```
label: {}
```

接着将标签名放在 `goto` 语句后，即可跳转至目标语句，如下所示：

```
goto label;
```

练习 3-17 简单地显示了 goto 语句的用法和格式，将控制流从循环中跳出。

【练习 3-17】

输出从 1 到 10 的整数，在输出整数 5 时跳出，从整数 5 往后不再输出，使用语句如下。

```
for (int i = 1; i < 11; i++)
{
    Console.WriteLine(i);
    if (i == 5)
    {
        Console.Write("\n");
        goto comehere;
    }
}
comehere:
{
    Console.WriteLine("到 5 了，结束了");
}
```

除了跳出循环语句，goto 语句另外一种用法是跳出 switch 语句并转移到另一个 case 标签，如练习 3-18。

【练习 3-18】

定义整型 num 表示第几个季节，定义字符串 title 表示季节名称，有如下的代码：

```
int num = 4;
string title="";
switch (num)
{
    case 1:
        title = "春天";
        break;
    case 2:
        title = "夏天";
        break;
    case 3:
        title = "秋天";
        break;
    case 4:
        title = "冬天";
        goto case 2;
}
Console.WriteLine(title);
```

执行结果为：

夏天

由结果看出，因为 num 变量使用的是常数 4，所以控制流将执行 case 4，但在 case 4

中使用 `goto` 语句将控制流转向了 `case 2`，导致显示结果为 `case 2` 中的夏天。`case` 在这里起到了标签的作用。

3.6 异常处理语句

程序中不可避免存在无法预知的反常情况，这种反常称为异常。C#为处理在程序执行期间可能出现的异常提供了内置支持，由正常控制流之外的代码处理。

本节介绍 C#内置的异常处理，包括使用 `throw` 抛出异常，使用 `try` 尝试执行代码，并在失败时使用 `catch` 处理异常。

3.6.1 throw

`throw` 语句用于发出在程序执行期间出现异常的信号。通常与 `try catch` 语句或 `try finally` 语句结合使用。

`throw` 语句将引发异常，当异常引发时，程序查找处理此异常的 `catch` 语句。也可以用 `throw` 语句重新引发已捕获的异常。`throw` 只是用在程序中的一条语句，在实际应用中如练习 3-19 所示。

【练习 3-19】

定义变量 `name` 表示用户名，`name` 是不能为空的，在使用前必须赋值，为了确保 `name` 不为空，有以下语句。

```
string name = null;
if (name == null)
{
    throw (new System.Exception());
}
Console.WriteLine("The name is null");
```

执行语句将引发异常。若将 `if` 语句中 `{}` 内的语句块注释，或直接将整个 `if` 语句注释，程序可以正常进行，输出 `The name is null`，但有了 `throw` 语句，程序将中断并提示异常。

但这样的异常是放任的，没有处理的，需要使用 `catch` 语句处理异常，即下一节要讲解的 `try catch` 语句。

3.6.2 try catch

`throw` 语句只是抛出异常，异常的处理需要 `try catch` 语句。`try catch` 语句由一个 `try` 语句块后跟一个或多个 `catch` 子句构成，执行时首先尝试运行 `try` 语句块，若引发了异常则执行 `catch` 语句块并完成，若没有异常则正常完成。多个 `catch` 子句指定不同的异常处理程序。流程图如图 3-23 所示。

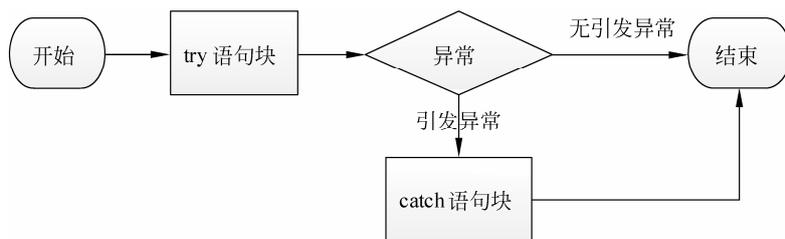


图 3-23 try catch 语句流程

图 3-23 并不是标准流程图，是形象描述的流程图。在 try catch 语句开始后首先执行 try 语句块，若无异常则结束，有异常则执行 catch 语句块并结束。try catch 语法格式如下：

```

try
{语句块}
catch ()
{语句块}
catch ()
{语句块}
  
```

格式中 try 与 catch 后的 {} 不需要加分号，catch 子句使用时可以不带任何参数，这种情况下它捕获任何类型的异常，并被称为一般 catch 子句。若 try 后只有一个 catch 语句，则 catch 后的 () 可以省略，如练习 3-20 所示。

【练习 3-20】

将练习 3-19 改进，添加异常的获取和处理，使用 try catch 语句如下。

```

string name = null;
try
{
    if (name == null)
    {
        throw new Exception();
    }
}
catch
{
    Console.WriteLine("name is null");
}
  
```

运行结果为：

```
name is null
```

catch 子句还可以接受从 System.Exception 派生的对象参数，这种情况下它处理特定的异常，如练习 3-21 所示。

【练习 3-21】

定义变量 name 表示用户名，name 是不能为空的，在使用前必须赋值，若没有赋值

则指出异常，使用语句如下。

```
string name = null;
try
{
    if (name == null)
    {
        throw new Exception();
    }
}
catch (Exception e)
{
    Console.WriteLine(e);
}
```

运行结果如图 3-24 所示：



图 3-24 指出异常

在同一个 try catch 语句中可以使用一个以上的特定 catch 子句。这种情况下 catch 子句的顺序很重要，因为会按顺序检查 catch 子句。将先捕获特定程度较高的异常，而不是特定程度较小的异常，如练习 3-22 所示。

【练习 3-22】

将练习 3-21 改变，添加一个特定程度高的 catch 子句，再使用捕获所有类型的 catch 子句，使用语句如下。

```
string name = null;
try
{
    if (name == null)
    {
        throw new ArgumentNullException();
    }
}
catch (ArgumentNullException e)
{
    Console.WriteLine("first {0}", e);
}
catch (Exception e)
{
    Console.WriteLine("second {0}", e);
}
```

执行结果如图 3-25 所示。



图 3-25 多个 catch 结果

除了 throw 语句和 try 语句块中抛出的异常，由 catch 语句可以再次引发异常，格式如下：

```
//参数可以省略
catch(参数)
{
    throw(参数);
}
```

在 try 语句块内部时应该只初始化其中声明的变量；否则，完成该块的执行前可能发生异常。如以下的代码示例：

```
int i;
try
{
    i = 0;
}
catch
{
}
Console.Write(i);
```

变量 i 在 try 语句块外声明，而在语句块内初始化。在使用 Write(i) 语句输出时产生编译器错误：使用了未赋值的变量。

3.6.3 try catch finally

finally 语句块用于清除 try 语句块中分配的所有资源，以及在 try 语句块结束时必须执行的代码，无论是否有异常发生。

finally 语句块放在 catch 语句块后，控制总是传递给 finally 语句块，与 try 语句块的退出方式无关。流程图如图 3-26 所示。

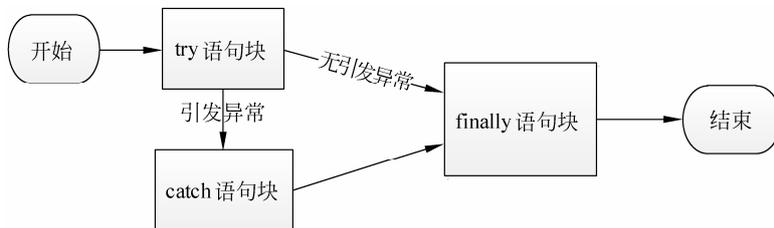


图 3-26 try catch finally 流程图

图 3-26 并不是标准流程图，是形象化的流程图。程序从进入 `try` 语句后若没有引发异常，则进行 `finally` 语句块并结束；若引发了异常，则先进行 `catch` 语句块，接着执行 `finally` 语句块并结束。语法如下：

```
try { }  
catch { }  
finally { }
```

`catch` 和 `finally` 一起使用的常见方式是：在 `try` 语句块中获取并使用资源；在 `catch` 语句中处理异常；在 `finally` 语句块中释放资源。`catch` 语句块可以省略，直接使用 `try` 和 `finally` 语句块。

【练习 3-23】

定义除数变量 `dividenum` 和被除数变量 `num`，两个变量都不能为 0，默认 `num` 为 6，`dividenum` 为 2，求两个数的商，有以下语句。

```
int num=0;  
int dividenum = 0;  
try  
{  
    if (num == 0 || dividenum == 0)  
        { throw new Exception(); }  
}  
catch  
{  
    num = 6;  
    dividenum = 2;  
}  
finally  
{ num = num / dividenum; }  
Console.Write(num);
```

执行结果为：

3

从练习 3-23 看得出，语句执行从 `try` 开始，在引发异常后由 `catch` 捕获并处理，之后交由 `finally` 语句。代码中的 `catch` 可以省略，如以下代码：

```
int num = 0;  
int dividenum = 0;  
try  
{  
    if (num == 0 || dividenum == 0)  
    {  
        num = 6;  
        dividenum = 2;  
    }  
}
```

```

}
finally
{ num = num / dividenum; }
Console.Write(num);

```

执行结果与练习 3-23 一致。

与直接使用 try catch 不同，在 finally 语句块中可以初始化 try 语句块以外的变量，如以下代码：

```

int i;
try
{

}
catch
{

}
finally
{i=123;}
Console.Write(i);

```

执行结果为：

```
123
```

3.7 实验指导 3-1：输出等腰梯形

使用一种符号，如 '@'、'#'、'*' 或 '\$' 等，输出一个等腰梯形，在梯形的中间垂直轴线使用另一种符号，达到如下所示的效果。

```

*****$*****
*****$*****
*****$*****
*****$*****
*****$*****

```

通过实现效果看得出，图形是有规律地循环输出，需要用到循环语句。而图像由两部分构成：

- 一部分是符号，构成梯形的主体。
- 另一部分是空格，用来控制格式，使输出为等腰梯形。

但两部分不能分开，每一行都要有符号和空格，因此两部分的关系是并列的，可以用两个变量表示两部分的字符串。

整体的效果：梯形由 5 行构成，每一行又分为对称的两部分。以对称轴左侧为例，符号每一行多一个，符号数目与空格数目的和为 10。两边的符号数目和即为 10 减去空

格数，乘以 2。中间轴的另一个符号需要使用条件语句，当进行到中间时改变符号，并接着进行下一个循环。

因此，对该图形的输出，首先需要确定整体循环的次数，5 行的图形循环 5 次。

接着是内部的循环，先看空格：空格每一行少一个，总数需要递减。循环数要跟整体循环关联，否则每次循环数一样，将输出矩形的空格。因此只需将总循环数递减，即可使空格数目与总循环数相等。

再看符号，符号与空格数的关系已经明确，即 $(10 - \text{空格数}) \times 2$ ，但因中间有其他符号，可以使用条件语句在循环至中间时改变符号，并接着执行下一个循环，需要使用跳转语句 `continue`。

每个循环都需要将变量字符串累加，但每次循环前，若字符串不为空字符，则输出结果与设想不同。因此在每一行结束时，变量字符串需要清空。

定义每一行的字符串变量 `trapezoid`；定义空格部分字符串变量 `trapezoid1`；定义字符部分字符串变量 `trapezoid2`，具体代码如下：

```
string trapezoid="";
string trapezoid1 = "";
string trapezoid2 = "";
for (int i = 5; i >0; i--)
{
    for (int j = i; j >0;j-- )
    {
        trapezoid1 = trapezoid1 + " ";
    }
    for (int k =( 10 - i)*2; k >= 0; k--)
    {
        if (k == 10 - i)
        {
            trapezoid2 = trapezoid2 + "$";
            continue;
        }
        trapezoid2 = trapezoid2 + "*";
    }
    trapezoid = trapezoid1 + trapezoid2;
    Console.WriteLine(trapezoid);
    trapezoid = "";
    trapezoid1 = "";
    trapezoid2 = "";
}
```

执行结果如图 3-27 所示。

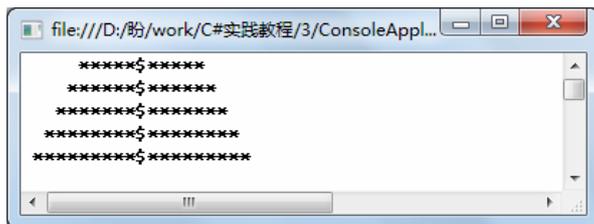


图 3-27 等腰梯形

3.8 思考与练习

一、填空题

1. 选择语句有 if 语句、if else 语句、_____和 switch 语句。
2. 跳转语句有 break 语句、continue 语句、_____和 goto 语句。
3. throw 语句属于_____语句。
4. 2 个人参加选举（李贺、林峰），分别用 2 个整型变量表示他们的票数，则应填入横线的内容是_____。

```
switch (name)
{
    case "李贺":
        numLH ++;
        break;
    case _____:
        numLF ++;
        break;
}
```

5. do while 循环先执行语句块，后进行_____判断。

二、选择题

1. 下列选项中，不属于嵌套的是_____。

A.

```
for ()
{ if () {} }
```

B.

```
for ()
{ for ()
{ }
```

```
}
```

C.

```
switch ()
{
    case
    break;
}
```

D.

```
if ()
{ if () {} }
```

2. 以下说法不正确的是_____。
 - A. continue 语句不能用于选择语句
 - B. 一个分号就能表示一条语句
 - C. if 语句块 {} 后不需要分号
 - D. if 条件语句的 () 内有 3 个表达式，因此有 3 个分号
3. 以下_____不属于跳转语句。
 - A. break 语句
 - B. throw 语句
 - C. continue 语句
 - D. return 语句
4. 以下代码的输出结果中有_____个

```
for (int a = 0; a < 6; a++)
{
    for (int i = 0; i < a; i++)
    {
        Console.Write(a);
    }
    Console.WriteLine("");
}
```

A. 4 个

B. 3 个

C. 2 个

D. 1 个

5. 以下语句块中, 不能对语句块外的变量赋值的是_____。

- A. if 语句块 B. for 语句块
C. try 语句块 D. while 语句块

6. 以下代码的输出结果是_____。

```
int i;  
try  
{ i = 0; }  
catch  
{ i = 1; }  
finally  
{ i = 2; }  
Console.Write(i);
```

A. 0

B. 1

C. 2

D. 产生编译器错误: 使用了未赋值的变量

三、简答题

1. 简要概述语句的分类。
2. 简单说明 if 和 switch 的区别。
3. 简单说明 for 和 do while 的区别。
4. 简述跳转语句的种类。