第 1 章

初步体验 C#

如果本书的第 I 部分应该描述一般性的对象概念，那么为什么要从关于 C#的介绍性章节开始呢？

- 可以确定的是，对象都是“语言中立”的，因此从本书第 I 部分中学到的对象基本概念以及从本书第 II 部分中学到的对象建模，都可以在任何面向对象的编程语言(OOPL)中得到实现。
- 阅读少量的代码示例有助于掌握对象概念，但是也可以使用与语言无关的伪代码——一种表达计算机逻辑的自然语言方式，这种方式不拘泥于 C#等任何具体语言的语法——作为第 I 部分和第 II 部分中的代码示例。

回到最初的问题：为什么要这么早就介绍 C#语法？原因在于我们希望您能从一开始就习惯于 C#语言，编写本书的目的不仅是介绍对象和对象建模，而且最终是为了显示如何将对象翻译为 C#语言代码。因此，在第 I 部分和第 II 部分的代码示例中，为了隐藏某些复杂的逻辑，我们确实会使用一些伪代码，但仍然会主要使用实际的 C#语法。只要记住，除非特别标明，否则在本书第 I 部分和第 II 部分中学习到的对象概念同样适用于其他 OOPL。

本章主要介绍如下主题：

- C#编程语言的许多优点
- 预定义的 C#类型、这些类型的运算符以及使用这些类型构成的表达式
- 详细分析一个简单的 C#程序
- C#的代码块结构
- 各种类型的 C#表达式
- 循环和其他流程控制结构
- 向屏幕输出消息，主要用于测试代码的运行情况
- C#编程样式的要素

如果已经精通 C、C++或 Java 编程技术，您就会发现许多 C#语法与这些编程技术非常类似，并且应该能够快速浏览本章。

如果您已经完全了解了 C#语言的基础知识，则可以跳至第 2 章开始学习。

1.1 C#入门指南

可能您已经急于开始编写、编译和运行 C#程序，但是我们特意没有现在就介绍如何在您的计算机上下载、安装 C#和.NET Framework，也没有急于介绍如何编译程序等细节内容。以下是本书的组织方式：

- 本书第Ⅰ部分集中介绍对象的概念——也就是“什么”是对象；我们并不想让您分心于了解在机器上配置 C#开发环境的各种细节，您应该专心学习这些基本概念。
- 本书第Ⅱ部分集中介绍对象建模——也就是“如何”设计应用程序以最有效地使用对象。我们并不想让您在没有绘制适当的 OO“蓝图”之前就尝试编程。
- 第Ⅲ部分是“最终的结局”——使用 C#代码实现对象模型，编写可用的学生选课系统(Student Registration System，SRS)。

本书将通篇介绍 C#代码，但是在第Ⅰ部分和第Ⅱ部分中将主要介绍代码片段和简单的示例。当学习到本书的第Ⅲ部分时，您应该已经很好地理解了 OO 编程概念，这时就会开始真正深入研究如何开发完整的 C#应用程序。

1.2 使用 C#的原因

可以使用任何 OOPL 构建 SRS 系统，那么为什么要使用 C#呢？继续往下阅读，您将很快了解具体的原因。

1.2.1 实践出真知

C#的设计人员仔细研究了之前的其他 OOPL，从中吸取经验和教训。他们借鉴了 C++、Java、Eiffel 和 Smalltalk 等语言的最佳特性，并且添加了这些语言所没有的一些功能和特性。另一方面，他们也抛弃了在其他语言中被证明是最为麻烦的特性。结合以上两点，C#是功能强大而又易于学习的编程语言。

这并不是说 C#是完善的语言——任何语言都不是完善的！——但它确实比之前的许多语言有了明显的改进。

1.2.2 C#是集成应用开发架构的一部分

C#语言集成在微软的.NET Framework 中，.NET Framework 是微软用于开发应用程序和管理其运行时环境的功能强大的综合性平台，它主要支持 C#、C++、J#和 Visual Basic 编程语言，但也提供了称为跨语言互操作性(Cross-Language Interoperability)的功能，通过该功能可以使通过不同编程语言创建的对象彼此协作。.NET Framework 的一个核心要素是公共语言运行库(Common Language Runtime，CLR)，它负责对任何.NET Framework 程序执行运行库管理，加载和运行.NET Framework 程序，并且为.NET Framework 程序提供支持服务。

通过公共语言规范(Common Language Specification，CLS)，.NET Framework 提供了

它所支持的各种语言(C#、C++、Visual Basic 和 JavaScript)之间高度的协同工作能力。CLS 定义了每种.NET 语言都必须遵守的一组公共的类型和行为，允许开发者将 C#代码与采用其他任何语言编写的代码无缝地集成在一起。

.NET Framework 也包含一个大型的库集合，称为.NET Framework 类库(.NET Framework Class Library，FCL)，FCL 提供了在 Windows 平台上开发应用程序所需的几乎所有通用功能。您将发现 FCL 已经为您完成了许多编程工作，其范围包括文件访问、数学计算和数据库连接等。C#语言和.NET Framework 为您的所有编程需求提供了“一站式购物”。

可以从如下地址中查找关于.NET Framework 的更多介绍：

http://msdn.microsoft.com/en-us/library/default.aspx。

1.2.3 C#是彻底的面向对象语言

在 C#和 Java 这些较新的 OOPL 出现之前，最为广泛使用的一种 OOPL 是 C++。C++实际上是对非 OOPL 的 C 语言的面向对象扩展。因此，C++提供了许多“后门”，从而开发人员可以非常容易地编写“非 OO”代码。实际上，许多熟练的 C 程序员只是将 C++看作是“更好的 C”，而没有适当地学习如何设计面向对象的应用程序，因此在大多数情况下都以过程语言(非 OO)的方式来使用 C++。

实际情况是，C#从一开始就是纯粹的 OOPL。在本章后面的详细讨论中，我们将会指出 C#中的一切都是对象：

- 基本的值类型，如 int 和 double，都从 Object 类继承而来。
- 所有的图形用户界面(Graphical User Interface，GUI)构件——窗口、按钮、文本输入栏、滚动栏、列表和菜单等——都是对象。
- 所有函数都附加到对象上，这些函数称为方法。C/C++中有脱离对象的方法，而 C#中没有。
- 甚至是 C#程序的入口点(现在称为 Main 方法)都不再独立存在，而是被绑定到类中，本章后面将详细说明具体的原因。

因此，C#特别适合于编写面向对象的应用程序，然而，在本章的前言中已经指出，仅仅使用这种面向对象的语言并不能保证所产生的应用程序真正做到面向对象！除此之外，您必须理解本书的两个基本目标：(a)如何从根本上设计应用程序以最有效地使用对象；(b)如何正确地使用 C#语言。

1.2.4 C#是免费的语言

C#最有价值的特性之一在于它是免费的语言！您可以免费从微软开发者网络(Microsoft Developer Network，MSDN)中下载 C#编译器以及所需的其他所有类库和实用程序。本书的附录 A 中将详细介绍如何在您的机器上安装 C#。

1.3 C#语言基础

对于以前从未见过 C#代码的读者，本章的剩余部分将介绍 C#编程语言的基本语法。记住，现在只是简单介绍 C#，这些知识仅仅足够帮助您理解本书第 I 部分和第 II 部分中的代码示例。在本书的第III部分(第 13~16 章)中将更为深入地介绍 C#语言，通过其构建一个功能完备的学生选课系统(SRS)应用程序。

注意

如果还没有花时间阅读本书的前言，那么现在正是时候！在前言中已将 SRS 系统的需求作为一个案例进行了介绍。

对比伪代码和真正的 C#代码

在本章开始时就提及，本书第 I 部分和第 II 部分的代码示例中将偶尔使用一些伪代码来隐藏与示例本身关系不大的逻辑细节。在同时使用伪代码和真正的代码时，为了避免产生混淆，使用斜体字表示伪代码，而使用普通字体表示真正的代码。

下面是真正的 C#语法：

```
for (int i = 0; i <= 10; i++) {
```

而下面是伪代码：

```
  compute the grade for the ith Student
}
```

本书后面还会多次提醒您这一点，使您不会在无意中尝试输入伪代码并对其进行编译。

1.4 详细分析一个简单的 C#程序

图 1-1 显示了一个最简单的 C#应用程序，即经典的“Hello”程序。

图 1-1 详细分析一个简单的 C#程序

下面依次讨论这个简单程序中的关键元素。

1.4.1 “using System;”语句

该程序的第一行

```
using System;
```

是使得程序可以正确编译和运行的必要条件，它向编译器提供在 System 名称空间中定义的类型信息，名称空间是多个预定义 C#编程元素的逻辑分组(在 C#中则是前面提及的 FCL 的一部分)。

在本书的第 13 章中将会详细介绍名称空间，现在只需要记住 using System;语句是使得程序(特别是 Console.WriteLine(“Hello!”);这一行代码)可以正确编译的必要条件。

using 是一个 C#关键字，关键字也称为保留字，它代表语言中有特殊含义的单词，因此程序员不可以使用关键字作为变量、函数或后面将学习的其他 C#构件(building block)的名称。在学习本书的过程中，您将会遇到其他许多 C#关键字。

1.4.2 注释

程序中接下来的一行是单行注释：

```
//这个简单的程序演示了一些基本的 C#语法。
```

除了单行注释之外，C#语言也支持带分隔符的 C 语言注释样式，这种注释可以跨越多行。带分隔符的注释以正斜杠后跟星号(/*)开头，而以星号后跟正斜杠结束(*/)，这两个分隔符之间的任何字符都被视为注释，因此会被编译器忽略，无论注释跨越多少行都是如此。

```
/* 这是单行的 C 样式注释。*/

/* 这是多行的 C 样式注释，采用这种注释可以临时注释掉整段代码，而不直接删除它们。当编
   译器遇到第一个“斜杠星号”时，它就会将接下来输入的内容视为注释；甚至是合法的代码
   行(如下所示)都被视为注释行，从而被编译器忽略，直到遇到配对的“星号斜杠”。
x = y + z;
a = b / c;
j = s + c + f;
*/
```

注意

还有用于 XML 文档文件中的第三种 C#注释类型。在 XML 文档中，采用三条斜杠(///)表示注释。

注意，注释不能嵌套；也就是说，如下的代码将不能通过编译：

```
/* 注释开始…
x=3;
```

```
/* 哇！在第一个注释结束之前，我们错误地尝试嵌套第二个注释！这将导致编译问题，因为编
译器将忽略第二个(内部)注释的开始符号。我们已经位于第一个注释中，因此在尝试终止第二个
(内部)注释时，编译器会认为我们是在终止第一个(外部)注释…*/
z=2;
*/ 然后，当我们尝试终止第一个(外部)注释时，编译器将提示这行代码无效。
```

当编译器在最后一行代码中遇到我们希望的“外部”注释结束符号*/时，将报告如下的编译错误：

```
error: Invalid expression term '/'
error: ; expected
```

1.4.3 类声明/“包装器”

接下来介绍类的“包装器”——更合适的名称是类声明(class declaration)，其采用{…}这样的形式，其中花括号包括类执行的主要逻辑，同时也包括类的其他构件，如下所示：

```
class name

{...}

e.g.,

class SimpleProgram
{...}
```

在后面的章节中，您将学习类的所有相关内容，如何命名类，特别是为什么一开始就需要类包装器。现在只需要注意到 class 是 C#的另一个关键字，而 SimpleProgram 是我们发明的一个名称即可。

1.4.4 Main 方法

在 SimpleProgram 类声明中，可以看到程序的起始函数，该函数在 C#中称为 Main 方法。Main 方法是 C#程序的入口点。在调用程序可执行文件时，系统将调用 Main 方法来启动应用程序。

注意

对于类似于上面示例的简单程序，可以在单个方法中包含所有处理逻辑。另一方面，对于更复杂的应用程序，Main 方法就可能无法包括整个系统的所有处理逻辑。在本书后面的章节中，您将学习如何构造突破 Main 方法的局限性的应用程序。

Main 方法的第一行如下所示：

```
static void Main() {
```

该行定义了 Main 方法的头(header)，必须如示例中所显示的那样编写(在第 13 章中将会介绍一种例外情况，即在需要有选择地接受命令行参数时的情况)。

Main 方法的主体包括在花括号中({…})，其中包括了一行语句：

```
Console.WriteLine("Hello!");
```

该语句在屏幕上输出如下信息：

```
Hello!
```

后面将会详细讨论该语句的语法，但是现在需要注意到在语句的末尾使用了分号。如同在 C、C++和 Java 中那样，将分号放置在每条 C#语句的末尾。花括号{…}用于分隔代码块，在本章后面的“代码块和变量作用域”一节中将对此进行详细讨论。

在更为复杂的程序中，一般还会在 Main 方法中声明变量、创建对象和调用其他方法。

现在我们已经见过了简单的 C#程序，接下来将更为详细地研究一些基本的 C#语法特性。

1.5 预定义类型

一般来说，C#是一种强类型的编程语言，在声明变量的同时也必须声明其类型。声明变量类型的一种作用是告诉编译器为该变量分配多少内存空间。

C#语言和.NET Framework 使用了公共类型系统(Common Type System，CTS)，这是定义一组类型以及这些类型的行为的规范。CTS 定义了多种类型，这些类型分为两个系列(family)：值类型和引用类型。值类型和引用类型也可以被声明为泛型，这意味着它们可以代表多种类型。本章将集中讨论 C#的预定义值类型(也称为简单类型(simple type))，同时也会介绍 string 类型，然而 string 类型是一个引用类型。

C#语言支持各种简单类型，其中最常用的类型如下所示(它们都是 C#关键字)：

- bool：布尔值，可以为 true 或 false
- char：16 位的 Unicode 字符
- byte：8 位的无符号整数
- short：16 位的有符号整数
- int：32 位的有符号整数
- long：64 位的有符号整数
- float：32 位的单精度浮点数
- double：64 位的双精度浮点数

声明为简单类型的每个变量可以表示单个整数、浮点数、布尔值、字节或字符值。

1.6 变量

如前所述，在程序中使用变量之前，必须声明该变量的名称和类型。在初次声明变量时可以提供初始值，也可以在程序的后面对该变量赋值。例如，下面的代码片段声明了两个简单类型变量。第一个变量的类型是 int，在声明该变量时为其提供初始值。第二

个变量声明的类型是 double，在随后的代码行中将对其进行赋值。

```
int count = 3;

double total;
// 中间的代码…省略细节
total = 34.3;
```

使用 true 或 false 关键字为 bool 类型的变量赋值：

```
bool blah;
blah = true;
```

布尔类型变量通常用作表明是否应该有条件地执行某些代码的标识，如同下面的示例所示：

```
bool error = false; // 初始化标识。
//...

// 在后面的程序(伪代码)中
if (发生某些错误情况) {
  // 将标识设置为 true 以表明发生错误。
  error = true;
}
//...

// 在后面的程序中
if (error == true) {
    // 伪代码。
    执行正确的操作
}
```

注意

本章后面将会专门讨论 if 语句的语法，if 语句是 C#中的一种流程控制语句。

可以将字面值赋给 char 类型的变量，在赋值时将该值(单个 Unicode 字符)放在单引号中，如下所示：

```
char c = 'A';
```

1.6.1 变量的命名约定

大多数变量名使用所谓的 Camel 命名法：变量名中的第一个字母采用小写，随后每个单词中的第一个字母采用大写，其他字母都采用小写。

注意

在随后的章节中介绍其他对象概念时，将会详细阐述命名变量的规则。

例如，下面的变量名就遵循了 C#的变量命名约定：

```
int grade;
double averageGrade;
string myPetRat;
bool weAreFinished;
```

如前所述，不能将 C#关键字用作变量名：

```
int float; //不能通过编译——“float”是关键字
```

1.6.2 变量初始化和赋值

在 C#中，根据在程序中声明变量的方式和位置，存在多种不同类型的变量。在声明某些变量类型时，使用默认值对其进行初始化。局部变量(local variable)是在方法或其他代码块中声明的变量，在声明这种变量时没有为其提供任何默认值，因此在语句中访问该变量的值之前，必须显式地为其赋值。例如，在如下的代码片段中声明了两个局部整型变量：foo 和 bar。这段代码为变量 foo 赋值，但是没有为变量 bar 赋值，然后尝试将两个变量相加：

```
static void Main() {
        int foo; // 局部变量
        int bar; // 另一个局部变量

        foo = 3; // 为 foo 赋值，但是没有为 bar 赋值。
        foo = foo + bar; // 这一行不能通过编译
```

如果尝试编译这段代码，就会获得关于其中最后一行代码的如下编译错误消息：

```
error: use of unassigned local variable 'bar'
```

编译器表明已经声明了局部变量 bar，但是没有定义它的值。为了修正这个错误，需要在尝试将 bar 的值与 foo 的值相加之前显式地对 bar 进行赋值：

```
int foo;
int bar;

foo = 3;
bar = 7; // 现在对这两个变量赋值。

foo = foo + bar; // 这一行现在正确编译。
```

注意

关于变量初始化的内容，其复杂程度已经超出了此处的讨论范围。在第 13 章中处理

对象的内部工作时，您将学习到不同的自动初始化规则。

1.7 字符串

本章接下来讨论更为重要的一种预定义类型，即 string 类型。

注意

需要记住的是，与本章中介绍的其他 C#类型不同，string 并不是值类型，而是引用类型，前面已经提及过这一点。此处只是简要讨论字符串，在第 13 章中才会详细讨论 string 作为引用类型的意义所在。

string 变量类型表示一系列 Unicode 字符。可以通过多种方法创建和初始化 string 类型变量。其中最简单和最常用的方法是声明 string 类型的变量，然后使用字符串字面值(string literal)对该变量赋值，字符串字面值就是用双引号包括起来的任何文本：

```
string name = "Zachary";
```

注意，在为 string 类型变量赋值时，使用双引号而非单引号来包括字符串字面值，即使该字符串字面值只包含一个字符也是如此：

```
string shortString = "A";  // 在将字面值赋给 string 类型变量时使用双引号
string longString = "supercalifragilisticexpialadocious"; // (同上)

char c = 'A';              // 在将字面值赋给 char 类型变量时使用单引号
```

将初始值作为占位符赋给 string 类型变量的两种常用方法如下所示：

- 设置该初始值等于空字符串，表示为两个连续的双引号：

```
string s = "";
```

- 设置该初始值等于保留字 null，null 是 string 类型中“等价于零”的值(后面将会介绍其他引用类型/对象的等价于零的值)：

```
string s = null;
```

加号(+)运算符通常用于执行加法操作，但是当其和 string 类型变量结合使用时，则表示执行字符串连接操作。可以使用+运算符将任意数量的 string 类型变量或 string 字面值连接在一起：

```
string x = "foo";
string y = "bar";
```

```
string z = x + y + "!"; // z 现在等于"foobar!"；x 和 y 不变
```

第 13 章中将介绍许多可以对字符串执行的操作，并将深入介绍字符串的面向对象特性。

1.8 区分大小写

C#是区分大小写的语言，即在 C#中必须区分大写和小写的使用。例如：

- 拼写相同但是大小写不同的变量名具有不同的用途：例如，x(小写)和 X(大写)就表示不同的变量。
- 所有关键字都采用小写：public、class、int、bool 等。不要“创造性地”使用大写字母形式的关键字，编译器将强烈反对这一点！
- Main 方法名称的首字母必须大写。

1.9 C#表达式

C#中的简单表达式如下所示(此外，还有一些必须与对象协同使用的表达式类型，第 13 章中将介绍这些表达式)：

- 常量：7、false
- char(字符)字面值：‘A’、‘&’
- string 字面值：“foo”
- 到目前为止已经讨论过的声明为预定义类型的变量名：myString、x
- 使用某个 C#二元运算符(本章后面将详细讨论)组合的上述任意两项：x+2
- 使用某个 C#一元运算符(本章后面将详细讨论)修改的上述任意一项：i++
- 用括号包括的上述任意简单表达式：(x+2)

1.9.1 赋值语句

使用赋值运算符=实现对变量的赋值。赋值语句由放置在=左边的(已经声明的)变量名以及放置在=右边的可计算出类型合适的值的表达式组成。例如：

```
count = 1;

total = total + 4.0; // 假设 total 被声明为双精度类型变量

price = cost + (a + b)/length; // 假设已经正确声明所有变量
```

1.9.2 算术运算符

C#语言提供了许多基本算术运算符，如表 1-1 所示。

表 1-1

+	相加
-	相减
*	相乘
/	相除
%	求模(%运算符左边的操作数除以右边的操作数得到的余数)

+和-运算符也可以作为前缀，表示正数和负数：-3.7、+42。

除了简单赋值运算符=之外，还有一些特殊的组合赋值运算符，它们将变量赋值与算术操作结合起来。用于算术操作的组合赋值运算符如表 1-2 所示。

表 1-2

+=	a += b 等同于 a = a + b
-=	a -= b 等同于 a = a - b
*=	a *= b 等同于 a = a * b
/=	a /= b 等同于 a = a / b
%=	a %= b 等同于 a = a % b

注意

组合赋值运算符没有增加任何新的功能，提供它们只是为了方便简化代码。例如，

```
total = total + 4.0;
```

可以替换为

```
total += 4.0;
```

此处要介绍的最后两个算术运算符是递增(++)和递减(--)运算符，它们用于给整型变量的值加 1 或减 1，或者给浮点型变量的值加 1.0 或减 1.0。递增和递减运算符也可以用于 char 变量。例如，考虑如下代码片段：

```
char c = 'e';
c++;
```

在执行这段代码时，变量 c 将被赋予值‘f’，因为‘f’是 Unicode 有序字符系列中字符‘e’后面的字符。

递增和递减运算符可用作前缀或后缀。

如果将运算符放置在它所操作的变量之前(作为前缀)，则在将该变量的值用于语句

中的任何赋值操作之前执行递增或递减操作。

如果将运算符放置在它所操作的变量之后(作为后缀)，则在将该变量的值用于语句中的任何赋值操作之后执行递增或递减操作。

例如，考虑如下代码片段，其中使用了前缀递增(++)运算符：

```
int a = 1;
int b = ++a; // a 的值递增为 2，然后 b 被赋值 2
```

执行这两行代码之后，变量 a 的值将为 2，变量 b 的值也为 2，这是因为在第二行代码中，在将 a 的值赋给 b 之前执行对变量 a 的递增操作(从 1 到 2)。以上两行代码在逻辑上等同于如下 3 行代码：

```
int a = 1;
a = a + 1;
int b = a;
```

现在查看递增运算符在相同代码片段中用作后缀的情况：

```
int a = 1;
int b = a++; // b 将被赋值为 1，然后 a 递增为 2
```

在执行这两行代码之后，变量 b 的值将为 1，而变量 a 的值则是 2，这是因为在第二行代码中，在将变量 a 的旧值赋给变量 b 之后执行对变量 a 的递增操作(从 1 到 2)。以上两行代码在逻辑上等同于如下 3 行代码：

```
int a = 1;
int b = a;
a = a + 1;
```

下面是稍微复杂一些的示例：

```
int y = 1;
int z = 2;
int x = y++ * ++z; // x 将被赋值为 3，因为 z 的值在用于乘法操作之前从 2 递增到 3，
                   // 而 y 的值在被使用之前保持为 1。
```

稍后将会看到，递增和递减运算符经常用于循环和其他流程控制结构中。

1.9.3 求值表达式和运算符优先级

可以通过在各种不同简单表达式的周围嵌套括号来构建任意复杂的表达式，如((((4/x) + y) * 7) + z)。编译器从内到外、从左到右计算每对括号内的表达式。假设已经声明和初始化 x、y 和 z，如下所示：

```
int x = 1;
```

```
int y = 2;
int z = 3;
```

则如下赋值语句右边的表达式：

```
int answer = ((8 * (y + z)) + y) * x;
```

将被依次求值如下：

```
((8 * (y + z)) + y) * x
  ((8 * 5) + y) * x
    (40 + y) * x
      42 * x
        42
```

如果没有括号，则某些运算符在用于表达式求值时将获得较高的优先级。例如，乘法或除法操作默认在加法或减法操作之前执行。可以通过使用括号显式地修改运算符的自动优先级，括号内的操作将在括号外的操作之前执行。考虑如下代码片段：

```
int j = 2 + 3 * 4; // j 将被赋值 14
int k = (2 + 3) * 4; // k 将被赋值 20
```

在第一行代码中没有使用括号，乘法操作先于加法操作执行，因此整个表达式的求值结果为 2+12=14，这相当于显式地写为“2+(3*4)”。

在第二行代码中显式地使用括号来包括操作“2+3”，因此加法操作将首先执行，相加的结果与 4 相乘，得到整个表达式的值为 5*4=20。

1.9.4 逻辑运算符

逻辑表达式以指定的方式比较两个(简单的或复杂的)表达式 *exp1* 和 *exp2*，求出一个可能为 true 或 false 的布尔值。

为了创建逻辑表达式，C#提供了以下关系运算符，如表 1-3 所示。

表 1-3

exp1 == *exp2*	如果 *exp1* 等于 *exp2*，则为 true(注意，使用了两个等于号)
exp1 > *exp2*	如果 *exp1* 大于 *exp2*，则为 true
exp1 >= *exp2*	如果 *exp1* 大于等于 *exp2*，则为 true
exp1 < *exp2*	如果 *exp1* 小于 *exp2*，则为 true
exp1 <= *exp2*	如果 *exp1* 小于等于 *exp2*，则为 true
exp1 != *exp2*	如果 *exp1* 不等于 *exp2*，则为 true(!读作“不”)
!*exp*	如果 *exp* 为 false，则为 true；如果 exp 为 true，则为 false

除了这些关系运算符，C#还提供了逻辑运算符，逻辑运算符可以与关系运算符结合

使用，从而创建涉及多次比较的复杂逻辑表达式。表 1-4 列出了各种逻辑运算符。

表 1-4

&&	逻辑“与”
||	逻辑“或”
!	逻辑“非”(!运算符使逻辑表达式的值从 true 变为 false 或相反)

逻辑“与”和“或”运算符是二元运算符，它们左右两边的操作数都必须是有效的逻辑表达式，从而才可以计算得出布尔值。如果使用了&&运算符，则左右两边的操作数都必须为 true 才能使组合逻辑表达式为 true。使用||运算符时，只要左边或右边任意一个操作数为 true，则组合逻辑表达式为 true。

下面的示例使用逻辑“与”运算符编写如下组合逻辑表达式：如果 x 大于 2.0 且 y 不等于 4.0。

```
if (x > 2.0 && y != 4.0) {
    // 伪代码
    do some stuff...
}
```

注意，由于>和!=运算符比&&运算符有更高的优先级，因此不需要像下面这样插入额外的括号：

```
if ((x > 2.0) && (y != 4.0)) {
    // 伪代码.
    do some stuff...
}
```

逻辑表达式在流程控制结构中很常见，本章后面将讨论这些控制结构。

1.10 隐式类型转换和显式类型转换

C#支持隐式类型转换(implicit type conversion)。例如，尝试将某个变量 y 的值赋给另一个变量 x，如下所示：

```
x = y;
```

并且这两个变量最初被声明为不同的类型，则 C#仍然会尝试执行赋值操作，自动将 y 的值的类型转换为 x 的类型，但前提是执行该操作不能损失数据精度(在这一方面，C#不同于 C 和 C++，因为后两者即使损失精度也会执行自动类型转换)。通过查看如下的示例可以很好地理解这一点：

```
int x;
double y;
y = 2.7;
x = y; // 尝试将 double 值赋给 int 变量；这行代码在 C 和 C++中可以通过编译，而在 C#
       // 中不可以
```

在上述的这段代码中，尝试将 y 的双精度值 2.7 复制到 x 中，而 x 被声明为 int 类型。如果执行这条赋值语句，则会截去 y 的小数部分，从而 x 获得整数值 2。这样就发生了精度损失，也称为窄化转换(narrowing conversion)。C 或 C++编译器允许执行这种赋值，隐式地截短值；然而，C#编译器不会假设我们意图执行这种截短操作，而是在最后一行生成一个错误：

```
Error:: Cannot implicitly convert type 'double' to type 'int'
```

为了明确地告诉 C#编译器我们愿意接受精度损失，就必须执行显式类型转换(explicit cast)，将前面表达式中的值转换为括号中的所需目标类型。换句话说，必须按照如下方式重新编写上述示例中的最后一行，让 C#编译器接受它：

```
int x;
double y;
y = 2.7;
x = (int) y; // 这段代码现在将通过编译，C#编译器可以“松口气”，因为明确地告诉它
             // 想要执行窄化转换
```

当然，如果执行反向赋值，如下所示，则 C#不会对最后一行产生任何问题，因为在这个示例中将较低精度的值(2)赋给具有较高精度的变量 y，即 y 的值将为 2.0：

```
int x;
double y;
x = 2;
y = x;       // 将 int 值赋给 double 变量；y 的值将是 2.0
```

这种转换称为宽化转换(widening conversion)；在 C#中自动执行这种转换，并且不需要执行显式的类型转换。

注意，在 C#中将常量值赋给 float 变量时有一种特殊情况；语句

```
float y = 3.5;   // 不能编译!
```

将生成一个编译器错误，因为 C#会将类似于 3.5 这样的带有小数部分的数字常量值自动视为具有更高精度的 double 值，所以编译器将会因为精度损失而再次拒绝执行转换。

为了执行这样的赋值，必须显式地将浮点常量转换为 float 值：

```
float y = (float)3.5; // 可以编译；此处使用了类型转换。
```

或者，也可以使用后缀 F，强制编译器将赋值语句右边的常量视为浮点值：

```
float y = 3.5F; // 可以编译，因为我们指示将该常量视为浮点值，而非双精度值。
```

提示

还有一种可选方法是直接使用 double 变量而非 float 变量来表示浮点数值。在 SRS 应用程序中声明浮点变量时，一般使用 double 而非 float，这样可以避免类型转换操作。

不可以对 char 类型执行隐式的转换，另外，也不可以将 bool 类型转换为另一种类型(无论是隐式或显式)。

本书后面将会介绍其他与对象相关的类型转换。

1.11 循环和其他流程控制结构

很少有程序从头到尾逐行地连续执行。相反，程序的执行流程有一定的条件。可能需要让程序在满足条件时执行某个代码块，而在不满足该条件时执行另一个代码块。程序可能必须重复执行相同的代码块。C#语言提供了许多不同类型的循环和其他流程控制结构来处理这些情况。

1.11.1 if 语句

if 语句是基本的条件分支语句，在满足表示为逻辑表达式的某个条件时执行一行或多行代码。或者，在不满足该条件时执行放在关键字 else 后面的一行或多行代码。if 语句中 else 子句的使用是可选的。

if 语句的基本语法如下所示：

```
if (condition) {
  execute whatever code is contained within the braces if condition is met
}
```

或者添加可选的 else 子句：

```
if (condition) {
  execute whatever code is contained within the braces if condition is met
}
else {
  execute whatever code is contained within the braces if condition is NOT met
}
```

如果 if 或(可选的)else 关键字后面只有一条可执行语句，则可以省略大括号，如下所示，但是通常认为始终使用大括号是良好的编码习惯：

```
// 伪代码
if (condition)
    single statement to execute if true;
else

    single statement to execute if false;
```

作为布尔表达式简单形式的单个布尔变量自然可以用作 if 语句的逻辑表达式/条件。例如，下面编写的代码是完全可以接受的：

```
// 使用这个 bool 变量作为标识，当特定的操作完成时将该标识设置为 true。
bool finished;

// 初始化为 false。
finished = false;

// 插入的代码，其中标识可能设置为 true…细节已省略。

// 测试标识。下一行等同于：if (finished == true) {
if (finished) {
    Console.WriteLine("we are finished");
}
```

在该示例中，逻辑表达式作为 if 语句的条件，该条件响应“是否 finished”或“finished 是否等于 true”。

！运算符可用于否定逻辑表达式。在该表达式为 false 时执行与 if 语句关联的代码块：

```
// 等同于：if (finished == false)
if (!finished) {
// 如果 finished 变量被设置为 false，这段代码就会执行。
Console.WriteLine("we are not finished");
}
```

在该示例中，逻辑表达式作为 if 语句的条件，该条件响应“是否未 finished”或“finished 是否等于 false”。

在测试是否相等时，记住必须使用两个连续的等于号，而不是一个等于号：

```
// 注意使用两个等于号(==)来测试是否相等。
if (x == 3) {
  y = x;
}
```

注意

刚开始学习 C#的程序员常犯的错误是尝试使用单个等于号测试是否相等，如同下面的示例所示：

```
if (x = 3) {...}
```

在 C#中，if 判断必须基于有效的逻辑表达式；x = 3 不是逻辑表达式，而是赋值表达式。

上面的 if 语句不能在 C#中通过编译，然而却可以在 C 和 C++编程语言中通过编译，这是因为在这些语言中，if 判断基于对表达式求值为整数值 0(等同于 false)或非 0(等同于 true)。

可以使用嵌套的 if-else 构造来测试多个条件。如果使用这种嵌套构造，内部的 if 语句(加上可选的 else 语句)被放置在外部 if 语句的 else 部分中。

两层嵌套的 if-else 构造的基本语法如下所示：

```
if (condition1) {
  // 执行这段代码
}
else {
  if (condition2) {
     // 执行其他的代码
  }
  else {
    // 如果不满足任何条件，则执行这段代码
  }
}
```

if-else 构造的嵌套层数没有任何限制，但是尽量不要嵌套太多层。

上面示例中所示的嵌套 if 语句也可以写为没有嵌套的形式，如下所示：

```
if (condition1) {
    // 执行这段代码
}
else if (condition2) {
    // 执行其他的代码
}
else {
     // 如果不满足任何条件，则执行这段代码
}
```

这两种形式在逻辑上相等。

下面的示例使用嵌套的 if-else 构造，基于员工销售额和服务时间来确定员工的奖金额度：

```
using System;
```

```
public class IfDemo
{
  static void Main() {
    double sales = 40000.0;
    int lengthOfService = 12;
    double bonus;

    if (sales > 30000.0 && lengthOfService >= 10) {
      bonus = 2000.0;
    }
    else if (sales > 20000.0) {
      bonus = 1000.0;
    }
    else {
      bonus = 0.0;
    }

    Console.WriteLine("Bonus = " + bonus);
  }
}
```

下面是这段示例代码生成的输出：

```
Bonus = 2000.0
```

1.11.2 switch 语句

switch 语句类似于 if-else 构造，因为它允许根据条件判断执行一行或多行代码。然而，不像 if-else 构造那样对逻辑表达式求值，switch 语句是将整数、字符、枚举或字符串表达式的值与一个或多个 case 标签定义的值进行比较。如果发现匹配，就会执行匹配的 case 标签后面的代码。可以包括可选的 default 标签，以定义在整数、字符、枚举或字符串表达式的值不匹配任何 case 标签时执行的代码。

switch 语句的一般语法如下所示：

```
switch (expression) {
   case value1:
     // 如果表达式匹配 value1，则执行这些代码
     break;
   case value2:
     // 如果表达式匹配 value2，则执行这些代码
     break;
   // 根据需要使用更多 case 标签…
   case valueN:
     // 如果表达式匹配 valueN，则执行这些代码
     break;
   default:
     // 如果不匹配任何 case 标签，则执行这些默认代码
```

```
        break;
}
```

例如：

```
int x;

// x 被赋值，具体细节忽略…

switch (x) {
    case 1:
      // 伪代码。
      do something based on the fact that x equals 1
      break;
    case 2:
      // 伪代码。
      do something based on the fact that x equals 2
      break;
    default:
      // 伪代码。
      do something if x equals something other than 1 or 2
      break;
}
```

注意如下方面：

- 对于 switch 关键字后面圆括号中的表达式，其值的类型必须是 string 或整数值。
- case 标签后面的值必须是常量值(整数常量、字符字面值或字符串字面值)。
- 使用冒号而非分号结束 case 和 default 标签。
- 给定 case 标签后面的语句不需要用括号包围起来，它们更类似于语句列表而非代码块。

与 if 语句不同，在发现匹配并且执行了匹配的 case 标签后面的代码之后，switch 语句不会自动终止。为了退出 switch 语句，就必须使用跳出语句(jump statement)——一般是 break 语句。如果在任一 case 标签中没有包括跳出语句，程序就会继续执行，“贯穿(fall through)”下一个 case 或 default 标签。可以像下面这样利用这一行为：假设多个 case 标签需要执行相同的逻辑，则可以将它们堆叠起来，如下所示：

```
// 假设 x 已经被声明为 int 变量
switch (x) {
    case 1:
    case 2:
    case 3:
        // 如果 x 等于 1、2 或 3，则执行这些代码
        break;
    case 4:
        // 如果 x 等于 4，则执行这些代码
```

```
        break;
}
```

如果需要在一系列互斥的选项中进行选择，switch 语句就非常有用。下面的示例使用 switch 语句，基于 country 变量的值给 capital 变量赋值。如果没有发现匹配，则将 capital 变量赋值为“not in the database”。

```
using System;

public class SwitchDemo
{
  static void Main() {
    string country;
    string capital;
    country = "India";

    // 该 switch 语句将 country 变量的值与 3 个 case 标签的值进行比较。如果没有发现匹
    // 配，则执行 default 标签后面的代码。
    switch (country) {
      case "England":
        capital = "London";
        break;
      case "India":
        capital = "New Delhi";
        break;
      case "USA":
        capital = "Washington D.C.";
        break;
      default:
        capital = "not in the database";
        break;
    }

    Console.WriteLine("The capital of " + country + " is " + capital);
  }
}
```

以上代码示例的输出如下所示：

```
The capital of India is New Delhi
```

1.11.3 for 语句

for 语句是一种编程构造，用于执行给定次数的一条或多条语句。for 语句的一般语法如下所示：

```
for (initializer; condition; iterator) {
```

```
    // condition 为 true 时执行的代码
}
```

for 语句定义了 3 个放在 for 关键字后面的圆括号中的元素，以分号分隔这些元素。

初始化表达式(initializer)一般用于为循环控制变量(loop control variable)赋予初始值。该变量可以被声明为初始化表达式的一部分，也可以在 for 语句之前的代码中声明。例如：

```
// 在 for 语句中声明循环控制变量 i
    for (int i = 0; condition; iterator) {
    // 在 condition 为 true 时执行的代码
}
    // 注意，当 for 循环退出时，i 不再可用。
```

或者：

```
// 在程序中的 for 语句之前声明循环控制变量 i：
    int i;

for (i = 0; condition; iterator) {
    // 在 condition 为 true 时执行的代码
}
// 注意，因为 i 在 for 循环开始之前被声明，所以在 for 循环退出时 i 仍然可用。
```

条件表达式(condition)是逻辑表达式时，一般涉及循环控制变量：

```
for (int i = 0; i < 5; iterator) {
    // 只要 i 小于 5，就执行这些代码
}
```

迭代表达式(iterator)通常用于递增或递减循环控制变量：

```
for (int i = 0; i < 5; i++) {
    // 只要 i 小于 5，就执行这些代码
}
```

注意到在初始化表达式和条件表达式后面使用了分号，而在迭代表达式后面没有使用分号。

下面是 for 循环的操作过程：

- 当程序执行至 for 语句时，第一次执行初始化表达式，并且只执行这一次。
- 然后计算条件表达式，如果该表达式的值为 true，则执行圆括号后面的代码块。
- 该代码块执行完成后，执行迭代表达式。
- 然后重新计算条件表达式，如果条件表达式的值仍然为 true，则再次执行代码块和更新语句。

该过程持续重复，直到条件表达式的值为 false，此时 for 循环退出。

下面是用嵌套 for 语句生成简单乘法表的简单示例。在对应的 for 语句中声明循环控制变量 j 和 k。只要满足对应 for 语句中的条件，就会执行该 for 语句后面的代码块。++运算符用于在每次执行对应的代码块之后递增 j 和 k 的值。

```
using System;

public class ForDemo
{
  static void Main() {
    // 计算简单的乘法表。

    for (int j = 1; j <= 4; j++) {
      for (int k = 1; k <= 4; k++) {
          Console.WriteLine(j + " * " + k + " = " + (j * k));
      }
    }
  }
}
```

下面是该示例的输出：

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

注意

注意 forDemo 示例中字符串连接运算符+的使用；int 变量 j 和 k 的值的字符串表示与字符串字面值“*”和“=”连接起来。

for 语句后面圆括号内的 3 个元素都可省略(但是不能省略两个用于分隔元素的分号)。如果省略初始化表达式，则必须在 for 语句之前声明和初始化循环控制变量：

```
int i = 0;
for (; i < 5; i++) {
// 只要 i 小于 5，就执行一些操作
}
```

如果省略迭代表达式，则必须确保在 for 循环体内显式地更新循环控制变量的值以避免无限循环：

```
for (int i = 0; i < 5; ) {
    // 只要 i 小于 5，就执行一些操作

  // 显式地递增 i。
  i++;
}
```

如果省略条件表达式，则条件表达式的值始终为 true，从而可能导致无限循环：

```
for (;;) {
    // 无限循环!
}
```

注意

在本章后面的“跳出语句”一节中会介绍可以使用跳出语句中断循环。

与其他流程控制结构一样，如果在 for 语句后只指定了一条语句，则可以省略大括号(然而，在任何情况下都使用大括号是一种良好的编程习惯)：

```
for (int i = 0; i < 3; i++)
    sum = sum + i;
```

1.11.4 while 语句

while 语句在功能上类似于 for 语句，两者都用于重复执行相关的代码块。然而，如果在循环最初开始时不知道执行这段代码的次数，则 while 语句就是较好的选择，因为只要满足指定的条件，while 语句就会继续执行。

while 语句的一般语法如下所示：

```
while (condition) {
    // 条件为 true 时执行的代码
}
```

条件表达式可以是简单或复杂的逻辑表达式，该表达式的值为 true 或 false。例如：

```
int x = 1;
int y = 1;

while (x < 20 || y < 10) {
    // 伪代码。
    presumably do something that affects the value of either x or y
}
```

当程序执行至 while 语句时，首先会对条件表达式求值。如果值为 true，则执行条件表达式后面的代码块。当代码块执行结束后，再次对条件表达式求值；如果值仍然为 true，

则重复执行该过程，直到条件表达式的值为 false，此时 while 循环退出。

下面的简单示例举例说明了 while 循环的使用。名为 finished 的 bool 变量最初被设置为 false，该变量用作标识：只要 finished 的值为 false，while 循环后面的代码块就会持续执行。该代码块中的某条语句会最终将 finished 的值设置为 true，此时 while 循环会在下一次重新测试条件时退出。

```
using System;

public class WhileDemo
{
  static void Main() {
    bool finished = false;
    int i = 0;
    while (!finished) {
      Console.WriteLine(i);
      i++;
      if (i == 3)
        finished = true; // 设置标识值
    }
  }
}
```

这段代码的输出如下所示：

```
0
1
2
```

与其他流程控制结构一样，如果在条件表达式后面只指定了一条语句，则可以省略大括号(但是，在任何情况下都使用大括号是一种良好的编程习惯)：

```
while (x < 20)
   x = x * 2;
```

1.11.5 do 语句

使用 while 循环时，在(有条件地)执行其后面的代码块之前对条件表达式求值。因此，如果条件表达式的值一开始就为 false，则循环体中的代码就可能永远不会执行。do 循环类似于 while 循环，不同之处是在对条件表达式求值之前执行代码块，这样就可以保证循环的代码块至少执行一次。

下面是 do 语句的一般语法：

```
do {
  // 要执行的代码
} while (condition);
```

与 while 语句中的情况相同，do 语句的条件表达式是一个求得布尔值的逻辑表达式。在包围条件表达式的括号后面放置分号，用来表明 do 语句的结束。如果知道需要至少执行一次循环体以初始化值，则一般使用 do 循环。

```
bool flag;

do {
    // 无论 flag 的初始设置是什么都执行一些代码，然后根据 flag 的值估算是否应该再次执
    // 行循环代码。可以在循环中将 flag 的值设置为 true 或 false，以指示循环是否应该
    // 再次执行
} while (flag);
```

1.12 跳出语句

在前面讨论的循环和流程控制结构中，其中一些会在满足(或不满足)某种条件时自动退出，另外一些则不会自动退出。C#语言定义了许多跳出语句，用于将执行程序重定向到代码中的其他语句。本节中将要讨论的两种跳出语句是 break 和 continue 语句。另一种跳出语句是用于退出方法的 return 语句，在第 4 章中将讨论该语句。

本章前面已经介绍过将 break 语句与 switch 语句结合使用的情况。break 语句也可以用于突然性地终止 do、for 或 while 循环。在循环执行过程中遇到 break 语句时，循环就会立刻终止，并且程序执行直接转到循环或流程控制结构之后的代码行。

```
// 该循环计划执行 4 次…
for (int j = 1; j <= 4; j++) {
    // …但是当 j 的值为 3 时，下面的 if 测试就会通过，它所控制的 break 语句会执行，并
    // 且跳出循环。
     if (j == 3)
        break;

    //另一方面，如果 if 测试失败，就会跳过 break 语句，输出 j 的值，并且继续执行循环
    Console.WriteLine(j);
}

//如果 break 语句执行，就立刻转到循环后面的代码行。
Console.WriteLine("Loop finished");
```

上述代码片段产生的输出如下所示：

```
1
2
Loop finished
```

与 break 语句不同的是，continue 语句用于退出循环的当前迭代，但是不终止整个循

环执行。continue 语句将程序的执行转回到循环的顶部(for 循环的迭代表达式部分)，并且不会结束当前正在执行的迭代过程：

```
//该循环计划执行 4 次…
for (int j = 1; j <= 4; j++) {
  // …但是当 j 的值为 3 时，下面的 if 测试就会通过，并且“跳回”到 for 语句的 j++部分，
  // j 递增为 4…
  if (j == 3)
      continue;
  // …因此下面的代码行在 j=3 时不会执行，但是在 j=1、2 或 4 时会执行。
  Console.WriteLine(j);
}
Console.WriteLine("Loop finished");
```

这段代码产生的输出如下所示：

```
1
2
4
Loop finished
```

在循环中过多地使用 break 和 continue 语句会导致代码难以跟踪和维护，因此最好只在必须使用这些语句的时候才添加它们。

1.13 代码块和变量作用域

C#(C、C++和 Java 也是如此)是一种块结构语言(block structured language)。本章前面已提及，代码块是用大括号包括起来的形如{…}的 0 行或多行代码。

- 方法声明，如 SimpleProgram 的 Main 方法，定义了一个块。
- 类声明，如 SimpleProgram 类整体，也定义了一个块。
- 如您所见，许多流程控制语句都定义了代码块。

块可以嵌套至任意层深度：

```
public class SimpleProgram
{
    // 在“类”代码块中(一层深度)。
    static void Main() {
      // 在“Main 方法”代码块中(两层深度)。
      int x = 3;
      int y = 4;
      int z = 5;

      if (x > 2) {
        // 在嵌套代码块中(第 3 层)。
```

```
        if (y > 3) {
          // 在更深一层的另一个嵌套代码块中(第 4 层)。
          // (可以不断深入!)
        } // 结束第 4 层代码块。
          // (可以在第 3 层添加其他的代码)
      }   // 第 3 层结束!
      // (可以在第 2 层添加其他的代码)
    } // 第 2 层结束!
    // (可以在第 1 层添加其他的代码)
  } // 第 1 层到此才结束。
```

变量名的作用域被定义为代码的一部分，在该作用域内变量名对编译器保持已定义状态。变量名的作用域一般从最初声明该变量开始，直至在其中声明该变量的代码块的结束(右)大括号为止。只有在声明变量的代码块中，该变量才称为在作用域中(in scope)。一旦程序执行退出某个代码块，在该代码块中声明的任何变量就会超出作用域，不能再为程序所用。

为了演示变量的作用域，接下来编写称为 ScopeDemo 的程序，如下所示。ScopeDemo 声明了 3 个嵌套的代码块：一个是 ScopeDemo 类声明，另一个是 Main 方法，最后一个是 Main 方法体内的 if 语句。

```
public class ScopeDemo
{
  static void Main() {
    double cost = 2.65;

    if (cost < 5.0) {
      double discount = 0.05; // 在 if 代码块内部声明变量
      // 具体细节省略
    }

    // 当 if 代码块退出时，变量 discount 超出作用域，编译器不再能够识别它。如果尝试
    // 在随后的语句中使用该变量，编译器将生成错误。

    double refund = cost * discount; // 这段代码行不能编译，变量 discount 超
                                     // 出作用域
}
```

在上面的示例中，在组成 Main 方法体的代码块内部声明了名为 cost 的变量，而在与 if 语句关联的代码块内部声明了名为 discount 的变量。当 if 语句代码块退出时，discount 变量就超出了作用域。如果在程序的后面尝试访问 discount 变量，如同下面的代码行所示：

```
double refund = cost*discount;
```

编译器就会生成如下的错误：

```
error:: The name 'discount' does not exist in the current context
```

注意，在外层代码块中声明的变量可被该声明语句后面的任何内层代码块访问。例如，在上面的 ScopeDemo 示例中，在声明语句后面的嵌套 if 语句代码块内部可以访问 cost 变量。

1.14 输出到屏幕

大多数应用程序通过在应用程序图形用户界面上显示消息来和用户沟通。然而，有时需要在命令行窗口中以“快而脏”的方式运行验证程序是否正确执行的程序，这时就需要在命令行窗口中显示简单的文本消息(第 13 章中将介绍如何从命令行运行 C#程序)。在第 16 章中讨论如何创建 C#图形用户界面之前，命令行就是程序与“外部世界”沟通的主要途径。

使用如下的语法将文本消息输出到屏幕：

```
Console.WriteLine(expression to be printed);
```

Console.WriteLine 方法可以接受非常复杂的表达式，并且尽其所能将这些表达式变为单个 String 值，然后在屏幕上显示该值。下面是一些示例：

```
Console.WriteLine("Hi!"); // 输出 string 字面值/常量。

string greeting = "Hi!";
Console.WriteLine(greeting); // 输出 string 变量的值。

string s = "foo";
string t = "bar";
Console.WriteLine(s + t); // 使用字符串连接运算符(+)输出"foobar"

int x = 3;
int y = 4;

Console.WriteLine(x); // 将 x 的 int 值转换为 string 值，并且向屏幕输出值 3

Console.WriteLine(x + y); // 计算 x 与 y 相加的和，然后向屏幕输出值 7
```

注意最后一行代码中的加号(+)，该加号被解释为整数加法运算符而非字符串连接运算符，因为它分隔两个都声明为 int 类型的变量。因此，3+4 的和计算为 7，然后输出值 7。然而，在下面的示例中，得到的是不同的(不是我们所需的)结果：

```
Console.WriteLine("The sum of x plus y is: " + x + y);
```

上面一行代码将产生如下的输出：

```
The sum of x plus y is: 34
```

原因何在？

大多数表达式的计算顺序是从左到右，因此第一个加号分隔一个字符串字面值和一个 int 值时，它被解释为字符串连接运算符，从而将 x 的值转换为 string，并且产生临时的 string 值“The sum of x plus y is: 3”。

第二个加号连接临时的 string 值和一个 int 值(y)，因此也被解释为字符串连接运算符，从而将 y 的值转换为 string，并且产生最终的 string 值“The sum of x plus y is: 34”，这就是最终输出到屏幕的内容。

为了输出正确的 x 加 y 之和，必须强制让第二个加号解释为整数加法运算符，方法是使用嵌套的括号包围加法表达式：

```
Console.WriteLine("The sum of x plus y is: " + (x + y));
```

嵌套的括号使最内层的表达式最先被计算；编译器现在看到第二个加号分隔两个 int 值，因此将其作为整数加法运算符。然后，第一个加号分隔一个 string 值和一个 int 值，因此依然被视为字符串连接运算符，从而最终造成这条输出语句在屏幕上显示正确的消息：

```
The sum of x plus y is: 7
```

在编写涉及复杂表达式的代码时，最好多使用括号以使编译器能够清楚地了解我们的意图。额外的括号绝对不会产生副作用！

1.14.1 Write 和 WriteLine 的对比

使用 Console.WriteLine(…)时，将会输出括号中包括的任何表达式，后跟一个行结束符(line terminator)。下面的代码片段：

```
Console.WriteLine("First line.");
Console.WriteLine("Second line.");
Console.WriteLine("Third line.");
```

产生如下输出：

```
First line.
Second line.
Third line.
```

作为对比，下面的语句也会输出括号中包括的任何表达式，但是表达式后面没有行结束符：

```
Console.Write(expression to be printed);
```

将 Write 和 WriteLine 结合起来使用，就可以通过一系列 Write 语句创建单行输出，如同下面的示例所示：

```
Console.Write("C");         // 使用 Write。
Console.Write("SHA");       // 使用 Write。
Console.WriteLine("RP");    // 注意最后一条语句中使用 WriteLine。
```

这个代码片段产生单行输出：

```
CSHARP
```

通过将较长输出语句的内容划分为多个连接的 string，然后用加号将这些 string 连接起来，就可以使程序清单更易于阅读：

```
statement;
another statement;
Console.WriteLine( "Here is an example of how " +
                   "to break up a long print statement " +
                   "with plus signs.");

yet another statement;
```

虽然上面的语句断开为 3 行代码，但是仍然产生单行输出：

```
Here is an example of how to break up a long print statement with plus signs.
```

注意，如果上面的示例中没有加号，代码就不能通过编译，因为字符串字面值不会自动换行。

1.14.2 转义序列

C#定义了一系列转义序列(escape sequence)，在 string 表达式中可以使用这些转义序列表示特殊的字符，如换行和制表符字符。表 1-5 列出了最常用的转义序列。

表 1-5

\n	换行
\b	退格
\t	制表符
\v	垂直制表符
\\	反斜杠
\’	单引号

\"	双引号

可以在传递给 Write 和 WriteLine 方法的表达式中包括一个或多个转义序列。例如，考虑如下的代码片段：

```
Console.WriteLine("Presenting...");
Console.WriteLine("\n...for a limited \"time\" only...\n");
Console.WriteLine("\tBailey the Wonder Dog!");
```

在执行上面的代码时，会显示下面的输出：

```
Presenting…
...for a limited "time" only...

        Bailey the Wonder Dog!
```

在第二行输出之前和之后都用了一个空行，这是因为在第二行语句中插入了额外的\n 转义序列；单词 time 放在双引号中，这是因为在第二行语句中也使用了\"转义序列；通过使用\t 转义序列，第三行输出向右缩进了一个制表符位置。

1.15 C#样式的要素

优秀程序员的特征之一就是他们能够编写易于阅读的代码。您的职业生涯不会总是在山顶独自编写代码，因此需要让您的同事能够使用和修改您的程序。下面的一些原则和约定将帮助您编写清晰而易于阅读的 C#程序。

1.15.1 适当地使用缩进

使 C#程序易于阅读的一种最佳方法是通过适当地使用缩进来清楚地描述语句的层次结构。代码块中的语句应该相对于包围该代码块的开始/结束代码行进行缩进(即相对于带有大括号的行进行缩进)。MSDN 网页上的示例使用了 4 个空格，但是一些程序员使用两个空格，其他程序员则更喜欢使用 3 个空格。本书中的示例都使用两个空格作为缩进约定。

提示

如果使用的是 Visual Studio，可以作为一个 VS 选项配置缩进间距。

为了理解缩进如何让程序易于阅读，可考虑如下两个程序。在第一个程序中没有使用缩进：

```
using System;
```

```
public class StyleDemo
{
static void Main() {
string name = "cheryl";
for (int i = 0; i < 4; i++) {
if (i != 2) {
Console.WriteLine(name + " " + i);
}
}
Console.WriteLine("what's next");
}
}
```

显而易见的是，人们需要非常仔细地阅读该程序才能指出该程序的功能。这是一段不易于阅读的代码。

现在查看使用了适当缩进的相同程序。代码块中的每条语句对相对于包围的代码块缩进了两个空格。现在就可以更方便地理解这段代码的功能。例如，可以清楚地看到 if 语句在 for 语句的代码块内部。如果 if 语句的条件表达式为 true，就会调用 WriteLine 方法。同样显而易见的是，最后一个 WriteLine 方法调用位于 for 循环的外部。虽然这两个版本的程序在执行时会产生相同的结果，但是第二个版本更易于阅读。

```
using System;

public class StyleDemo
{
  static void Main() {
    string name = "Cheryl";
    for (int i = 0; i < 4; i++) {
      if (i != 2) {
        Console.WriteLine(name + " " + i);
      }
    }
    Console.WriteLine("What's next");
  }
}
```

这段代码的输出如下所示：

```
Cheryl 0
Cheryl 1
Cheryl 3
What's next
```

不正确的缩进会使程序难以阅读，因此难以调试——如果出现因为括号不对称而产生编译错误，这种错误消息通常出现在程序的后面，而不是出现在发生问题的位置。

例如，如下的程序在第 11 行中遗漏了起始大括号，但是编译器直到第 25 行才报告该错误！

```
using System;
public class Indent2
{
  static void Main() {
    int x = 2;
    int y = 3;
    int z = 1;

    if (x >= 0) {
      if (y > x) {
        if (y > 2) // 在第 11 行中遗漏了起始大括号，但是…
          Console.WriteLine("A");
          z = x + y;
        }
        else {
          Console.WriteLine("B");
          z = x - y;
        }
      }
      else {
        Console.WriteLine("C");
        z = y - x;
      }
    }
    else Console.WriteLine("D"); // 编译器到第 25 行才报告错误
  }
}
```

在这种情况下，编译器生成的错误消息相当难以理解，它指出第 25 行存在问题，这并不能帮助我们实际地定位第 11 行中所出现的真正问题：

```
IndentDemo.cs (25,5) error:
Invalid token 'else' in class, struct, or interface member declaration.
```

然而，如果已经执行了适当的缩进，则可以较为容易地找到遗漏的大括号。

有时会存在多层嵌套缩进或者单行语句过长的情况，在编译器中查看或打印出来时就会产生“换行”：

```
while (a < b) {
    while (c > d) {
      for (int j = 0; j < 29; j++) {
        x = y + z + a + b - 125*
(c * (d / e) + f) - g + h + j - l - m - n + o +
p * q / r + s;
```

```
      }
    }
}
```

为了避免发生这种情况，最好在空格或标点符号处断开存在问题的行：

```
while (a < b) {
  while (c > d) {
    for (int j = 0; j < 29; j++) {
      // 推荐这样换行。
      x = y + z + a + b - 125*(c * (d / e) + f) - g +
        h + j - l - m - n + o + p * q / r + s;
    }
  }
}
```

1.15.2 明智地使用注释

使代码更易于阅读的另一个重要特性是较多地使用有意义的注释。在编写代码时始终需要记住的是，您知道自己在尝试做什么，但是尝试阅读您的代码的其他人可能并不能理解(如果长时间没有查看自己编写的代码，我们有时甚至需要提醒自己为什么要这样做)。

如果某段代码可能引起疑惑，就需要添加注释：

- 在注释中包括充分的详情以清楚地阐明您的意图。
- 确保注释有价值，而不要画蛇添足。下面的注释就完全没有用，因为它纯粹是多余的注释：

```
//求声明 x 为整数，并且为其赋初始值 3。
int x = 3;
```

- 将注释缩进到与应用该注释的代码块或语句相同的缩进层次

为了通过示例说明注释可以使代码易于阅读的重要性，接下来回顾本章前面的一个示例：

```
using System;

public class IfDemo
{
  static void Main() {
    double sales = 40000.0;
    int lengthOfService = 12;
    double bonus;

    if (sales > 30000.0 && lengthOfService >= 10) {
      bonus = 2000.0;
```

```
    }
    else {
      if (sales > 20000.0) {
        bonus = 1000.0;
      }
      else {
        bonus = 0.0;
      }
    }

    Console.WriteLine("Bonus = " + bonus);
  }
}
```

因为缺少注释，尝试阅读这段代码的人可能很难理解该程序要实现的业务逻辑，现在查看相同的程序，但是其中添加了清晰的、具有描述性的注释：

```
using System;

// 该程序计算员工的奖金额度。
//
// Jacquie Barker 和 Grant Palmer 编写于 2008 年 3 月 5 日。
public class IfDemo
{
    static void Main() {
      // 季度销售额(以美元为单位)。
      double sales = 40000.0;

      // 在职时间(以月为单位)。
      int lengthOfService = 12;

      // 奖励的奖金总额(以美元为单位)。
      double bonus;

      // 如果员工销售额超过 3 万美元并且员工在公司的在职时间超过 10 个月，则奖励两千
      // 美元。
      if (sales > 30000.0 && lengthOfService >= 10) {
          bonus = 2000.0;
      }
      else {
        // 本季度销售额超过两万美元的员工，无论其在职时间多长，均奖励一千美元。
        if (sales > 20000.0) {
          bonus = 1000.0;
        }
        // 销售额在两万美元以下的员工没有奖金。
        else {
          bonus = 0.0;
        }
```

```
        }

        Console.WriteLine("Bonus = " + bonus);
    }
}
```

程序现在变得更易于理解，因为添加的注释解释了代码每一部分计划完成的任务。

1.15.3 大括号的放置

对于使用大括号{…}来表示代码块开始/结束的块结构语言(如 C、C++、Java 和 C#)，有两种关于如何放置代码块左/开始大括号的编码样式。

第一种样式是将左大括号放在开始代码块的代码行末尾，而匹配的右/结束大括号独自占用一行：

```
public class Test { // 左大括号与类声明位于同一行

    static void Main() { // 方法头也是如此
    for (int i = 0; i < 3; i++) { // 控制流程块依然如此
      Console.WriteLine(i);

    // 每个结束大括号独自占用一行:
    }
  }
}
```

另一种开始大括号放置样式是每个开始大括号独自占用一行：

```
public class Test
{
    static void Main()
    {
        for (int i = 0; i < 3; i++)
      {
        Console.WriteLine("i");
      }
    }
}
```

还有一种可能性是混合使用这两种样式：第二种样式(大括号位于独立的行中)用于类声明，而第一种样式(大括号位于与初始代码行相同的行中)用于其他任何场合：

```
// 类声明的左大括号位于自己的行中:
public class Test
{
    // 但是在其他所有场合中，左大括号位于与初始代码行相同的行中。
    static void Main() {
```

```
        for (int i = 0; i < 3; i++) {
          Console.WriteLine(i);

        // 每个结束大括号独自占用一行:
        }
    }
}
```

不存在绝对正确或错误的样式，因为编译器不会特别要求使用某种样式。然而，在代码中保持一致性是良好的习惯，因此应该选择一种大括号放置样式并坚持使用该样式。

无论遵循何种样式，都必须将代码块的结束大括号与代码块中的第一行代码缩进相同的空格数，从而使它们并排显示，本章前面对此进行过讨论。

1.15.4 自说明的变量名

与缩进和注释一样，选择变量名时的目标就是使程序易于阅读，因此尽可能选择自说明的变量名。除了循环控制变量之外，应避免使用单个字母作为变量名。应该尽量少用缩写，除非是经常使用并且开发人员熟知的缩写。考虑如下的变量声明：

```
int grd;
```

变量名 grd 应该表示什么并不非常清楚。该变量是代表网格(grid)、等级(grade)还是葫芦(gourd)？更好的方法是拼写出整个单词：

```
int grade;
```

另一种极端情况是变量名过长，如同下面的示例所示，尝试阅读这段代码清单的人都会感到不知所措：

```
double averageThirdQuarterReturnOnInvestment;
```

减少变量名长度的同时保持其描述性有时是一项具有挑战性的任务，但是确实应该尝试将变量名长度保持在合理的范围内。

注意

在本书后面部分介绍方法和类等其他 OO 构件时，还会讨论它们的命名约定问题。

.NET Framework 提供了一系列命名指导原则，以提倡统一的 C#程序样式。如果您希望阅读关于 C#命名约定的完整详情，可以在如下站点中查找这些命名指导原则：http://msdn.microsoft.com/en-us/library/ms229045.aspx。

1.16 本章小结

本章中讨论了 C#的一些优点，包括如下：

- C#天生就是 OOPL，它借鉴了许多语言，并且青出于蓝。
- C#被设计为纯粹的面向对象语言。
- C#是 Microsoft .NET Framework 的一部分，因此从中获益良多。
- 可以从 MSDN 网站上免费下载 C#。

注意

当然，本章中并没有完整地介绍 C#的所有优点。在随后的章节中将介绍使 C#成为功能强大的 OOPL 的其他许多特性。

除了研究 C#的一些优点之外，本章也介绍了 C#语法的一些基本元素。特别地，本章中介绍了如下内容：

- 详细分析了一个简单的 C#程序
- 讨论预定义的简单类型和 string 类型
- 研究如何声明和初始化值类型变量
- 介绍如何将一种类型的值强制转换为另一种类型
- 讨论了算术表达式、赋值表达式和逻辑表达式
- 介绍了循环和其他在 C#中可用的流程控制结构
- 研究了如何定义代码块以及变量作用域的概念
- 学习了如何使用 Write 和 WriteLine 方法向控制台输出文本消息
- 讨论了良好 C#编程样式的一些要素

关于 C#还有很多需要学习的内容——您需要了解更多知识才能构建本书第III部分中的 SRS 应用程序——但是首先需要研究一些基本的对象概念。因此，接下来学习第 2 章！

1.17 练习

(1) 研究 Microsoft 的 C#语言之旅网站，其网址为：http://msdn.microsoft.com/en-us/library/ 67ef8sbd.aspx。

记录本章中没有提及的 C#特性的优点。

(2) 访问 Microsoft .NET Framework 主页，其地址为：http://msdn.microsoft.com/en-us/library/w0x726c2.aspx

记住，C#可以使用.NET Framework 提供的所有库和其他功能。

(3) 使用 for 循环和 continue 语句创建一段代码，将 1~10 之间的偶数写入控制台。

(4) 使用您所知道的代码块定义技术和适当的缩进技术，使如下这段代码更易于阅读：

```
int count = 0;
for (int j = 0; j < 2; j++) {
count = j;
for (int k = 0; k < 3; k++)
count++;
Console.WriteLine("count = " + count);
}
```

(5) 将目前为止所学习的 C#相关知识与您所熟悉的另一种编程语言进行比较，两种语言的相同点是什么？不同点又是什么？

(6) 给定如下这些初始变量声明和赋值语句：

```
int a = 1;
int b = 1;
int c = 1;
```

计算下面的表达式：

```
((((c++ + --a) * b) != 2) && true)
```