第 2 章 C#编程概述

2

本章内容

一个简单的C#程序

标识符

关键字

Main：程序的起始点

空白

语句

从程序中输出文本

注释：代码的注解

2.1 一个简单的 C#程序

本章将为学习C#打基础。因为本书中会广泛地使用代码示例，所以我们先来看看C#程序的样子，还有它的不同部分代表什么意思。

我们从一个简单程序开始，逐个解释它的各组成部分。这里将会介绍一系列主题，从C#程序

的结构到产生屏幕输出程序的方法。

有这些源代码作为初步铺垫，我就可以在余下的文字中自由地使用代码示例了。因此，与后面的章节详细阐述一两个主题不同，本章将接触很多主题并只给出最简单的解释。

让我们先观察一个简单的C#程序。完整的源程序在图2-1上面的阴影区域中。如图所示，代码包含在一个名称为SimpleProgram.cs的文本文件里。当你阅读它时，不要担心能否理解所有的细节。表2-1对代码进行了逐行描述。图中左下角的阴影区域展示了程序的输出结果，右半边是程序各部分的图形化描述。

当代码被编译执行时，它在屏幕的一个窗口中显示字符串“ Hi there!”。

第5行包含两个相邻的斜杠。这两个字符以及这一行中它们之后的所有内容都会被编译器忽略。这叫做单行注释。

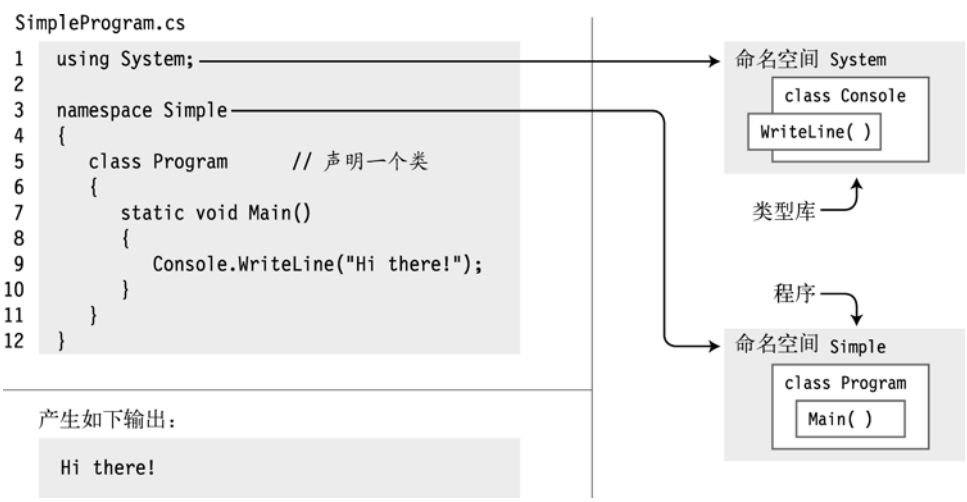


图2-1 SimpleProgram程序

表2-1 SimpleProgram程序的逐行描述

行 号	描 述
行1	告诉编译器这个程序使用System命名空间的类型
行3	声明一个新命名空间，名称为Simple • 新命名空间从第4行的左大括号开始一直延伸到第12行与之对应的右大括号 • 在这部分里声明的任何类型都是该命名空间的成员
行5	声明一个新的类类型，名称为Program • 任何在第6行和第11行的两个大括号中间声明的成员都是组成这个类的成员
行7	声明一个名称为Main的方法作为类Program的成员 • 在这个程序中，Main是Program类的唯一成员 • Main是一个特殊函数，编译器用它作为程序的起始点
行9	只包含一条单独的、简单的语句，这一行组成了Main的方法体 • 简单语句以一个分号结束 • 这条语句使用命名空间System中的一个名称为Console的类将消息输出到屏幕窗口 • 没有第1行的using语句，编译器就不会知道在哪里寻找类Console

SimpleProgram 的补充说明

C#程序由一个或多个类型声明组成。本书的大部分内容都是用来解释可以在程序中创建和使用的不同类型。程序中的类型可以以任何顺序声明。在SimpleProgram中，只声明了class类型。

命名空间是与某个名称相关联的一组类型声明。SimpleProgram使用两个命名空间。它创建了一个名称为Simple的新命名空间，并在其中声明了其类型（类program），还使用了System命名空间中定义的Console类。

要编译这个程序，可以使用Visual Studio或命令行编译器。如果使用命令行编译器，最简单的形式是在命名窗口使用下面的命令：

```
csc SimpleProgram.cs
```

在这条命令中，csc是命令行编译器的名称，SimpleProgram.cs是源文件的名称。CSC是指“ C-Sharp编译器”。

2.2 标识符

标识符是一种字符串，用来命名如变量、方法、参数和许多后面将要阐述的其他程序结构。

可以通过把有意义的词连接成一个单独的描述性名称来创建自文档化（self-documenting）的标识符，可以使用大写和小写字母（如CardDeck、PlayersHand、FirstName和SocialSecurityNum）。某些字符能否在标识符中特定的位置出现是有规定的，这些规则如图2-2所示。

字母和下划线（a-z、A-Z和_）可以用在任何位置。

数字不能放在首位，但可以放在其他的任何地方。

@字符只能放在标识符的首位。虽然允许使用，但不推荐将@作为常用字符。

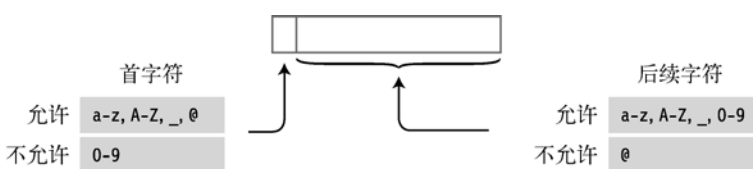


图2-2 标识符中允许使用的字符

标识符区分大小写。例如，变量名myVar和MyVar是不同的标识符。

举个例子，在下面的代码片段中，变量的声明都是有效的，并声明了不同的整型变量。但使用如此相似的名称会使代码更易出错并更难调试，后续需要调试代码的人会很不爽。

```
// 语法上有效，但非常混乱
int totalCycleCount;
```

```
int TotalCycleCount;
int TotalcycleCount;
```

我将在第7章介绍推荐的C#命名约定。

2.3　关键字

关键字是用来定义C#语言的字符串记号。表2-2列出了完整的C#关键字表。

关于关键字，一些应该知道的重要内容如下。

关键字不能被用做变量名或任何其他形式的标识符，除非以@字符开始。

所有C#关键字全部都由小写字母组成（但是.NET类型名使用Pascal大小写约定）。

表2-2　C#关键字

abstract	const	extern	int	out	short	typeof
as	continue	false	interface	override	sizeof	uint
base	decimal	finally	internal	params	stackallo c	ulong
bool	default	fixed	is	private	static	unchecked
break	delegate	float	lock	protected	string	unsafe
Byte	do	for	long	public	struct	ushort
case	double	foreach	namespace	readonly	switch	using
catch	else	goto	new	ref	this	virtual
char	enum	if	null	return	throw	void
checked	event	implicit	object	sbyte	true	volatile
class	explicit	in	operator	sealed	try	while

上下文关键字是仅在特定的语言结构中充当关键字的标识符。在那些位置，它们有特别的含意。两者的区别是，关键字不能被用做标识符，而上下文关键字可以在代码的其他部分被用做标识符。上下文关键字如表2-3所示。

表2-3　C#的上下文关键字

add	ascending	async	await	by	descendin g	dynamic
equals	from	get	global	group	in	into
join	let	on	orderby	partial	remove	select
set	value	var	where	yield		

2.4 Main：程序的起始点

每个C#程序必须有一个类带有Main方法（函数）。在先前所示的SimpleProgram程序中，它被声明在Program类中。

每个C#程序的可执行起始点在Main中的第一条指令。

Main必须首字母大写。

Main的最简单形式如下：

```
static void Main( )
{
   更多语句
}
```

2.5 空白

程序中的空白指的是没有可视化输出的字符。程序员在源代码中使用的空白将被编译器忽略，但使代码更清晰易读。空白字符包括：

空格（Space）；

制表符（Tab）；

换行符；

回车符。

例如，下面的代码段会被编译器完全相同地对待而不管它们表面上的区别。

```
// 很好的格式
Main()
{
   Console.WriteLine("Hi, there!");
}
```

```
// 连在一起
Main(){Console.WriteLine("Hi, there!");}
```

2.6 语句

C#的语句和C、C++的语句非常相似。本节将介绍语句的常用形式，详细的语句形式将在第9章介绍。

语句是描述一个类型或告诉程序去执行某个动作的一条源代码指令。

简单语句以一个分号结束。

例如，下面的代码是一个由两条简单语句组成的序列。第一条语句定义了一个名称为var1的整型变量，并初始化它的值为5。第二条语句将变量var1的值打印到屏幕窗口。

```
int var1 = 5;
System.Console.WriteLine("The value of var1 is {0}", var1);
```

块

块是一个由成对大括号包围的0条或多条语句序列，它在语法上相当于一条语句。

可以使用之前示例中的两条语句创建一个块。用大括号把语句包围起来，如下面的代码所示。

```
{
   int var1 = 5;
   System.Console.WriteLine("The value of var1 is {0}", var1);
}
```

关于块，一些应该知道的重要内容如下。

语法上只需要一条语句，而你需要执行的动作无法用一条简单的语句表达的情况下，考虑使用块。

有些特定的程序结构只能使用块。在这些结构中，不能用简单语句替代块。

虽然简单语句以分号结束，但块后面不跟分号。（实际上，由于被解析为一条空语句，所以编译器允许这样，但这不是好的风格。）

```
{           以分号结束
               ↓                                                    以分号结束
   int var2 = 5;                                                        ↓
   System.Console.WriteLine("The value of var1 is {0}", var1);
}
 ↑   不以分号结束
```

2.7 从程序中输出文本

控制台窗口是一种简单的命令提示窗口，允许程序显示文本并从键盘接受输入。BCL提供一个名称为Console的类（在System命名空间中），该类包含了输入和输出数据到控制台窗口的方法。

2.7.1 Write

Write是Console类的成员，它把一个文本字符串发送到程序的控制台窗口。最简单的情况下，Write将文本的字符串字面量发送到窗口，字符串必须使用双引号括起来。

下面这行代码展示了一个使用Write成员的示例：

```
Console.Write("This is trivial text.");
                        ↑
                   输出字符串
```

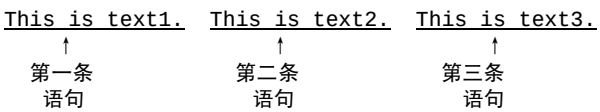
这段代码在控制台窗口产生如下输出：

```
This is trivial text.
```

另外一个示例是下面的代码，发送了3个文本字符串到程序的控制台窗口：

```
System.Console.Write ("This is text1. ");
System.Console.Write ("This is text2. ");
System.Console.Write ("This is text3. ");
```

这段代码产生的输出如下，注意，Write没有在字符串后面添加换行符，所以三条语句都输出到同一行。



2.7.2 WriteLine

WriteLine是Console的另外一个成员，它和Write实现相同的功能，但会在每个输出字符串的结尾添加一个换行符。

例如，如果使用先前的代码，用WriteLine替换掉Write，输出就会分隔在多行：

```
System.Console.WriteLine("This is text1.");
System.Console.WriteLine("This is text2.");
System.Console.WriteLine("This is text3.");
```

这段代码在控制台窗口产生如下输出：

```
This is text1.
This is text2.
This is text3.
```

2.7.3 格式字符串

Write语句和WriteLine语句的常规形式中可以有一个以上的参数。

如果不只一个参数，参数间用逗号分隔。

第一个参数必须总是字符串，称为格式字符串。格式字符串可以包含替代标记。

替代标记在格式字符串中标出位置，在输出串中该位置将用一个值来替代。

替代标记由一个整数及括住它的一对大括号组成，其中整数就是替换值的数字位置。跟着格式字符串的参数称为替换值，这些替换值从0开始编号。

语法如下：

```
Console.WriteLine(格式字符串（含替代标记），替换值0，替换值1，替换值2，……);
```

例如，下面的语句有两个替代标记，编号0和1；以及两个替换值，它们的值分别是3和6。

```
                                                  替代标记
                                                  ↓     ↓
Console.WriteLine("Two sample integers are {0} and {1}.", 3, 6);
                   ↑                                    ↑
                  格式字符串                           替换值
```

这段代码在屏幕上产生如下输出：

```
Two sample integers are 3 and 6.
```

2.7.4 多重标记和值

在C#中，可以使用任意数量的替代标记和任意数量的值。

值可以以任何顺序使用。

值可以在格式字符串中替换任意次。

例如，下面的语句使用了3个标记但只有两个值。请注意，值1被用在了值0之前，而且值1被使用了两次。

```
Console.WriteLine("Three integers are {1}, {0} and {1}.", 3, 6);
```

这段代码在屏幕上显示如下：

```
Three integers are 6, 3 and 6.
```

标记不能试图引用超出替换值列表长度以外位置的值。如果引用了，不会产生编译错误，但会产生运行时错误（称为异常）。

例如，在下面的语句中有两个替换值，在位置0和1。而第二个标记引用了位置2，位置2并不存在。这将会产生一个运行时错误。

```
                                                    位置0 位置1
                                                      ↓   ↓
Console.WriteLine("Two integers are {0} and {2}.", 3   6);    //错误
                                                  ↑
                                          位置2的值不存在
```

2.7.5 格式化数字字符串

贯穿本书的示例代码将会使用WriteLine方法来显示值。每次，我们都使用由大括号包围整数组成的简单替代标记形式。

然而在很多时候，我们更希望以更合适的格式而不是一个简单的数字来呈现文本字符串的输出。例如，把值作为货币或者某个小数位数的定点值来显示。这些都可以通过格式化字符串来实现。

例如，下面的代码由两条打印值500的语句组成。第一行没有使用任何其他格式化来打印数字，而第二行的格式化字符串指定了数字应该被格式化成货币。

```
Console.WriteLine("The value: {0}."  , 500);        // 输出数字
Console.WriteLine("The value: {0:C}.", 500);        // 格式为货币
                                ↑
                          格式化为货币
```

这段代码产生了如下的输出：

```
The value: 500.
```

```
The value: $500.00.
```


两条语句的不同之处在于，格式项以格式说明符形式包括了额外的信息。大括号内的格式说明符的语法由3个字段组成：索引号、对齐说明符和格式字段（format field）。语法如图2-3所示。

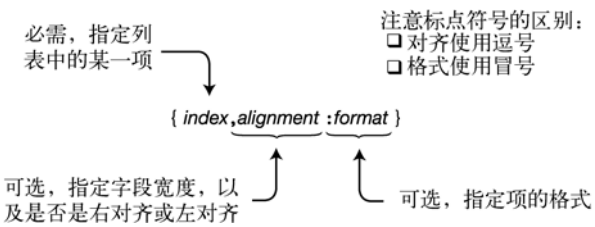


图2-3 格式说明符的语法

格式说明符的第一项是索引号。如你所知，索引指定了之后的格式化字符串应该格式化列表中的哪一项。索引号是必需的，并且列表项的数字必须从0开始。

1. 对齐说明符

对齐说明符表示了字段中字符的最小宽度。对齐说明符有如下特性。

对齐说明符是可选的，并且使用逗号来和索引号分离。

它由一个正整数或负整数组成。

整数表示了字段使用字符的最少数量。

符号表示了右对齐或左对齐。正数表示右对齐，负数表示左对齐。

```
    索引——使用列表中的第0项
                ↓
Console.WriteLine("{0, 10}", 500);
                    ↑
对齐说明符——在10个字符的字段中右对齐
```

例如，如下格式化int型变量myInt的值的代码显示了两个格式项。在第一个示例中，myInt的值以在10个字符的字符串中右对齐的形式进行显示；第二个示例中则是左对齐。格式项放在两个竖杠中间，这样在输出中就能看到它们的左右边界。

```
int myInt = 500;
Console.WriteLine("|{0, 10}|", myInt);                    // 右对齐
Console.WriteLine("|{0,-10}|", myInt);                    // 左对齐
```

这段代码产生了如下的输出，在两个竖杠的中间有10个字符：

```
|       500|
|500       |
```

值的实际表示可能会比对齐说明符指定的字符数多一些或少一些：

如果要表示的字符数比对齐说明符中指定的字符数少，那么其余字符会使用空格填充；

如果要表示的字符数多于指定的字符数，对齐说明符会被忽略，并且使用所需的字符进行表示。

2. 格式字段

格式字段指定了数字应该以哪种形式表示。例如，应该被当做货币、十进制数字、十六进制数字还是定点符号来表示？

格式字段有三部分，如图2-4所示。

冒号后必须紧跟着格式说明符，中间不能有空格。

格式说明符是一个字母字符，是9个内置字符格式之一。字符可以是大写或小写形式。大小写对于某些说明符来说比较重要，而对于另外一些说明符来说则不重要。

精度说明符是可选的，由1～2位数字组成。它的实际意义取决于格式说明符。

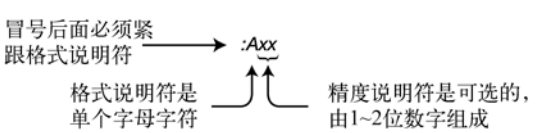


图2-4　标准的格式字段字符串

如下代码是格式字符串组件语法的一个示例：

```
          索引——使用列表中的第0项
                        ↓
Console.WriteLine("{0:F4}", 12.345678);
                        ↑
              格式组件——4位小数的定点数
```

如下代码给出了不同格式字符串的一些示例：

```
double myDouble = 12.345678;
Console.WriteLine("{0,-10:G} -- General",                       myDouble);
Console.WriteLine("{0,-10} -- Default, same as General",        myDouble);
Console.WriteLine("{0,-10:F4} -- Fixed Point, 4 dec places",    myDouble);
Console.WriteLine("{0,-10:C} -- Currency",                      myDouble);
Console.WriteLine("{0,-10:E3} -- Sci. Notation, 3 dec places", myDouble);
Console.WriteLine("{0,-10:x} -- Hexadecimal integer",           1194719 );
```

这段代码产生了如下的输出：

```
12.345678  -- General
12.345678  -- Default, same as General
12.3457    -- Fixed Point, 4 dec places
$12.35     -- Currency
1.235E+001 -- Sci. Notation, 3 dec places
123adf     -- Hexadecimal integer
```

3. 标准数字格式说明符

表2-4总结了9种标准数字格式说明符。第一列在说明符名后列出了说明符字符。如果说明符字符根据它们的大小写会有不同的输出，就会标注为区分大小写。

表2-4 标准数字格式说明符

名字和字符	意　义
货币	使用货币符号把值格式化为货币，货币符号取决于程序所在PC的区域设置
C、c	精度说明符：小数位数
	示例：Console.WriteLine("{0:C}",12.5);
	输出：$12.50
十进制数	十进制数字字符串，需要的情况下有负数符号。只能和整数类型配合使用
D、d	精度说明符：输出字符串中的最少位数。如果实际数字的位数更少，则在左边以0填充
	示例：Console.WriteLine("{0:D4}",12);

	输出：0012
定点	带有小数点的十进制数字字符串。如果需要也可以有负数符号
F、f	精度说明符：小数的位数

（续）

名字和字符	意 义
定点	示例：Console.WriteLine("{0:F4}",12.3456789);
F、f	输出：12.3457
常规	在没有指定说明符的情况下，会根据值转换为定点或科学记数法表示的紧凑形式
G、g	精度说明符：根据值
	示例：Console.WriteLine("{0,G4}",12.345678);
	输出：12.35
十六进制数	十六进制数字的字符串。十六进制数字A～F会匹配说明符的大小写形式
X、x	精度说明符：输出字符串中的最少位数。如果实际数的位数更少，则在左边以0填充
区分大小写	示例：Console.WriteLine("{0:x}",180026);
	输出：2bf3a
数字	和定点表示法相似，但是在每三个数字的一组中间有逗号或空格分隔符。从小数点开始往左数。使用逗号还是空格分隔符取决于程序所在PC的区域设置
N、n	精度说明符：小数的位数
	示例：Console.WriteLine("{0:N2}",12345678.54321);
	输出：12,345,678.54
百分比	表示百分比的字符串。数字会乘以100
P、p	精度说明符：小数的位数
	示例：Console.WriteLine("{0:P2}",0.1221897);
	输出：12.22%
往返过程	保证输出字符串后如果使用Parse方法将字符串转化成数字，那么该值和原始值一样。Parse方法将在第25章描述
R、r	精度说明符：忽略
	示例：Console.WriteLine("{0:R}",1234.21897);
	输出：1234.21897
科学记数法	具有尾数和指数的科学记数法。指数前面加字母E。E的大小写和说明符一致
E、e	精度说明符：小数的位数

区分大小写	示例：Console.WriteLine("{0:e4}",12.3456789); 输出：1.2346e+001

2.8 注释：为代码添加注解

你已经见过单行注释了，所以这里将讨论第二种行内注释——带分隔符的注释，并提及第三种类型，称为文档注释。

带分隔符的注释有两个字符的开始标记（/*）和两个字符的结束标记（*/）。

标记对之间的文本会被编译器忽略。

带分隔符的注释可以跨任意多行。

例如，下面的代码展示了一个跨多行的带分隔符的注释。

```
 ↓ 跨多行注释的开始
/*
    这段文本将被编译器忽略
    带分隔符的注释与单行注释不同
    带分隔符的注释可以跨越多行
*/
 ↑注释结束
```

带分隔符的注释还可以只包括行的一部分。例如，下面的语句展示了行中间注释出的文本。该结果就是只声明了一个变量var2。

```
  注释开始
     ↓
int /*var 1,*/ var2;
            ↑
         注释结束
```

说明 C#中的单行注释和带分隔符的注释与C和C++中的相同。

2.8.1 关于注释的补充

关于注释，有其他几点重要内容需要知道。

嵌套带分隔符的注释是不允许的，一次只能有一个注释起作用。如果你打算嵌入注释，首先开始的注释直到它的范围结束都有效。

注释类型的范围如下。

对于单行注释，一直到行结束都有效。

对于带分隔符的注释，直至遇到第一个结束分隔符都有效。

下面的注释方式是不正确的：

```
↓创建注释
/*尝试嵌套注释
   /*   ← 它将被忽略，因为它在一个注释的内部
       内部注释
   */ ← 注释结束，因为它是遇到的第一个结束分隔符
*/    ← 产生语法错误，因为没有开始分隔符

↓创建注释     ↓它将被忽略，因为它在一个注释的内部
//单行注释    /*嵌套注释？
          */   ←产生语法错误，因为没有开始分隔符
```

2.8.2 文档注释

C#还提供第三种注释类型：文档注释。文档注释包含XML文本，可以用于产生程序文档。这种类型的注释看起来像单行注释，但它们有三个斜杠而不是两个。文档注释会在第25章阐述。

下面的代码展示了文档注释的形式：

```
/// <summary>
/// This class does...
/// </summary>
class Program
{
   ...
```

2.8.3 注释类型总结

行内注释是被编译器忽略但被包含在代码中以说明代码的文本片段。程序员在他们的代码中插入注释以解释和文档化代码。表2-5总结了注释的类型。

表2-5 注释类型

类 型	开 始	结 束	描 述
单行注释	//		从开始标记到该行行尾的文本被编译器忽略
带分隔符的注释	/*	*/	从开始标记到结束标记之间的文本被编译器忽略
文档注释	///		这种类型的注释包含XML文本，可以使用工具生成程序文档。详细内容参见第25章