

.NET 精选系列

SAMS

21天 学通C#

〔美〕Bradley L. Jones 著
信达工作室 译

Microsoft
net

美术编辑：胡平利

21天 学通 C#

ISBN 7-115-10199-X



9 787115 101990 >

ISBN7-115-10199-X/TP·2831

定价：60.00 元

人民邮电出版社

7P312C
Q78

.NET 精选系列

21 天学通 C#

[美] Bradley L. Jones 著

信达工作室 译



A0989646

人民邮电出版社

图书在版编目 (CIP) 数据

21 天学通 C# / (美) 琼斯 (Jones, B.L.) 著; 信达工作室译.

—北京: 人民邮电出版社, 2002.3

(.NET 精选系列)

ISBN 7-115-10199-X

I. 2... II. ①琼... ②信... III. 语言—程序设计

IV. TP312

中国版本图书馆 CIP 数据核字 (2002) 第 009826 号

版权声明

Bradley L. Jones: Sams Teach Yourself C# in 21 Days

Copyright © 2002 by Bradley L. Jones.

Authorized translation from the English language edition published by Sams.

All rights reserved.

本书中文简体字版由美国 Sams 出版公司授权人民邮电出版社出版, 未经出版者书面许可, 对本书任何部分不得以任何方式复制或抄袭。

版权所有, 侵权必究。

.NET 精选系列

21 天学通 C#

-
- ◆ 著 [美] Bradley L. Jones
译 信达工作室
责任编辑 李 际
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
读者热线 010-67180876
北京汉魂图文设计有限公司制作
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销
 - ◆ 开本: 787×1092 1/16
印张: 36.5
字数: 963 千字 2002 年 3 月第 1 版
印数: 1-4 000 册 2002 年 3 月北京第 1 次印刷

著作权合同登记 图字: 01-2001-4093 号

ISBN 7-115-10199-X/TP·2831

定价: 60.00 元

本书如有印装质量问题, 请与本社联系 电话: (010) 67129223

作者简介

Bradley L. Jones 是包括 CodeGuru.com、Developer.com 和 Javascripts.com 等在内的众多著名开发人员网站的系统管理员，还是 internet.com 网站的 EarthWeb 频道的执行编辑。Bradley 使用 C# 的时间比大多数开发人员都长，因为他被邀请在 beta 版发布之前对其进行试用。Bradley 拥有使用 C、C++、PowerBuilder、SQL Server 以及许多其他工具和技术进行开发工作的经验。另外，它还是一位畅销书作者，编写了最早的“21 天”图书——《21 天学通 C 语言》。

前 言

欢迎使用本书,通过本书的名称,您可能猜到了,作者编写本书旨在让您用 21 天的时间学通编程语言 C#。本书分为 21 天课程,其中每天的课程都能在几个小时(或一个晚上)完成。读者在 21 天中每天都花 2~3 个小时,则将轻松地学习完本书。当然,这不一定都是连续的 21 个晚上,也不一定非得在晚上阅读。

其中每天课程都可以在 1~2 小时内阅读完毕,有些可能所需的时间长些,而有些可能短些。如果读者期望仅通过阅读便能学通 C#,将肯定会失望。相反,您应该将一半的时间用于阅读,另一半的时间用于输入其中的代码、完成小测验和练习。这看起来工作量好像很大,但每天课程都可以在一个晚上完成。

其中每天课程都包括小测验和练习,它们是为帮助您巩固对当天内容的理解而设计的。阅读完每天课程的内容后,读者应该知道所有的小测验答案,如果不是这样,则应该复习该课中的一些内容。练习给读者提供了应用所学知识的机会。通常,练习的重点在于理解代码、识别常见的代码错误或根据当天的课程编写代码。

小测验和大部分练习的答案都可以在附录 A 中找到,在参考其中的答案之前,请自己尽可能地解决。

阅读本书时,您还将发现其几个特点。书中包含“提示”、“注意”和“警告”。“提示”提供了很有帮助的建议;“注意”提供了一些您可能感兴趣的信息;而“警告”则提醒您注意一些可能遇到的问题。本书中一个很特别的部分是每一课程最后的“问与答”一节,其中包含您在阅读该课程时可能提出的问题及其答案。这些问题可能涉及到与该课程相关的主题。

对读者的要求

作者对读者做了一些假设。假设读者有 C#编译器和 .NET 运行阶段环境。即使没有这些东西,读者也能阅读本书,但要完全理解其中

的内容将很困难。

作者假设读者是初中级程序员，如果您不是，也将从本书中学到很多东西，但在某些地方，您的进展速度将没有您希望得那么快。

本书不要求读者使用 Visual C#或 Visual Studio.NET 开发环境。许多图书要求读者使用微软公司的这些工具，但本书不要求。读者可以使用微软公司的工具，也可以使用许多其他的工具。阅读本书时，您将对此有更深刻的体会。

网站支持

没有人是十全十美的——特别是作者。加上本书介绍的是一种较新的编程语言，因此您可能会发现书中的一些问题。

本书已经多人审校，除了作者本人的校对外，还有一些人对其做了技术和开发方面的审阅。虽然经过这么多的审阅，但错误仍将难免。为防出现问题，多个网站都提供了本书的勘误表，其中出版社网站的网址为：<http://www.sampublishing.com/>；另外作者也为本书专门创建了一个网站，其网址为 www.TeachYourselfCSharp.com，我将在这里发布勘误表。

源代码

作者认为，学习编程语言的最佳方式是输入代码并运行之。为此，笔者不想在初中级图书中附带光盘。但我也知道，并非每个人都认同这种观点，因此我在网站 www.TeachYourselfCSharp.com 上提供了本书的源代码。

本书是为学习编写的，读者可以使用其中的源代码，对其进行修改、扩展，甚至提供他人。购买本书后，您便有权以任何您认为合适的方式使用其中的代码，只有一种情况例外，那就是不能将其用于其他的 C#教学材料中。

作 者

目 录

第一周课程

第 1 天课程 C#初步	2
1.1 C#是什么	2
1.2 为何使用 C#	2
1.2.1 C#简单	3
1.2.2 C#现代	3
1.2.3 C#是面向对象的	3
1.2.4 C#强大、灵活	3
1.2.5 C#包含的单词为数不多	4
1.2.6 C#是模块化的	4
1.2.7 C#将流行	4
1.3 C#和其他编程语言的比较	5
1.4 编程前的准备工作	5
1.5 程序开发周期	6
1.5.1 创建源代码	6
1.5.2 执行 C#程序	7
1.5.3 编译 C#源代码	8
1.5.4 完成开发周期	8
1.6 第 1 个 C#程序	9
1.6.1 输入并编译 hello.cs	10
1.7 C#程序的类型	12
1.8 总结	12
1.9 问与答	12
1.10 作业	13

1.10.1 小测验	13
1.10.2 练习	14
第2天课程 了解 C#程序	16
2.1 C#应用程序	16
2.1.1 注释	17
2.2 C#应用程序的基本组成部分	20
2.2.1 空白	20
2.2.2 C#关键字	21
2.2.3 字面值	21
2.2.4 标识符	21
2.3 C#应用程序的结构	21
2.3.1 C#表达式和语句	21
2.3.2 空语句	21
2.4 分析程序清单 2.1	22
2.4.1 第1~4行: 注释	22
2.4.2 第5、7、13、17、21和23行: 空白	22
2.4.3 第6行: using 语句	22
2.4.4 第8行: 类声明	22
2.4.5 第9、11、26和27行: 标点字符	22
2.4.6 第10行: Main()	22
2.4.7 第14、15和16行: 声明	22
2.4.8 第20行: 赋值语句	23
2.4.9 第24行和25行: 调用函数	23
2.5 面向对象编程 (OOP)	23
2.5.1 面向对象的概念	23
2.5.2 对象和类	24
2.6 显示基本信息	24
2.6.1 打印其他信息	26
2.7 总结	26
2.8 问与答	27
2.9 作业	27
2.9.1 小测验	27
2.9.2 练习	28
第3天课程 使用变量存储信息	31
3.1 变量	31
3.1.1 变量名	31
3.2 使用变量	32

3.2.1	声明变量	33
3.2.2	给变量赋值	33
3.2.3	初始化变量	34
3.2.4	使用未初始化的变量	35
3.3	了解计算机内存	36
3.4	C#数据类型	37
3.5	数值变量类型	37
3.5.1	整型数据类型	39
3.5.2	浮点数	44
3.5.3	Decimal	45
3.5.4	布尔型	45
3.5.5	检查和不检查	45
3.5.6	简单数据类型	46
3.6	字面值和变量	48
3.6.1	数值型字面值	48
3.6.2	布尔型字面值	49
3.6.3	字符串字面值	49
3.7	常量	49
3.8	引用类型	50
3.9	总结	50
3.10	问与答	51
3.11	作业	52
3.11.1	小测验	52
3.11.2	练习	52
第 4 天课程	使用运算符	54
4.1	运算符的类型	54
4.1.1	单目运算符	54
4.1.2	双目运算符	55
4.1.3	三目运算符	55
4.2	标点符号	55
4.3	基本的赋值运算符	56
4.4	数学/算术运算符	56
4.4.1	加减	56
4.4.2	乘法运算符	57
4.4.3	执行单目数学运算	59
4.5	关系运算符	61
4.5.1	If 语句	62

4.5.2 条件逻辑运算符	63
4.6 逻辑按位运算符	66
4.7 类型运算符	66
4.8 sizeof 运算符	66
4.9 条件运算符	66
4.10 运算符优先级	67
4.10.1 改变优先级次序	68
4.11 转换数据类型	69
4.12 理解操作数提升	70
4.13 给足够勇敢者	70
4.13.1 在内存中存储变量	70
4.13.2 移位运算符	71
4.13.3 逻辑运算符	72
4.14 总结	74
4.15 问与答	75
4.16 作业	75
4.16.1 小测验	75
4.16.2 练习	75
第 5 天课程 控制语句	77
5.1 控制程序流程	77
5.2 使用选择语句	77
5.2.1 再谈 if 语句	77
5.2.2 switch 语句	81
5.2.3 switch 语句的控制类型	84
5.3 使用循环语句	85
5.3.1 while 语句	85
5.3.2 do 语句	88
5.3.3 for 语句	90
5.3.4 foreach 语句	93
5.3.5 再谈 break 和 continue 语句	93
5.4 使用 goto	93
5.4.1 标签语句	94
5.5 程序流程命令的嵌套	95
5.6 总结	95
5.7 问与答	95
5.8 作业	96
5.8.1 小测验	96

5.8.2 练习	96
第6天课程 类	98
6.1 再谈面向对象编程	98
6.1.1 封装	98
6.1.2 多态	99
6.1.3 继承	99
6.1.4 重用	99
6.1.5 对象和类	99
6.2 定义类	99
6.3 类声明	100
6.3.1 类成员	101
6.4 数据成员（字段）	101
6.4.1 访问数据成员	102
6.4.2 使用数据成员	103
6.4.3 将类用作数据成员	104
6.4.4 类型嵌套	106
6.5 静态变量	106
6.6 应用程序类	108
6.7 属性	109
6.8 名称空间	111
6.8.1 嵌套名称空间	113
6.9 总结	113
6.10 问与答	113
6.11 作业	113
6.11.1 小测验	113
6.11.2 练习	114
第7天课程 类方法和成员函数	115
7.1 方法初步	115
7.2 使用方法	115
7.3 包含方法的程序的流程	118
7.4 方法的格式	118
7.4.1 方法头	118
7.4.2 方法的返回数据类型	119
7.4.3 给方法命名	119
7.4.4 方法体	119
7.5 给方法传递值	123
7.5.1 静态方法	125

7.5.2 不同的参数访问属性	125
7.6 类方法的类型	129
7.6.1 属性存取器方法	130
7.6.2 构造函数	130
7.6.3 析构函数	134
7.7 总结	136
7.8 问与答	137
7.9 作业	137
7.9.1 小测验	137
7.9.2 练习	138
第 1 周复习	139
程序 WR01.cs	139
XML 文档	148
代码概览	150
Main 方法	150
GetMenuChoice 方法	150
菜单选项	151
point 类	151
line 类	151
其他类	151
 第二周课程	
第 8 天课程 高级数据存储方式：结构、枚举和数组	153
8.1 结 构	153
8.1.1 结构与类	153
8.1.2 结构成员	154
8.1.3 结构的嵌套	156
8.1.4 结构方法	157
8.1.5 结构的构造函数	158
8.1.6 结构的析构函数	160
8.2 枚举	160
8.2.1 修改枚举的默认值	162
8.2.2 修改枚举的底层类型	164
8.3 使用数组存储数据	165
8.3.1 声明数组	166

8.3.2 多维数组	170
8.3.3 创建包含的数组长度不同的数组	171
8.3.4 检查数组长度和边界	172
8.3.5 在类和结构中使用数组	173
8.3.6 使用 foreach 语句	173
8.4 总结	174
8.5 问与答	175
8.6 作业	175
8.6.1 小测验	175
8.6.2 练习	176
第 9 天课程 关于方法的高级主题	177
9.1 重载方法	177
9.1.1 重载函数	178
9.1.2 构造函数重载	180
9.1.3 理解方法的特征标	183
9.2 使用不同数目的参数	184
9.2.1 使用 params 来指定多种数据类型	186
9.2.2 详谈 params	187
9.2.3 Main 方法和命令参数	188
9.3 作用域	189
9.3.1 局部作用域	189
9.3.2 区分类变量和局部变量	192
9.3.3 类作用域限定符	192
9.4 不能用于创建对象的类	192
9.4.1 私有构造函数	193
9.5 再谈名称空间	195
9.5.1 给名称空间命名	195
9.5.2 声明名称空间	195
9.5.3 using 和名称空间	197
9.6 总结	198
9.7 问与答	198
9.8 作业	199
9.8.1 小测验	199
9.8.2 练习	199
第 10 天课程 处理异常	201
10.1 异常处理的概念	201
10.1.1 通过逻辑代码预防错误	201

10.1.2 导致异常的原因	202
10.2 异常处理	203
10.2.1 使用 try 和 catch	203
10.2.2 捕获异常信息	204
10.2.3 给 try 语句提供多个 catch 语句	206
10.2.4 理解异常的处理顺序	207
10.3 使用 finally 添加最后执行的操作	207
10.4 常见的异常	213
10.5 定义自己的异常类	213
10.6 引发自己的异常	215
10.6.1 重新引发异常	217
10.7 checked 语句和 unchecked 语句	218
10.7.1 checked 和 unchecked 的格式	219
10.8 总结	219
10.9 问与答	220
10.10 作业	220
10.10.1 小测验	220
10.10.2 练习	221
第 11 天课程 继承	222
11.1 继承的基本知识	222
11.1.1 简单继承	223
11.1.2 使用继承	225
11.1.3 在派生类的方法中使用基类的方法	229
11.2 多态	229
11.3 虚拟方法	231
11.4 抽象类	234
11.5 密封类	237
11.6 终极基类: Object	238
11.6.1 Object 类中的方法	238
11.6.2 装箱和拆箱	239
11.7 将关键字 is 和 as 用于类——类转换	241
11.7.1 关键字 is	241
11.7.2 关键字 as	243
11.8 由不同类型的对象组成的数组	244
11.9 总结	248
11.10 问与答	248
11.11 作业	248

11.11.1 小测验	249
11.11.2 练习	249
第 12 天课程 输入和输出	251
12.1 理解控制台输入和输出	251
12.2 格式化信息	251
12.2.1 格式化数字	254
12.2.2 格式化日期和时间	259
12.2.3 显示枚举中的值	262
12.3 使用字符串	263
12.3.1 字符串方法	264
12.3.2 特殊的字符串格式符——@	266
12.3.3 创建字符串	266
12.4 从控制台获取信息	269
12.4.1 使用 Read 方法	269
12.4.2 使用 ReadLine 方法	270
12.4.3 Convert 类	271
12.5 总结	274
12.6 问与答	274
12.7 作业	275
12.7.1 小测验	275
12.7.2 练习	275
第 13 天课程 接口	276
13.1 接口	276
13.1.1 类和接口之比较	277
13.1.2 使用接口	277
13.1.3 为何使用接口	277
13.2 定义接口	277
13.2.1 定义带方法成员的接口	278
13.2.2 在接口中指定属性	281
13.3 使用多个接口	283
13.4 显式接口成员	284
13.5 从已有的接口派生出新的接口	287
13.6 隐藏接口成员	287
13.7 总结	289
13.8 问与答	289
13.9 作业	289
13.9.1 小测验	289

13.9.2 练习	290
第 14 天课程 索引器、代表和事件	291
14.1 使用索引器	291
14.2 代表	294
14.3 事件	298
14.4 创建事件	298
14.4.1 事件的代表	298
14.4.2 EventArgs 类	298
14.4.3 事件类的代码	299
14.4.4 创建事件处理程序	300
14.4.5 将事件处理程序和事件关联起来	301
14.4.6 将所有的东西组合起来	301
14.4.7 多个事件处理程序	303
14.4.8 删除事件处理程序	305
14.5 总结	307
14.6 问与答	307
14.7 作业	307
14.7.1 小测验	307
14.7.2 练习	308
第 2 周复习	309
用于表示扑克牌的枚举	318
card 结构	318
deck 类	318
CardGame 类	319
查看整副牌	319
总结	319

第三周课程

第 15 天课程 使用.NET 基类	322
15.1 .NET 框架中的类	322
15.1.1 通用语言规范	322
15.1.2 用于组织类型的名称空间	322
15.1.3 ECMA 标准	323
15.1.4 查看.NET 框架类	323
15.2 使用定时器	323

15.3 获取目录和系统环境信息	325
15.4 使用数学函数	327
15.5 使用文件	330
15.5.1 复制文件	330
15.5.2 获取文件信息	334
15.6 使用数据文件	335
15.6.1 理解流	335
15.6.2 读取文件的步骤	335
15.6.3 用于创建和打开文件的方法	336
15.6.4 使用其他文件类型	342
15.7 总结	342
15.8 问与答	342
15.9 作业	343
15.9.1 小测验	343
15.9.2 练习	343
第 16 天课程 创建 Windows 窗体	345
16.1 使用 Windows 和窗体	345
16.2 创建 Windows 窗体	345
16.2.1 编译选项	346
16.2.2 分析第一个 Windows 窗体应用程序	347
16.2.3 Application.Run 方法	347
16.3 定制窗体的外观	348
16.3.1 窗体的标题栏	348
16.3.2 窗体的大小	350
16.3.3 窗体的颜色和背景	352
16.3.4 边框	355
16.4 将控件加入到窗体中	356
16.4.1 使用标签来显示文本	357
16.4.2 建议使用控件的方法	360
16.4.3 使用按钮	361
16.4.4 使用文本框	366
16.4.5 使用其他控件	369
16.5 总结	369
16.6 问与答	370
16.7 作业	370
16.7.1 小测验	370
16.7.2 练习	370

第 17 天课程 创建 Windows 应用程序	372
17.1 使用单选按钮	372
17.1.1 将单选按钮分组	372
17.1.2 使用容器	376
17.2 使用列表框	379
17.2.1 在列表框中添加选项	379
17.3 在窗体中加入菜单	383
17.3.1 创建一个基本菜单	383
17.3.2 创建多个菜单	385
17.3.3 设置菜单的核对标记	388
17.3.4 创建弹出式菜单	392
17.4 显示弹出式对话框和窗体	393
17.4.1 MessageBox 类	394
17.4.2 Microsoft Windows 中已有的对话框	396
17.4.3 弹出自己的对话框	399
17.5 总结	401
17.6 问与答	402
17.7 作业	402
17.7.1 小测验	402
17.7.2 练习	402
第 18 天课程 Web 开发	405
18.1 创建 Web 应用程序	405
18.2 组件的概念	406
18.3 Web 服务	406
18.3.1 创建简单的组件	406
18.3.2 创建 Web 服务	408
18.3.3 创建代理	410
18.3.4 调用 Web 服务	413
18.4 创建常规 Web 应用程序	414
18.4.1 Web 表单	415
18.4.2 创建一个基本的 ASP.NET 应用程序	415
18.4.3 ASP.NET 控件	417
18.5 总结	424
18.6 问与答	424
18.7 作业	425
18.7.1 小测验	425
18.7.2 练习	425

第 19 天课程 C#中的编译指令和调试技术	426
19.1 何为调试	426
19.2 错误类型	426
19.3 查找错误	427
19.4 逐行检查代码——跟踪代码	427
19.5 预处理器编译指令	427
19.5.1 预处理声明	428
19.5.2 条件处理 (#if、#elif、#else、#endif)	431
19.5.3 报告代码中的错误和警告 (#error、#warning)	432
19.5.4 修改行号	434
19.5.5 区域简介	435
19.6 使用调试器	435
19.7 总结	436
19.8 问与答	436
19.9 作业	436
19.9.1 小测验	436
19.9.2 练习	436
第 20 天课程 运算符重载	439
20.1 再谈函数重载	439
20.2 运算符重载	439
20.2.1 重载运算符	442
20.2.2 重载基本的双目数学运算符	443
20.2.3 重载基本的单目数学运算符	445
20.2.4 重载关系运算符	449
20.2.5 重载逻辑运算符	452
20.2.6 运算符重载情况小结	455
20.3 总结	456
20.4 问与答	456
20.5 作业	456
20.5.1 小测验	456
20.5.2 练习	457
第 21 天课程 反射和属性	458
21.1 反射	458
21.2 属性 (attribute)	463
21.2.1 何为属性	463
21.2.2 使用属性	464

21.2.3 使用多个属性	465
21.2.4 使用带参数的属性	465
21.2.5 定义自己的属性	465
21.2.6 访问被关联的属性信息	469
21.2.7 将各部分组合起来	470
21.2.8 单次使用的属性和多次使用的属性	473
21.3 总结	474
21.4 问与答	474
21.5 作业	474
21.5.1 小测验	474
21.5.2 练习	475
第3周复习	476
应用所学的知识	476
展示您的程序	476
附录 A 作业答案	477
第1天课程的答案	477
小测验	477
练习	478
第2天课程的答案	479
小测验	479
练习	479
第3天课程的答案	481
小测验	481
练习	481
第4天课程的答案	482
小测验	482
练习	483
第5天课程的答案	484
小测验	484
练习	484
第6天课程的答案	489
小测验	489
练习	489
第7天课程的答案	492
小测验	492
练习	493
第8天课程的答案	496

目 录

小测验	496
练习	497
第 9 天课程的答案	502
小测验	502
练习	503
第 10 天课程的答案	505
小测验	505
练习	506
第 11 天课程的答案	508
小测验	508
练习	508
第 12 天课程的答案	509
小测验	509
练习	510
第 13 天课程的答案	512
小测验	512
练习	512
第 14 天课程的答案	514
小测验	514
练习	515
第 15 天课程的答案	521
小测验	521
练习	522
第 16 天课程的答案	525
小测验	525
练习	525
第 17 天课程的答案	529
小测验	529
练习	529
第 18 天课程的答案	532
小测验	532
练习	532
第 19 天课程的答案	534
小测验	534
练习	535
第 20 天课程的答案	536
小测验	536
练习	537

第 21 天课程的答案	542
小测验	542
练习	542
附录 B C#关键字	548
附录 C C#命令行编译器标记	553
输出选项	553
/out:<file>	553
/target:<type>或/t:<type>	553
/define:<symbol list>或/d: <symbol list>	553
/doc:<file>	553
输入选项	553
/recurse:<wildcard>	553
/reference:<file list>或/r:<file list>	553
/addmodule:<file list>	554
资源选项	554
/win32res:<file>	554
/win32icon:<file>	554
/resource:<resinfo>或/res:<resinfo>	554
/linkresource:<resinfo>或/linkres:<resinfo>	554
代码生成选项	554
/debug[+ -]	554
/debug:{full pdbonly}	554
/optimize[+ -]或/o[+ -]	554
/incremental[+ -]或/incr[+ -]	554
错误和警告选项	554
/warnaserror[+ -]	554
/warn:<n>或/w<n>	554
/nowarn:<warning list>	554
编程语言选项	555
/checked[+ -]	555
/unsafe[+ -]	555
杂项	555
@<file>	555
/help 或/?	555
/nologo	555
/noconfig	555
高级选项	555

/baseaddress:<address>	555
/bugreport:<file>	555
/codepage:<n>	555
/utf8output	555
/main:<type>或/m:<type>	555
/fullpaths	555
/filealign:<n>	556
/nostdlib[+ -]	556
/lib:<file list>	556
附录 D 不同的计数系统	557
十进制	557
二进制	557
十六进制	558

第一周课程



欢迎使用本书。如果您还不知道阅读本书大部分内容所需的知识，请参阅前言，其中还介绍了本书的内容。

现在开始阅读第 1 周的课程，它将帮助您牢固地掌握编写 C# 程序的基本知识。第 1 天的课程“C# 初步”将带您进入 C# 程序，并介绍创建 C# 程序的步骤。

第 2 天课程“了解 C# 程序”介绍 C# 是如何适应 Microsoft .NET 框架的，还将介绍面向对象编程的基本原理。

第 3 天课程“使用变量存储信息”、第 4 天课程“使用运算符”、第 5 天课程“控制语句”将介绍 C# 编程中必备的核心编程概念，其中包括存储和操纵数据以及控制程序流程方面的概念。

第 6 天课程“类”和第 7 天课程“类方法和成员函数”介绍了类和类方法。类是面向对象编程的核心概念，因此也是 C# 编程的核心概念。

阅读完本周的课程后，您将学到 C# 编程的许多基础概念。您将发现，您已掌握了构建基本 C# 程序的工具和知识。

第 1 天课程

C#初步

欢迎阅读本书，今天将开始成为熟练的 C#程序员之旅。今天将介绍以下内容：

- 为什么说 C#是一种非常棒的编程语言；
- 程序开发周期中的各个步骤；
- 编写、编译和运行第 1 个 C#程序；
- 编译器和链接程序生成的错误消息；
- 可以使用 C#来创建的应用程序类型。

1.1 C#是什么

如果没有听说过 C#，则您购买本书的可能性不大，但很可能对其了解不深。C#（读作 see Sharp）是在 2000 年 6 月发布的，已经面世了很长时间。

C#是微软公司开发的一种新语言，并已提交给 ECMA 进行标准化。该新语言是由微软公司的一个小组开发的，带头人是 Anders Hejlsberg。有趣的是，Hejlsberg 是微软公司杰出的工程师，曾经开发了其他一些产品和语言，其中包括 Borland Turbo C++和 Borland Delphi。开发 C#时，Hejlsberg 着重借鉴现有语言中的优点，并做一些改进。

C#是一种功能强大的、灵活的编程语言，和所有的编程语言一样，它也可用于创建各种应用程序。C#的用途只受您的想象力的限制，该语言没有给您任何约束。C#已被用于创建动态网站、开发工具，甚至编译器。

C#是一种面向对象的编程语言（OOP）。其他编程语言也包含面向对象的特性，但很少是完全面向对象的。本章后面将对 C#和其他一些编程语言进行比较，同时介绍可用它来创建的应用程序类型。在第 2 天的课程“了解 C#程序”中，您将知道使用面向对象的编程语言意味着什么。

1.2 为何使用 C#

很多人认为，根本不需要一种新的编程语言，他们认为 Java、C++、Perl、Microsoft Visual Basic 以及其他的一些语言已提供了所需的所有功能。

C#是一种从 C 和 C++派生而来的语言，但却是完全重新开发的。微软公司借用了 C 和 C++中一些已有的东西，并加入了新特性，这些特性使该语言更容易使用。其中的许多特性与 Java 极其类

似。最终，微软公司在开发该语言时，有很多目标，其中一些包含在微软公司关于 C# 的声明中：

- C#简单；
- C#现代；
- C#是面向对象的。

除了上述原因外，还有其他一些原因：

- C#强大、灵活；
- C#语言包含的单词为数不多；
- C#是模块化的；
- C#将流行。

警告：下一节包含大量的技术术语，但不用担心。其中大部分术语对 C# 程序员而言，是无关紧要的！对于那些重要的术语，本书后面将对其进行解释。

1.2.1 C#简单

C#消除了诸如 Java 和 C++ 等语言中复杂的东西和缺陷，其中包括宏、模板、多重继承和可视化基类。这些内容可能让 C++ 开发人员困惑或带来问题。如果 C# 是您学习的第 1 种语言，则请放心，您不必花时间来学习这些主题。

C#之所以简单是由于它是基于 C 和 C++ 的。如果您熟悉 C 和 C++（甚至 Java），则您将发现 C# 在很多方面与它们极其类似。语句、表达式、操作符以及其他一些功能是直接来自 C 和 C++ 的，但做了一些改进，使该语言更为简单。其中包括消除了冗余的东西以及语法上的变化。例如，在 C++ 中使用成员时，可用的操作符有三个：“::”、“.” 和 “->”。在何时使用其中的某一个操作符是令人迷惑的。在 C# 中，这些操作符被一个操作符 “.” 所代替。对于新程序员而言，这种特性（以及其他一些特性）避免了许多困惑。

注意：如果您使用过 Java，并认为它简单，则您将发现 C# 也简单，大多数人并不认为 Java 简单，但却认为 C# 比 Java 和 C++ 都简单。

1.2.2 C#现代

什么因素决定一种语言为现代语言呢？一种现代语言必须包含诸如异常处理、无用单元收集、可扩展数据类型以及代码安全性等特性。C# 具备上述所有特性。如果您是一名新程序员，则可能想知道这些看起来很复杂的特性，阅读本书后，您将知道它们非常适用于 C# 编程技术！

注意：指针是 C 和 C++ 的有机组成部分，也是这些语言中最令人困惑的地方。C# 消除了指针导致的复杂性和麻烦。在 C# 中，自动无用单元收集和类型安全是其有机组成部分。如果您不熟悉指针、无用单元收集和类型安全的概念，请不要担心。本章后面将介绍这些概念。

1.2.3 C#是面向对象的

新术语：对于面向对象的语言而言，最重要的是封装、继承和多态，而 C# 支持所有这些特性。封装指的是将功能放在单个组件中；继承是一种将已有的代码和功能扩展到新的程序和组件中的结构化方式；多态指的是适应所需完成的工作的能力。关于这些术语和面向对象的更详细的解释，请参考第 2 天的课程。另外，本书将更为详细地讨论这些主题。

1.2.4 C#强大、灵活

正如前面指出的，使用 C# 您只受想象力的限制，该语言对您没有任何约束。C# 可用于创建字

处理程序、图形、电子表格，甚至其他语言的编译器。

1.2.5 C#包含的单词为数不多

新术语：C#只使用有限的几个单词。语言的功能是建立在关键字的基础之上的，C#包含的关键字不多。表1.1列出了C#使用的关键字，其中的大部分用于描述信息。您可能认为，包含的关键字越多，语言的功能越强大，其实并非如此。当您使用C#编写程序时，将发现它可用于完成任何任务。

表 1.1 C#中的关键字

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

注意：C#还使用了其他几个单词，虽然它们不是关键字，但应该将其看作是关键字。具体地说，它们是get、set和value。

1.2.6 C#是模块化的

新术语：C#代码可以（也应该）以程序块——被称为类的方式编写，类包含被称为成员方法的例程。这些类和方法可以在其他应用程序中被重用。通过给类和方法传递信息，可以创建出很有用的、可重用的代码。

注意：第2天的课程将介绍类，而第6天的课程将介绍如何创建类。

1.2.7 C#将流行

C#是最新的编程语言之一，编写本书时，对C#的流行程度还不得而知，但基于很多原因，它

肯定会成为一种非常流行的语言。其中最关键的原因之一是微软公司及其.NET 的远大前程。

微软公司希望 C#流行。虽然一个公司无法使其产品流行,但可以辅助其流行。不久前,微软公司在其操作系统 Bob 上遭受了重大的失败,虽然微软公司期望 Bob 流行,但失败了。

C#比 Microsoft Bob 成功的可能性更大,我不知道微软公司的员工在其日常工作中是否使用 Bob,但微软公司确实正在使用 C#,其许多产品中的一些代码已经使用 C#重新编写过。通过使用 C#,微软公司帮助证明了 C#确实能够满足程序员的需求。

Microsoft .NET 是 C#将成功的另一个原因。.NET 改变了创建和实现应用程序的方式,在.NET 中,几乎可以使用任何编程语言,C#被证明是可供选择的语言。明天的课程中包含介绍.NET 要点的一节。

C#还将因前面提到的所有特性而流行:简单性、面向对象、模块性、灵活性以及简明性。

1.3 C#和其他编程语言的比较

您可能听说过 Visual Basic、C++和 Java。您可能会问,C#和这些编程语言之间有何差别呢?是否应该学习上述三种语言之一,而不是 C#呢?

注意:在Internet上,与.NET相关的论坛讨论得最多的问题如下:

- Java 和 C#之间的区别何在?
- C#是否只是Java 的克隆而已?
- C#和 C++的区别何在?
- 应该学习 Visual Basic.NET 还是 C#?

微软公司宣称,C#融C++的强大和Visual Basic 的简易性于一体。C#确实很强大,但和Visual Basic 一样简单吗?它可能没有 Visual Basic 6 那么简单,但和 Visual Basic.NET (第7版)一样简单,Visual Basic.NET 是重新编写的。因此最终结果是,Visual Basic 确实不比 C#简单。事实上,无论编写任何程序,使用 C#所需的代码都将更少。

虽然 C#删除了 C++让程序员痛苦的一些特性,但其强大性或功能并没有降低。对于 C++很容易出现的编程错误,在 C#中已经可以完全避免了。这在开发程序的时间上,可以节省几个小时,甚至几天。学习本书介绍的主题后,您将对 C#和 C++之间的差别有更深入的了解。

另一种引起了人们很多关注的语言是 Java,与 C++和 C#一样,Java 也是基于 C 语言的。如果您决定以后学习 Java,您将发现在 C#中学到的许多知识也适用于 Java。

您可能还听说过 C 语言。很多人有这样的疑问,即在学习 C#、C++或 Java 之前,是否应学习 C 语言。简单地回答,这完全没有必要。

知道足够多的原因和结论后,您很可能购买本书,以便学习使用 C#来编写程序。接下来的几节将介绍开发程序的步骤,然后从头到尾开发一个简单的程序。

1.4 编程前的准备工作

您解决某个问题时,总是需要采取一些特定的步骤。首先必须确定问题。如果不知道问题是什么,则不可能找到解决之道!知道问题后,便可以制定解决问题的方案。有了方案,通常可以实现之。方案实现后,必须对结果进行测试,以便了解问题是否得到了解决。这种逻辑也适用于包括编程在内的其他许多领域。

使用 C#（或其他任何语言）创建程序时，应该按类似于下面的步骤进行：

1. 确定目标；
2. 确定编写程序时采用的方法；
3. 创建程序，以解决问题；
4. 运行程序，以查看结果。

目标（见第 1 步）可能是编写一个字处理器或数据库程序。一个极其简单的目标是将您的姓名显示在屏幕上。没有目标，则无法编写出有效的程序。

第 2 步是确定编写程序所采用的方法。是否需要一个计算机程序来解决问题？需要记录哪些信息？要使用哪些公式？这一步应确定需要什么以及解决方案的实现次序。

作为一个例子，假设有人叫您编写一个计算圆面积的程序，则第 1 步已经完成，因为您知道目标：计算圆的面积。第 2 步是确定为了计算面积需要什么。这里假设程序用户将提供圆的半径。知道这一点后，您便可以使用公式 πr^2 来获得答案。有了所需的信息后，便可以接着完成第 3 步和第 4 步，这两步被称为程序开发周期。

1.5 程序开发周期

程序开发周期有其自己的步骤。第 1 步是使用编辑器创建包含源代码的文件；第 2 步是编译源代码，以创建被称为可执行文件或库文件的中间文件；第 3 步是运行程序，以查看它是否按原来预想的情况运行。

1.5.1 创建源代码

新术语：源代码是一系列指示计算机执行要完成的任务的语句和命令。前面指出过，程序开发周期的第 1 步是在一个编辑器中输入源代码。例如下面是一行 C# 源代码：

```
System.Console.WriteLine("Hello, Mom!");
```

该语句命令计算机在屏幕上显示消息“Hello, Mom!”（现在，请不用担心该语句是如何工作的）。

1.5.1.1 使用编辑器

新术语：编辑器是一个可用于输入和保存源代码的程序，对于 C#，可用的编辑器很多，其中的一些是专门为 C# 开发的，而其他的不是。

编写本书时，为 C# 开发的编辑器为数不多，但随着时间的推移，将有更多的编辑器面世。微软公司在其 Visual Studio 产品中添加了 C# 功能，该产品包含 Visual C#。这是最主要的编辑器。但即使没有 Visual Studio.NET，仍然可以使用 C# 进行编程。

还有其他可用于 C# 的编辑器。和 Visual Studio.NET 一样，其中的很多让您不用离开编辑器便可以完成开发周期的所有步骤。更重要的是，这些编辑器大部分都将使用不同的颜色来显示您输入的文本，这使得查找错误的工作更为容易。许多编辑器甚至会提供关于您需要输入什么内容的帮助信息，并提供了一个健壮的帮助系统。

如果没有 C# 编辑器，也不用烦恼。大多数计算机系统都包含一个可用作编辑器的程序。如果您使用的操作系统是 Microsoft Windows，则可以将 NotePad 或 WordPad 用作编辑器；如果使用的是 Linux 或 UNIX 系统，则可以使用诸如 ed、ex、edit、emacs 或 vi 等编辑器。

大多数处理程序都使用特殊的代码来格式化其文档，其他程序无法正确地读取这些代码。许

多字处理程序（如 WordPerfect、Microsoft Word 和 WordPad）能够将源文件保存为文本格式。要将字处理程序的文件保存为文本文件，请在保存时选择文本选项。

注意：可以到计算机商店、计算机邮购目录和计算机编程杂志的广告中寻找其他的编辑器。编写本书时，下面几个编辑器已经面市：

- CodeWrite：一个为ASP、XML、HTML、C#、Perl、Python等提供特殊支持的编辑器，可以从www.premia.com下载。
- EditPlus：一种Internet-就绪（ready）的文本编辑器、HTML编辑器和Windows环境下的程序员编辑器。可以替代NotePad，同时为Web页开发人员和程序员提供了许多强大的特性，包括使用不同的颜色显示代码。该编辑器可以从www.editplus.com下载。
- JEdit：用于Java的开发源代码（Open-Source）编辑器，也可用于C#，能够以不同的颜色显示代码。该编辑器可以从<http://jedit.sourceforge.net>下载。
- Duncan Chen开发的Poorman IDE：提供了一个突显C#和Visual Basic.NET语法的编辑器，同时让您能够运行编译器和捕获控制台输出，因此您无需离开Poorman IDE。Poorman可以从www.geocities.com/duncanchen/poormanide.htm下载。
- Mike Krüger开发的SharpDevelop：是一个免费的Microsoft .NET平台上的C#编辑器。它是一个开放源代码编辑器，因此您可以从www.icsharpcode.net下载其源代码和执行文件。

1.5.1.2 给源代码文件命名

保存源代码文件时，应该提供一个描述其功能的名称。另外，对于C#程序的源代码文件，应将其扩展名设置为.cs。虽然您可以给源代码文件指定任何的文件名和扩展名，但.cs是公认的合适的扩展名。

1.5.2 执行C#程序

在深入介绍程序开发周期之前，需进一步了解C#程序是如何运行的，这非常重要。C#程序不同于使用其他编程语言创建的程序。

新术语：C#程序将在通用语言运行阶段环境（Common Language Runtime，或称CLR）中运行。这意味着如果您创建一个C#语言的可执行程序，并试图在一个没有安装CLR或兼容运行阶段环境的机器上运行它，则将无法执行。*可执行*意味着程序可以被计算机运行或执行。

创建运行阶段环境程序的好处在于可移植性。在诸如C和C++等以前的语言中，要创建一个可在不同的平台（操作系统）上运行的程序，必须编译不同的可执行程序。例如，编写C应用程序时，如果希望它能够在Linux和Windows系统上运行，则必须创建两个可执行程序——一个用于Linux机器，另一个用于Windows机器。使用C#时，只需要创建一个可执行程序，该程序可在上述任何机器上运行。

新术语：如果希望程序的执行速度尽可能快，则需要创建一个真正的可执行程序。计算机使用的是数字（二进制）指令，这被称为*机器语言*。程序必须从源代码被翻译为机器语言，这项工作是由一种叫作编译器的程序完成的。编译器利用源代码生成一个磁盘文件，该文件中包含与源代码语句对应的机器语言指令。对于诸如C和C++等程序，编译器将创建一个无需做进一步的工作便可被执行的文件。

对于C#，编译器并不生成机器语言文件，而是生成一个中间语言（Intermediate Language，或称IL）文件。由于这种文件不能被计算机直接执行，因此需要对其进行进一步翻译或编译，以便计算机能直接执行它，这项工作是由CLR或其他兼容的C#运行阶段环境完成的。

CLR 首先对 IL 文件做最终的编译,即将可移植的 IL 代码转换为计算机能够理解并运行的语言(机器语言)。CLR 实际上只编译程序中将使用的部分,这样可以节省时间。另外,当 IL 文件中的某部分被转换为汇编语言后,便无需再次进行编译,因为编译后的代码将被保存,以后该部分被执行时,将直接使用编译后的代码。

注意: 由于运行阶段环境需要编译 IL 文件,因此与诸如 C++ 等编译型语言相比,程序首次的运行时间可能稍长。程序首次被完全执行后,这种时间上的差别将不复存在,因为从此以后被执行的将是完全被编译的版本。

注意: C# 程序在最后一刻编译的特性被称为即时编译(Just In Time compiling, 或称 jitting)。

1.5.3 编译 C# 源代码

要创建 IL 文件,需要使用 C# 编译器。通常,您使用 `csc` 命令(后面带源代码文件的名称)来运行编译器。例如,要编译名叫 `radius.cs` 的源代码文件,可以输入下述命令行:

```
csc radius.cs
```

如果您使用的是图形开发环境,则编译工作将更为简单。在大多数图形环境中,您都可以通过单击“编译”工具栏按钮或选择合适的菜单选项来编译程序。代码被编译后,可以单击工具栏按钮“运行”或选择合适的菜单选项来执行程序。有关编译和运行程序的具体方法,请参阅编译器的手册。

新术语: 编译后,将得到一个 IL 文件。如果您查看对应目录中的文件清单,将发现一个扩展名为 `.exe` 的新文件,其文件名与源代码文件相同。该文件是编译后的程序,称为**组合体**(assembly),能够在 CLR 上运行。组合体文件包含 CLR 为执行程序所需的所有信息。

图 1.1 说明了从源代码到可执行文件的过程。

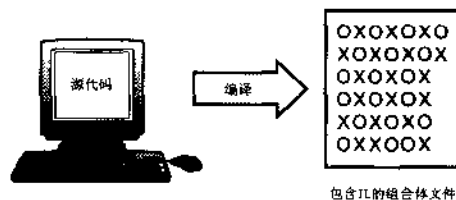


图 1.1 您编写的 C# 源代码被编译器转换为中间语言(IL)代码

注意: 通常有两种可交付使用的 C# 程序:可执行程序 and 库。本书前两周的课程重点介绍可执行文件,其扩展名为 `.exe`。也可以将 C# 用于另一种编程,即编写 ASP.NET 页面上的脚本。有关库的知识将在第三周的课程中介绍。

1.5.4 完成开发周期

程序被编译为 IL 文件后,便可以在命令行提示符下输入其名称来运行它(就像运行其他任何程序一样)。

运行程序时,如果得到的结果与原来预想的不同,则需要回到开发周期的第一步,确定导致问题的原因,并在源代码中进行更正。修改源代码后,需要对其重新进行编译,以创建更正后的可执行文件版本。这一过程将一直重复下去,直到程序的执行情况与预想的完全一致。

C# 开发周期

第 1 步: 使用编辑器编写源代码。C#源代码文件的扩展名通常为.cs (如 a_program.cs、database.cs 等);

第 2 步: 使用 C#编译器对程序进行编译。如果编译器发现程序没有任何错误, 则将生成一个扩展名为.exe 或.dll 的组合体文件。例如, 默认情况下, myprog.cs 被编译为 myprog.exe; 如果发现错误, 编译器将报告。在这种情况下, 您必须返回到第 1 步, 对源代码进行更正。

第 3 步: 在安装了 C#运行阶段环境 (如 CLR) 的机器上执行程序。此时, 应对程序进行测试, 以确定它是否正确地运行。如果不是, 则返回到第 1 步, 对源代码进行修改。

图 1.2 说明了程序开发的步骤。除了最简单的程序外, 您可能都得重复这一过程多次才能完成程序。即使是最有经验的程序员, 也不可能一步就编写出一个完整的、没有错误的程序! 由于需要重复编辑—编译—测试多次, 因此熟练使用工具——编辑器、编译器和运行阶段环境——至关重要。

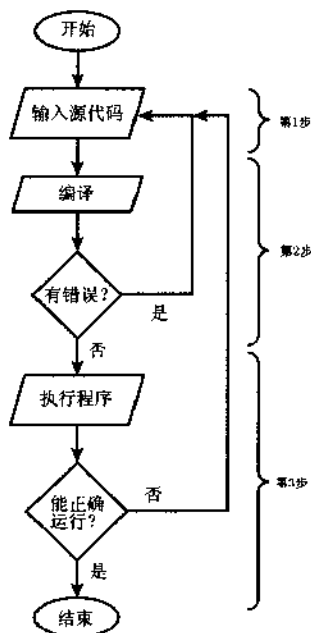


图 1.2 开发 C#程序的步骤

1.6 第 1 个 C#程序

您可能迫切地希望编写第 1 个 C#程序。为帮助您熟悉编译器, 程序清单 1.1 包含一个您很快就能完成的程序。当前, 您可能不明白其中的所有内容, 但可以体会一下编写、编译和运行真正的 C#程序的过程。

这里使用的是一个名为 hello.cs 的程序, 它只是将单词 “Hello, World!” 显示到屏幕上。该程序常被人们用来介绍如何编程, 对于学习 C#, 这也是一个很好的例子。hello.cs 的源代码如程序清单 1.1 所示。输入该程序清单时, 请不要输入左边的编号。

程序清单 1.1 hello.cs

```
1: class Hello
2: {
```

```
3: static void Main()
4: {
5:     System.Console.WriteLine("Hello, World!");
6: }
7: }
```

请务必按软件安装提示安装编译器。准备好编译器和编辑器后,请按下一节中介绍的步骤输入、编译和执行 `hello.cs`。

1.6.1 输入并编译 `hello.cs`

请按下述步骤输入并编译 `hello.cs`:

1. 启动编译器。

2. 使用键盘输入如程序清单 1.1 所示的 `hello.cs` 中的源代码。请不要输入其中的行号,这些行号只是为方便引用而提供的。在每一行结束时按 Enter 键。输入代码时,大小写一定要相同。C# 是区分大小写的,因此如果您改变了大小写,将发生错误。

警告: 如果您是 C 或 C++ 程序员,则很可能犯一种常见的错误。在 C 和 C++ 中, `main()` 是小写的,而在 C# 中, `Main()` 中的 M 是大写的。如果您输入的是小写的 `m`,将出错。

3. 使用文件名 `hello.cs` 保存源代码。

4. 查看目录中的文件,确保 `hello.cs` 被保存了。

5. 编译 `hello.cs`,如果使用的是命令行编译器,则使用下面的命令:

```
csc hello.cs
```

如果使用的是集成开发环境,则单击合适的图标或选择合适的菜单项。此时应显示一条消息,指出没有错误或警告。

6. 查看编译器给出的消息。如果没有错误或警告,则万事大吉。

如果您输入程序时有错,则编译器将捕获这种错误,并显示错误消息。例如,如果将单词 `Console` 错误地拼写为 `Consol`,则您将看到与下面类似的消息:

```
hello.cs(5,7): error CS0117: 'System' does not contain a definition for 'Consol'
```

7. 如果出现错误消息,则回到第 2 步。在编辑器中打开文件 `hello.cs`,仔细地将文件的内容与程序清单 1.1 进行比较,修改错误的地方,然后进入到第 3 步。

8. 至此,您的第一个 C# 程序编译好了,可以运行。如果显示文件名为 `hello` 的文件,则将看到如下两个文件:

- `hello.cs`: 使用编辑器创建的源代码文件;
- `hello.exe`: 编译 `hello.cs` 时生成的可执行程序。

9. 在命令行输入 `hello`,执行 `hello.exe`,屏幕上将显示 “Hello, World!”。

注意: 如果在 Windows 资源管理器中双击文件 `hello.exe`,将可能看不到结果。该程序将在命令行窗口的方式运行,当您双击该文件时,将打开一个命令行窗口,运行该程序,当程序运行完毕后窗口将被关闭。这一过程持续的时间非常短,您可能没有看到任何情况发生。因此,最好打开一个命令行窗口,切换到包含该程序的目录,然后在命令行下运行该程序。

祝贺您!您通过输入、编译,便可以运行第一个 C# 程序。得承认, `hello.cs` 是一个非常简单的程序,它没有完成任何有用的工作,但这只是开始。事实上,当前的大多数专家级程序员都是以这

种方式——通过编译一个“Hello World”程序——开始学习的。

1.6.1.1 编译错误

当编译器发现源代码的某些内容无法编译时，便会出现编译错误。拼写错误、排列错误以及其他一些错误都可能导致编译器无法进行编译。所幸的是，现代的编译器不仅仅停止编译，还会告诉您问题所在。这使得查找和更正源代码中的错误更为容易。

为证明这一点，可以故意在前面输入的 `hello.cs` 程序中引入一个错误。如果您完成了前面的步骤，则磁盘上将会有 `hello.cs` 的一个拷贝。使用编辑器，将光标移到第5行的最后，并删除分号，得到的 `hello.cs` 如程序清单1.2所示。

程序清单 1.2 包含一个错误的 `hello.cs`

```
1: class Hello
2: {
3:     static void Main()
4:     {
5:         System.Console.WriteLine("Hello, World!")
6:     }
7: }
```

接下来，保存该文件。然后输入如下所示的编译器命令，以编译该文件：

```
csc hello.cs
```

由于其中有错，因此编译将无法完成。编译器将显示一条与下面类似的消息：

```
hello.cs(5,48): error CS1002: ; expected
```

该行消息由下述三部分组成：

- `hello.cs`：有错误的文件的名称；
- `(5,48)`：指出错误的位置——第5行的第48个字符处；
- `error CS1002: ; expected`：对错误的描述。

该消息提供了大量的信息，指出当编译器编译到文件 `hello.cs` 的第5行的第48个字符时，期望找到一个分号，但没有。虽然编译器在检测和定位错误方面非常聪明，但它毕竟不是爱因斯坦。您必须使用C#语言方面的知识，对编译器提供的消息进行解释，并确定所报告的错误的实际位置。通常，可以在编译器指出的行中找到错误，但如果没有找到，则很可能在前一行。刚开始查找错误时，可能很困难，但不久后便可以得心应手。

结束该主题之前，来看另一个编译错误的例子。在编辑器中打开 `hello.cs`，并做如下修改：

1. 在第5行的末尾加上分号；
2. 删除单词 `Hello` 前面的双引号。

保存并编译该文件。这次编译器将显示与下面类似的消息：

```
hello.cs(5,32): error CS1010: Newline in constant
```

该错误消息正确地指出了错误的位置：第5行的第32个字符，这正是显示的第一个双引号所在的位置，但未能正确地指出代码中少了一个双引号。这里编译器对问题做出了猜想，虽然猜想离实际情况不远，但并不正确。

提示：如果编译器报告了多个错误，而您只找到一个。更正该错误并重新编译文件后，您可能发现只需更正这个错误，程序编译时便没有任何错误。

1.6.1.2 逻辑错误

您还可能犯另一种错误：逻辑错误。逻辑错误无法依靠编译器来发现，而只能依靠您自己。即使程序中的 C# 代码是完整无缺的，但仍然可能有错误。例如，假设您需要计算圆的面积，但使用的公式却是圆周率和半径乘积的两倍：

面积 = $2\pi r$

将上述公式输入到程序，并编译和执行该程序时，您将得到答案。C# 程序的语法是正确的，但每次运行该程序得到的答案却是错误的。因为其中的逻辑不正确，该公式计算的是圆周长，而不是面积。您应该使用公式 πr^2 。

无论编译器多么优秀，都无法发现逻辑错误。您必须通过仔细地核对代码并运行程序来找到这种错误。

1.7 C# 程序的类型

在结束今天的课程之前，有必要介绍一下使用 C# 可以创建哪些类型的应用程序。使用该语言可创建的程序类型很多：

- **控制台应用程序：**控制台应用程序是从命令行运行的，本书创建的都是控制台应用程序，它们主要是基于字符（或文本）的，因此理解起来相对较为简单。
- **Windows 应用程序：**也可以创建充分利用 Microsoft Windows 提供的图形用户界面（GUI）的 Windows 应用程序。
- **Web 服务：**Web 服务指的是可以通过 Web 调用的例程。
- **Web 表单/ASP.NET 应用程序：**ASP.NET 应用程序是在 Web 服务器上执行的，生成动态 Web 页。

除了上述应用程序外，C# 还可用于完成大量其他的工作，其中包括创建库、控件等。

1.8 总 结

在今天课程的开始，介绍了 C# 的特征，包括功能强大、灵活、面向对象等。另外，还指出了 C# 简单而现代。

今天探讨了编写 C# 程序的各个步骤——被称为程序开发过程。阅读后面的内容之前，您应该牢固地掌握编辑—编译—测试循环。

在程序开发中，错误是不可避免的。C# 编译器能够发现源代码中的错误，并显示错误消息，指出错误的特征和位置。使用这些信息，您可以更正源代码中的错误。但需要牢记的是，编译器并不总能够准确地指出错误的特征和位置。有时候，您需要使用 C# 方面的知识来找到导致错误消息的真正原因。

1.9 问与答

问：C# 程序能够在任何机器上运行吗？

10. 下述错误很可能位于哪一行?

my_prog.cs(35,6): error CS1010: Newline in constant

1.10.2 练习

1. 使用文本编辑器查看编译程序清单 1.1 时创建的 .exe 文件。 .exe 文件与源代码文件类似吗?
(退出编辑器时, 请别保存该文件)

2. 输入下述程序, 并编译之 (不要输入行号)。该程序有何功能?

```
1: // circle.cs - Using variables and literals
2: // This program calculates some circle stuff.
3: //-----
4:
5: using System;
6:
7: class variables
8: {
9:     public static void Main()
10:    {
11:        //Declare variables
12:
13:        int radius = 4;
14:        const double PI = 3.14159;
15:        double circum, area;
16:
17:        //Do calculations
18:
19:        area = PI * radius * radius;
20:        circum = 2 * PI * radius;
21:
22:        //Print the results
23:
24:        Console.WriteLine("Radius = {0}, PI = {1}", radius, PI );
25:        Console.WriteLine("The area is {0}", area);
26:        Console.WriteLine("The circumference is {0}", circum);
27:    }
28: }
```

3. 输入下述程序, 并编译之 (不要输入行号)。该程序有何功能?

```
1: class Aclass
2: {
3:     static void Main()
4:     {
5:         int x,y;
6:         for ( x = 0; x < 10; x++, System.Console.Write( "\n" ) )
7:             for ( y = 0; y < 10; y++ )
8:                 System.Console.Write( "X" );
9:     }
```

```
10: }
```

4. 下述程序有问题，请将其输入到编辑器中，并编译之。哪些行引发错误消息？

```
1: class Hello
2: {
3:     static void Main()
4:     {
5:         System.Console.WriteLine(Keep Looking!);
6:         System.Console.WriteLine(You'll find it!);
7:     }
8: }
```

5. 将练习 3 中的程序作如下修改，并重新编译和执行该程序。现在，该程序完成的工作是什么？

```
8:         System.Console.Write( "{0}", (char) 1 );
```

第2天课程

了解 C# 程序

除了了解程序的基本组成部分外，您还需要了解 C# 程序的结构。今天的课程将介绍以下内容：

- C# 应用程序的组成部分；
- C# 语句和表达式；
- 面向对象编程技术的特征；
- 封装、继承、多态和重用；
- 在程序中显示基本信息。

2.1 C# 应用程序

今天的课程首先重点探讨一个 C# 应用程序范例。通过程序清单 2.1，您将了解 C# 应用程序的一些重要组成部分。

程序清单 2.1 app.cs：一个 C# 应用程序范例

```
1: // app.cs - A sample C# application
2: // Don't worry about understanding everything in
3: // this listing. You'll learn all about it later!
4: //-----
5:
6: using System;
7:
8: class sample
9: {
10:     public static void Main()
11:     {
12:         //Declare variables
13:
14:         int radius = 4;
15:         const double PI = 3.14159,
16:         double area;
17:
18:         //Do calculation
19:
```

```
20:         area = PI * radius * radius;
21:
22:         //Print the results
23:
24:         Console.WriteLine("Radius = {0}, PI = {1}", radius, PI );
25:         Console.WriteLine("The area is {0}", area);
26:     }
27: }
```

您应该将该程序清单输入到编辑器中，然后使用编译器来创建可执行程序。可以将该程序保存为 `app.cs`，然后在命令提示符下输入下面的命令来编译该程序：

```
csc app.cs
```

如果您使用的是可视化编辑器，则应该可以通过菜单项选择编译器。

警告：记住，输入上述程序清单时，不要输入其中的行号。行号是为方便讨论程序清单而加入的。

运行该程序，将得到如下所示的输出：

```
Radius = 4, PI = 3.14159
The area is 50.3344
```

正如您看到的，该程序清单的输出简单明了。首先显示了半径和圆周率的值，然后显示了根据上述值计算得到的圆面积。

接下来的几节，将介绍该程序的各个部分。请不要因为不能理解其中的所有内容而担心，后面的课程将更详细地讨论这些内容。接下来的几节旨在让您有一个初步的认识。

2.1.1 注释

程序清单 2.1 的开始四行是注释。注释用于在程序中输入将被编译器忽略的信息。为何要输入被编译器忽略的信息呢？原因很多。

注释常被用来提供关于程序清单的描述性信息，例如身份证明信息。另外，通过加入注释，可以指出期望程序清单完成的功能。即使只有您自己使用程序清单，加入关于程序的功能及其如何完成这些功能的信息也是一个好主意。虽然您现在知道程序清单的功能——因为您要编写它，但以后您可能不知道当时您是如何想的。当您程序清单提供他人时，注释将帮助他们了解代码要做什么。注释也可用于提供程序清单的修订情况。

对于注释，需要知道的主要一点是，它们是为使用程序清单的程序员提供的。编译器将忽略它们。在 C# 中，可使用的注释有三种：

- 单行注释；
- 多行注释；
- 文档注释。

提示：注释将被编译器删除，因此在程序清单中加入注释，不会增加任何成本。有疑问时，应该包含注释。

2.1.1.1 单行注释

程序清单的第 1~4 行使用的都是单行注释，第 12、18 和 22 行也是单行注释。单行注释的格式如下：

```
//comment text
```

两个斜杠表示注释的开始，直到该行的结尾都被看作注释。

单行注释不一定非得从行首开始，注释前可以有 C#代码，但从两个斜杠开始到本行的结束都属于注释。

2.1.1.2 多行注释

程序清单 2.1 没有多行注释，但有时候注释可能需要跨越多行。在这种情况下，可以在每行的开始加上两个斜杠（就像第 1~4 行那样），也可以使用多行注释。

多行注释包含开始和结束标记。要开始多行注释，可以输入斜杠和星号：

```
/*
```

该标记之后的所有内容都是注释，直到您输入结束标记为止。结束标记为一个星号和斜杠：

```
*/
```

下面为一个注释：

```
/* this is a comment */
```

下面也是一个注释：

```
/* this is
a comment that
is on
a number of
lines */
```

对于上述注释，也可以以下面的方式输入：

```
// this is
// a comment that
// is on
// a number of
// lines
```

使用多行注释的优点在于，只需加入/*和*/便可以将一段代码作为注释。编译器将/*和*/之间的所有内容都看作注释，并忽略它们。

警告：不可以对多行注释进行嵌套，也就是说，不能在一个多行注释中包含另一个多行注释。

例如，下面的注释方式是错误的：

```
/*Beginning of a comment...
/* with another comment nested */
*/
```

2.1.1.3 文档注释

C#包含一种特殊的注释，让您能够自动创建外部文档。

这种注释使用三个斜杠，而不像单行注释那样使用两个斜杠。它还使用可扩展标记语言（XML）式的标记。XML 是一种用于标记数据的标准。虽然在 C#可以使用任何有效的 XML 标记，但常用的标记为<c>、<code>、<example>、<exception>、<list>、<para>、<param>、<paramref>、<permission>、<remarks>、<return>、<see>、<seealso>、<summary>和<value>。

文档注释被放置在代码清单中。程序清单 2.2 是一个使用文档注释的例子。您可以像以前那样

编译该清单，见第 1 天的课程“C#初步”。

程序清单 2.2 xmlapp.cs: 使用 XML 注释

```
1: // xmlapp.cs - A sample C# application using XML
2: //           documentation
3: //-----
4:
5: /// <summary>
6: /// This is a summary describing the class.</summary>
7: /// <remarks>
8: /// This is a longer comment that can be used to describe
9: /// the class. </remarks>
10: class MyApp
11: {
12:     /// <summary>
13:     /// The entry point for the application.
14:     /// </summary>
15:     /// <param name="args"> A list of command line arguments</param>
16:     public static void Main(string[] args)
17:     {
18:         System.Console.WriteLine("An XML Documented Program");
19:     }
20: }
```

编译并执行该程序清单时，将得到如下所示的输出：

```
An XML Documented program
```

要获得 XML 文档，必须以不同于以前的方法来编译该程序清单，即在命令行进行编译时，加上参数/doc，如下所示：

```
csc /doc:xmlfile xmlapp.cs
```

这样可使以后运行程序得到的结果与前面相同。区别在于还生成了一个名为 xmlfile 的文件，其中包含 XML 格式的文档。您可以给 XML 文件取任何名称。对于程序清单 2.2，生成的 XML 文件如下：

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>xmlapp</name>
  </assembly>
  <members>
    <member name="T:MyApp">
      <summary>
        This is a summary describing the class.</summary>
      <remarks>
        This is a longer comment that can be used to describe
        the class. </remarks>
    </member>
    <member name="M:MyApp.Main(System.String[])">
```

```

    <summary>
    The entry point for the application.
    </summary>
    <param name="args"> A list of command line arguments</param>
  </member>
</members>
</doc>

```

注意：介绍XML和XML文件不在本书的范围之内。

注意：如果您使用的是诸如Visual Studio.NET工具，则需要查看文档或帮助系统，来了解如何生成XML文档。即使您使用的是这种工具，仍可以在命令行编译程序。

2.2 C#应用程序的基本组成部分

编程语言由一系列组合词组成，计算机程序以一种有机的方式使用和格式化这些单词。C#的语言的组成部分如下：

- 空白；
- C#关键字；
- 字面值；
- 标识符。

2.2.1 空白

新术语：从程序清单2.1可以知道，它被格式化了，以便代码对齐并易于阅读。程序清单中的空白区被称为空白（whitespace）。这一术语源自白纸上看不到空格。空白可以由空格、制表符、换行符和回车组成。

编译器几乎总是忽略空白，因此您可以添加任意数目的空格、制表符或换行符。例如程序清单2.1的第14行：

```
int radius = 4;
```

这是一个格式化良好的行，各项之间有一个空格。可以在这行代码中加入更多的空格：

```
int    radius    =    4    ;
```

其运行方式与原来相同。实际上，C#编译器编译程序时，将删除多余的空格。您还可以将上述代码格式化为跨越多行：

```
int
radius
=
4
;
```

虽然其可读性很差，但仍然能够运行。

通常情况下，由于空格将被忽略，因此可以自由地使用它来帮助格式化代码，并提高其可读性。编译器唯一不忽略的是用双引号标示的文本中的空白。对于双引号间的文本，空白至关重要，

因为其中的文本将按其原来的情况使用。在前面的程序清单中，出现过文本被双引号标示的情况。在程序清单 2.1 中，第 24 行和 25 行便包含被双引号标示的文本。这些文本将被原原本本地显示到屏幕上。

提示：使用空白，可以提高代码的可读性。

2.2.2 C#关键字

第 1 天的课程介绍过，关键字是专用术语，它们有特殊的含义，组成了一种语言。C#语言有很多关键字，这些关键字在表 1.1 列出了。

使用 C#编程时，这些关键字有特殊的含义。阅读完本书后，您将知道这些含义。由于所有这些关键字都有特殊含义，因此被保留，您不能将其挪作他用。如果将程序清单 2.1（或本书的任何程序清单）和表 1.1 进行比较，您将发现清单的很大一部分内容是由关键字组成的。

注意：附录 C “C#关键字”包含了对每个 C#关键字的简短定义。

2.2.3 字面值

新术语：字面值是直观的硬编码值，它们的值与您看到的值一致。例如，数字 4 和 3.14159 都是字面值，另外，被标示在双引号之间的文本是字面文本。明天的课程将更详细地介绍字面值及其应用。

2.2.4 标识符

新术语：除了关键字和字面值外，C#程序中还可以使用其他的单词。这些单词被称为标识符。在程序清单 2.1 中，有很多标识符，包括第 6 行的 System、第 8 行的 sample、第 14 行的 radius、第 15 行的 PI、第 16 行的 area 以及第 22 行的 PI、radius 和 area 等。

2.3 C#应用程序的结构

单词和短语用于组成句子，而句子用于组成段落。同样，空白、关键字、字面值和标识符用于组成表达式和语句，而表达式和语句组成了程序。

2.3.1 C#表达式和语句

新术语：表达式类似于短语，是由关键字组成的代码片段。例如，下面是一个简单的范例：

```
PI = 3.14159
PI * radius * radius
```

语句类似于句子，它们表达一个完整的意思。语句通常以标点字符分号（；）结束。在程序清单 2.1 中，第 14、15、16 行都是语句。

2.3.2 空语句

有一种语句——空语句需要特别地提一下。正如前面介绍的，语句通常以分号结束。实际上，可以让单个分号自成一，这种语句不执行任何操作。由于其中没有任何表达式需要执行，因此这种语句被称为空语句。

您可能会问，为什么要包含一个不执行任何操作的空语句？虽然看起来您不会这样做，但阅读完本书后，您将发现空语句非常有用。第 5 天的课程将介绍空语句最流行的用途之一。

2.4 分析程序清单 2.1

学习一些概念后,有必要对程序清单 2.1 做仔细的讨论。接下来的几节将回顾一下程序清单 2.1 中的每一行代码。

2.4.1 第 1~4 行: 注释

正如前面介绍的,第 1~4 行包含的是将被编译器忽略的注释。这些注释是供您和其他任何复核源代码的人使用的。

2.4.2 第 5、7、13、17、21 和 23 行: 空白

第 5 行为空白行。前面介绍过,空白行只不过是空白而已,编译器将忽略它们。加入该行是为提高程序清单的可读性。第 7、13、17、21 和 23 行也是空白行,即使删除这些空白行,也不会影响程序的运行。

2.4.3 第 6 行: using 语句

第 6 行是一条语句,包含关键字 `using` 和字面值 `System`。和大多数语句一样,该语句也是以分号结束的。关键字 `using` 用于减少输入量。通常, `using` 关键字用于导入名称空间。第 6 天的课程将更详细地介绍名称空间和 `using` 关键字。

2.4.4 第 8 行: 类声明

C# 是一种面向对象编程 (OOP) 语言。面向对象语言使用类来声明对象。该程序定义了一个名为 `sample` 的类,阅读完本书后,您将理解类,而今天课程的后面将概要地介绍类。从第 6 天开始将详细介绍 C# 中的类。

2.4.5 第 9、11、26 和 27 行: 标点字符

第 9 行包含一个左花括号,其对应的右花括号位于第 27 行。第 11 行的左花括号与第 26 行的右花括号对应。这些括号用于包含和组织代码块。在接下来的四天中学习各种不同的命令后,您将明白如何使用这些花括号。

2.4.6 第 10 行: `Main()`

新术语: 计算机需要知道从哪里开始执行程序。C# 程序从 `Main()` 函数 (如第 10 行所示) 开始执行。函数是一组代码,通过调用函数名称可以执行这些代码。第 7 天的课程将详细介绍函数。`Main()` 函数很特别,因为它是程序的起点。

警告: 在 C# 应用程序中,代码的大小写至关重要。`Main()` 中,只有 `M` 是大写。对于 C 和 C++ 程序员,需要注意的是,在 C# 中, `main()` (小写的) 不起作用。

2.4.7 第 14、15 和 16 行: 声明

第 14、15 和 16 行包含用于创建标识符的语句,这些标识符用于存储信息。后面将使用这些标识符来完成计算。第 14 行声明了一个用于存储半径值的标识符,字面值 4 被赋给该标识符。第 15 行声明了一个用于存储圆周率的标识符 `PI`,其值被指定为 3.14159。第 16 行声明了一个没有被指定任何值的标识符。

2.4.8 第 20 行: 赋值语句

第 20 行包含一条简单的语句, 它将标识符 `PI` 与半径的平方相乘。然后将该表达式的结果赋给标识符 `area`。

2.4.9 第 24 行和 25 行: 调用函数

第 24、25 行是该程序清单中最复杂的表达式。这两行调用预定义的例程, 将信息输出到控制台 (屏幕)。本章后面将介绍这些预定义的函数。

2.5 面向对象编程 (OOP)

正如前面指出的, C# 是一种面向对象的语言。为充分利用 C#, 应了解面向对象语言的概念。接下来的几节将简要地介绍对象以及是什么使得一种语言成为面向对象语言的。随着阅读本书后面的内容, 将知道如何在 C# 中使用这些概念。

2.5.1 面向对象的概念

是什么使得一种语言成为面向对象的呢? 最明显的答案是, 该语言使用对象! 但这一答案并不全面。第 1 天的课程介绍过, 与面向对象语言相关的概念有三个:

- 封装;
- 多态;
- 继承。

还存在第四个概念: 重用, 这是使用面向对象语言得到的结果。

2.5.1.1 封装

封装指的是创建“包”, 其中包含您所需的所有东西。对于面向对象编程而言, 这意味着创建一个对象 (如圆), 该对象能够完成您需要对它执行的所有操作。这包括跟踪关于圆的所有信息 (如半径和圆心), 以及指导如何完成圆的功能, 如计算半径和绘制圆。

通过封装一个圆, 您让用户无需关心圆是如何工作的, 而只须知道如何与圆交互。用户为何需要知道有关圆的信息在内部是如何存储的呢? 只要他们能够让圆完成所需的工作, 便不应该知道。

2.5.1.2 多态

多态是呈现多种形态的功能。这可用于面向对象编程的两个领域。首先, 这意味着您可以以许多不同的方式调用对象和例程, 而得到的结果却是相同的。以圆为例, 您可能想调用圆对象来获得其面积。为此, 您可以使用三点或一个点和半径。无论使用哪种方式, 得到的结果都是相同的。在诸如 C 等过程性语言中, 为实现这两种计算面积的方法, 需要使用两个名称不同的例程。而在 C# 中, 仍然需要使用两个例程, 但它们的名称可以相同。您只需调用该例程并传递信息, 程序将自动根据传递的信息, 使用正确的例程。调用例程的用户无需为到底应该使用哪个例程操心, 而只要调用例程即可。

多态也能够使用多种格式, 第 11 天的课程将更详细地介绍。

2.5.1.3 继承

继承是最复杂的面向对象概念。圆对象已经很好, 但球对象是不是更好呢? 球只不过是一种特殊的圆, 它具有圆的所有特征, 只不过是三维的。可以这么说, 球是一个特殊的圆, 具有圆的所有属性, 同时还有一些其他属性。通过使用圆来创建球, 就将继承圆的所有属性。能够继承属性是继承的特征之一。

2.5.1.4 重用

之所以使用面向对象语言，最重要的原因之一是重用。创建类后，您可以重用它来创建大量的对象。通过使用继承以及前面描述的其他一些特性，您可以创建出可以在大量程序中以许多方式进行重用的例程。通过封装功能，您可以创建经过测试，并被证明能够正确运行的例程。这意味着您无需测试功能是如何起作用的，而只要正确地使用它即可。这使得重用例程快速而简单。

2.5.2 对象和类

新术语：了解面向对象语言的概念后，还要了解类和对象的差别，这至关重要。类对要创建的东西做了定义，实际创建的东西是对象。简单地说，类是用于创建对象的定义。

在描述类时，经常使用的一个类比是蛋糕刀。蛋糕刀定义了蛋糕的形状，它不是蛋糕，也不能食用。它只不过是一种构造，可不断地用于制作出蛋糕。使用蛋糕刀来制作蛋糕时，您知道每个蛋糕的形状都将相同，还知道可以使用蛋糕刀制作大量的蛋糕。

和蛋糕刀一样，类也可用于创建大量的对象。例如，拥有一个圆类后，便可以使用它来创建大量的圆对象。创建一个绘制圆的程序后，便可以通过一个圆类创建出大量的圆对象。您可以将雪人中的每个圆作为一个对象，但只需要一个类来定义它们。

您还可以创建大量其他的类，包括名称、卡片、应用程序、点、圆、地址、雪人（可以使用圆类）等等。

注意：从第6天起，将更详细地介绍类和对象。今天只是让您对面向对象的概念有个大概的了解。

2.6 显示基本信息

为使本书开始就更有趣一些，这里介绍两个可用于显示信息的例程。了解这两个例程后，您便能够显示一些基本信息。

贯穿本书，都将被用来显示信息的两个例程是：

- `System.Console.WriteLine();`
- `System.Console.Write();`

这两个例程将信息打印到屏幕上，它们以相同的方式打印信息，只有很小的差别。`WriteLine()` 在新的一行中打印信息，而 `Write()` 在打印信息时，不开始一个新行。

要显示到屏幕的信息位于括号内，如果要显示文本，则将它们用双引号标示。例如，下面的语句打印文本“Hello World”。

```
System.Console.WriteLine("Hello World");
```

注意：第一天的课程使用过该例程。

下面的范例打印另一些文本：

```
System.Console.WriteLine("This is a line of text");  
System.Console.WriteLine("This is a second line of text");
```

上述代码的输出如下：

```
This is a line of text  
This is a second line of text
```

下面两行代码的输出是什么呢?

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

如果您认为结果为:

```
Hello World!
```

那么您错了, 结果应为:

```
Hello
World!
```

其中每个单词都位于单独的一行, 如果上述两行代码中使用的是例程 `Write()`, 则结果将如下:

```
Hello World!
```

从上面可以知道, `WriteLine()` 在显示完文本后, 将移到下一行, 而 `Write()` 不会这样做。程序清单 2.3 演示了这两个例程的用法。

程序清单 2.3 `display.cs`: 使用 `WriteLine()` 和 `Write()`

```
1: // display.cs - printing with WriteLine and Write
2: //-----
3:
4: class display
5: {
6:     public static void Main()
7:     {
8:         System.Console.WriteLine("First WriteLine Line");
9:         System.Console.WriteLine("Second WriteLine Line");
10:
11:         System.Console.Write("First Write Line");
12:         System.Console.Write("Second Write Line");
13:
14:         // Passing parameters
15:         System.Console.WriteLine("\nWriteLine: Parameter = {0}", 123 );
16:
17:         System.Console.Write("Write: Parameter = {0}", 456);
18:     }
19: }
```

要在命令行编译该程序清单, 请使用下面的命令:

```
csc display.cs
```

如果您使用的是集成开发环境, 则可以选择相应的编译菜单项。

该程序清单的输出如下:

```
First WriteLine Line
Second WriteLine Line
First Write LineSecond Write Line
WriteLine: Parameter = 123
```

```
Write: Paramater = 456
```

分析：该程序清单的第8和9行使用`System.Console.WriteLine()`来打印两段文本，从输出中可以知道，这两个文本分别位于单独的一行中。第11和12行使用的是`System.Console.Write()`，这两行代码打印的文本位于同一行中，打印后没有换行。第15行和17行演示了如何在这两个例程中使用参数。

2.6.1 打印其他信息

除了打印双引号之间的文本外，还可以传递要在文本内打印的值。请看下面的范例：

```
System.Console.WriteLine("The following is a number: {0}", 456);
```

其输出如下：

```
The following is a number: 456
```

正如您看到的，“{0}”已被替换为文本后面的值，其格式如下：

```
System.Console.WriteLine("text", value);
```

其中Text可以是任何要显示的文本。“{0}”是一个值占位符，花括号表明这是占位符。“0”是一个指示器，指出使用双引号后面的第一个值。值和文本之间用逗号隔开。

要打印的文本中可以包含多个占位符，其中每个占位符使用接下来的一个序列数。要打印两个值，可以使用下面的方式：

```
System.Console.Write("Value1 is {0} and value2 is {1}", 123, "Brad");
```

上述代码的输出如下：

```
Value1 is 123 and value2 is Bard
```

本书后面将更详细的介绍这两个例程。

警告：第一个占位符的编号为0，而不是1。

注意：程序清单2.3的第15行还包含一些怪异的文本。该行中的“\n”并非错误，它是一个指示器，指出在打印后面的信息之前进行换行。第3天的课程将更详细地介绍它。

2.7 总 结

今天的课程继续为学习C#打基础。今天介绍了C#应用程序的一些基本组成部分，指出注释使得程序更易于理解。

您还知道C#应用程序的基本组成部分包括空白、C#关键字、字面值和标识符。通过一个应用程序，您知道如何将这各部分组合起来，创建一个完整的程序清单。这包括用作应用程序入口的特殊标识符：`Main()`。

探讨一个程序清单后，对面向对象编程技术做了简要的介绍，包括封装、多态、继承、重用等概念。

最后对如何使用`System.Console.WriteLine()`和`System.Console.Write()`做了概要性介绍。这两个例程都用于将信息显示到屏幕（控制台）上。

2.8 问与答

问：基于组件和基于对象之间的区别何在？

答：有些人认为 C# 是基于组件的。基于组件的开发可以看作是对面向对象编程技术的扩展。组件只不过是执行特定任务的独立代码片段。基于组件的编程需要创建大量可被重用的独立组件，您可以将它们链接起来，来创建应用程序。

问：还有哪些面向对象的语言？

答：其他面向对象的语言包括：C++、Java 和 SmallTalk。微软公司的 Visual Basic.NET 也可用于面向对象编程。还有其他的面向对象语言，只是上面列出的是最流行的。

问：何为组合（composition），这是一个面向对象术语吗？

答：许多人分不清组合和继承，但它们不是一回事。组合指的是在一个对象中使用另一个对象，图 2.1 组合了许多圆。这与前面介绍的球不同，球并不是由圆组成的，它是圆的扩展。现对组合与继承之间的区别总结如下：组合在一个类（对象）包含另一个类时发生；继承则在一个类（对象）是另一个类的扩展时发生。

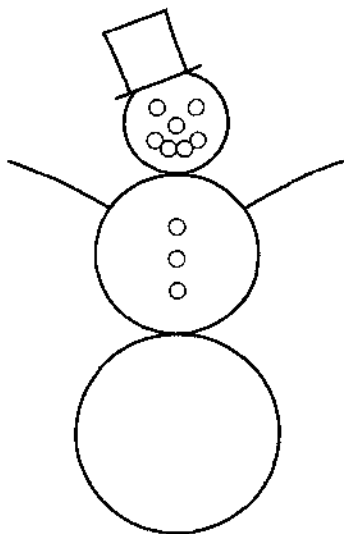


图 2.1 由多个圆组成的雪人

2.9 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

2.9.1 小测验

1. 在 C# 程序中，可以使用哪三种注释？
2. 如何将上述三种注释输入到 C# 程序中？
3. 空白对 C# 程序有何影响？
4. 下面哪些单词是关键字？

field、cast、as、object、throw、baseball、catch、football、fumble、basketball。

5. 字面值是什么?

6. 下面哪句话是正确的?

表达式是由语句组成的;

语句是由表达式组成的;

表达式和语句之间毫无关系。

7. 空语句是什么?

8. 面向对象编程最重要的概念有哪些?

9. WriteLine()和 Write()之间的区别何在?

10. 在 WriteLine()和 Write()中打印值, 什么被用作占位符?

2.9.2 练习

1. 输入并编译下面的程序 (listit.cs)。别忘了, 请不要输入行号。

```
1: // ListIT.cs - program to print a listing with line numbers
2: //-----
3:
4: using System;
5: using System.IO;
6:
7: class ListIT
8: {
9:     public static void Main(string[] args)
10:    {
11:        try
12:        {
13:
14:            int ctr=0;
15:            if (args.Length <= 0 )
16:            {
17:                Console.WriteLine("Format: ListIT filename");
18:                return;
19:            }
20:            else
21:            {
22:                FileStream f = new FileStream(args[0], FileMode.Open);
23:                try
24:                {
25:                    StreamReader t = new StreamReader(f);
26:                    string line;
27:                    while ((line = t.ReadLine()) != null)
28:                    {
29:                        ctr++;
30:                        Console.WriteLine("{0}: {1}", ctr, line);
31:                    }
32:                    f.Close();
33:                }
34:                finally
```



```
35:         {
36:             f.Close();
37:         }
38:     }
39: }
40: catch (System.IO.FileNotFoundException)
41: {
42:     Console.WriteLine ("ListIT could not find the file
43: {0}", args[0]);
44: }
45: catch (Exception e)
46: {
47:     Console.WriteLine("Exception: {0}\n\n", e);
48: }
49: }
50: }
```

2. 运行练习 1 中输入的程序，得到什么样的结果？在命令行输入下面的命令，再次运行该程序：

```
listit listit.cs
```

这次的运行结果是什么？

3. 输入、编译并运行下面的程序，该程序的运行结果是什么？

```
1: // ex0203.cs - Exercise 3 for Day 2
2: //-----
3:
4: class Exercise2
5: {
6:     public static void Main()
7:     {
8:         int x = 0;
9:
10:        for( x = 1; x <= 10; x++ )
11:        {
12:            System.Console.Write("{0:D3} ", x);
13:        }
14:    }
15: }
```

4. 下面的程序有问题。输入并编译该程序，哪一行会导致错误消息？

```
1: // bugbust.cs
2: //-----
3:
4: class bugbust
5: {
6:     public static void Main()
7:     {
```

```
8.         System.Console.WriteLine("\nA fun number is {1}", 123 );  
9:     }  
10: }
```

5. 编写将您的姓名打印到屏幕上的代码。

第3天课程

使用变量存储信息

开始编写程序后，您将很快发现您需要记录不同类型的信息，这可能是客户的姓名、您银行账户中的金额或您最喜欢的电影明星的年龄。为了记录这些信息，计算机程序需要使用某种方式来保存值。今天的课程将介绍以下内容：

- 变量是什么？
- 在 C# 如何给变量命名？
- 使用各种数值变量。
- 字符值和数字值之间的异同。
- 如何声明和初始化变量。

3.1 变 量

变量是计算机内存中被命名的数据存储单元。在程序中使用变量名称，可以引用其中存储的信息。例如，可以创建一个用于保存数字的、名为 `my_variable` 的变量，在该变量中，可以存储不同的数字。

您也可以创建变量来存储非数字信息。您可以创建一个名为 `BankAccount` 的变量，来存储银行的账号；创建一个名为 `email` 的变量来存储电子邮件地址；创建一个名为 `address` 的变量来存储某人的邮送地址。无论所存储的信息是何种类型，变量都可以用来获得其值。

3.1.1 变量名

要在 C# 程序中使用变量，您必须知道如何给变量命名。在 C# 中，变量名必须遵循以下规则：

- 可以包含字母、数字和下划线 (`_`)；
- 第一个字符必须是字母，也可以是下划线（但不推荐这样做）。下划线通常用于特殊命令，有时候其可读性不好。
- 区分大小写。C# 是区分大小写的，因此名称 `Count` 和 `count` 指的是两个不同的变量。
- 不能将 C# 关键字用作变量名。关键字是属于 C# 语言的单词（C# 关键字的完整列表，见附录 B “C# 关键字”）。

下表列出了一些合法和不合法的 C# 变量名范例。

变 量 名	合 法 性
Percent	合法
Y2x5_w7h3	合法
yearly_cost	合法
_2010_tax	合法, 但不推荐
checking#account	非法: 包含非法字符#
double	非法: 是一个C#关键字
9byte	非法: 第1个字符 为数字

由于C#是区分大小写的, 因此名称 `percent`、`PERCENT` 和 `Percent` 将被视为不同的变量。C#程序员通常只在变量名中使用小写字母, 虽然不要求这样做。程序员常常混合使用大小写。通常, 只有常量(本章后面将介绍)才全部大写。

变量名只要符合前面列出的规则即可, 例如计算圆面积的程序可以将半径的值存储在一个名为 `radius` 的变量中。这种变量名清晰地说明了其用途。当然, 您也可以使用名称 `x` 或 `billy_gates`, 但当其他人查看源代码时, 将不知道其含义。虽然使用描述性变量名, 其输入的时间可能长些, 但使程序更为清晰, 因此是值得的。

使用多个单词来创建变量名有许多命名约定。请看变量名 `circle_radius`。使用下划线将变量名中的单词隔开可使其更容易理解。另一种方式叫做 *Pascal 表示法*, 这种方式不使用下划线, 而是将每个单词的第一个字母大写, 即不是 `circle_radius`, 而是 `CircleRadius`。另一种越来越流行的表示法是 *骆驼表示法* (*Camel notation*)。骆驼表示法与 *Pascal 表示法* 类似, 只是变量的第一个字母是小写的而已。一种特殊的骆驼表示法叫 *匈牙利表示法*, 变量名中还包含其数据类型的信息(如是一个带小数数字还是文本), 以帮助您确定存储的信息的类型。本书使用下划线, 因为对大多数人来说, 这种方式更容易阅读。您应该确定自己要采用的方式。

应该	不应该
应使用描述性的变量名	命名变量时, 不要全部使用大写字母
命令变量时, 应采用并坚持一种方式	

注意: C#支持Unicode字符集, 这意味着可以存储和使用任何语言中的字母, 命名变量时, 也可以使用任何Unicode字符。

3.2 使用变量

在使用变量之前, 必须声明它。变量声明将变量的名称及其存储的信息类型告诉编译器。如果程序使用了未声明的变量, 编译器将生成错误消息。

声明变量也使计算机能够为变量预留内存。通过指出变量存储的信息的具体类型, 您可以获得

最佳的性能，并避免浪费内存。

3.2.1 声明变量

变量声明的格式如下：

```
typename varname;
```

其中, `typename` 指定了变量的类型, 在接下来的几节中, 您将知道 C# 中可用的变量类型; `varname` 是变量的名称。要声明可存储标准整数的变量, 可使用下面的代码行:

```
int my_number;
```

声明的变量名为 `my_number`, 数据类型为 `int`。在接下来的一节中将介绍, 数据类型 `int` 用于声明整数变量, 对于本例而言, `int` 非常合适。

也可以在一行中声明多个同一种数据类型的变量, 方法是使用逗号将变量隔开。这可以使程序清单更为简洁。请看下面的代码行:

```
int count, number, start;
```

该代码行声明了三个变量: `count`、`number` 和 `start`。其中每个变量的数据类型都是 `int`, 即整型 (`integer`)。

注意: 虽然在同一行声明多个变量更为简洁, 但不推荐总是这样做。有时候, 使用多个声明可使代码更容易阅读和理解。分别声明对性能的影响微乎其微。

3.2.2 给变量赋值

知道如何声明变量后, 接下来介绍如何存储值, 这很重要。毕竟变量是用来存储信息的。

将信息存储到变量中的格式如下:

```
varname = value;
```

其中, `varname` 是变量名, `value` 是被存储到变量中的值。例如, 要将数字 5 存储到变量 `my_variable` 中, 可以使用下面的代码:

```
my_variable = 5;
```

要修改变量的值, 只需将新的值赋给变量即可:

```
my_variable = 1010;
```

程序清单 3.1 演示了如何给变量赋值以及修改变量的值。

程序清单 3.1 var_values.cs: 给变量赋值

```
1: // var_values.cs - A listing to assign and print the value
2: //               of a variable
3: //-----
4:
5: using System;
6:
7: class var_values
8: {
9:     public static void Main()
```

```

10:  {
11:      // declare my_variable
12:      int my_variable;
13:
14:      // assign a value to my_variable
15:      my_variable = 5;
16:      Console.WriteLine("\nmy_variable contains the value {0}", my_variable);
17:
18:      // assign a new value to my_variable
19:      my_variable = 1010;
20:      Console.WriteLine("\nmy_variable contains the value {0}", my_variable);
21:  }
22: }

```

该程序清单的输出如下：

```

my_variable contains the value 5
my_variable contains the value 1010

```

分析：在编辑器中输入上述程序清单，然后编译并执行它。有关编译和执行的方法，请参阅第1天的课程。程序清单的开始三行是注释，第11、14和18行也包含注释，注释提供信息，编译器将忽略它们。第5行将名称空间System包含进来，您完成诸如将信息写到屏幕上等工作时需要该名称空间。第7行声明了类var_values，它将是您的程序。第9行声明了程序的入口——Main()函数。记住，Main()的首字母必须大写，否则将出错。

第12行是今天才开始介绍的内容，它声明了一个名叫my_variable的整型(int)变量。该行代码被执行后，计算机将知道存在一个名为my_variable的变量，并让您能够使用它。第15行使用该变量，并将5赋给它。第16行使用Console.WriteLine来显示变量my_variable的值。从输出可以知道，显示的值为5（在第15行指定的）。第19行将my_variable的值从5改为1010。由于第20行对Console.WriteLine的调用打印的是新值1010，而不是5，因此您知道赋给变量新值的操作生效了。

在第19行将值赋给my_variable后，值5被覆盖了。从此以后，程序不再知道5曾经存在过。

注意：使用变量之前必须声明它，但几乎可以在程序清单的任何地方进行声明。

3.2.3 初始化变量

您可能会问，是否可以在声明变量的同时给它赋值。是的，完全可以。事实上，确保在声明变量的同时初始化它，是一个很好的习惯。要在声明变量my_variable的同时，将其初始化为8，可以将前面的做法组合起来：

```
int my_variable = 8;
```

使用下面的格式，可以在声明的同时初始化任何变量：

```
typename varname = value;
```

也可以在同一行声明多个变量，并给它们赋值：

```
int my_variable = 8, your_variable = 1000;
```

上述代码行声明了两个名称分别是my_variable和your_variable的整型变量，前者的值被指定为

8, 后者被指定为 1000。这些声明之间以逗号分隔, 而语句以分号结束。程序清单 3.2 演示了如何使用这种语句。

程序清单 3.2 multi_variable.cs: 给多个变量赋值

```
1: // multi_variables.cs
2: // A listing to assign values to more than one variable.
3: //-----
4:
5: using System;
6:
7: class multi_variables
8: {
9:     public static void Main()
10:    {
11:        // declare the variables
12:        int my_variable = 8, your_variable = 1000;
13:
14:        // print the original value of my_variable
15:        Console.WriteLine("my_variable was assigned the value {0}, my_variable);
16:
17:        // assign a value to my_variable
18:        my_variable = 5;
19:
20:        // print their values
21:        Console.WriteLine("\nmy_variable contains the value {0}", my_variable);
22:        Console.WriteLine("\nyour_variable contains the value {0}", your_variable);
23:    }
24: }
```

该程序清单的输出如下:

my_variable was assigned the value 8

my_variable contains the value 5

your_variable contains the value 1000

分析: 该程序清单的第12行声明并初始化了两个变量, 其中变量my_variable被初始化为8, 而变量your_variable的值被初始化为1000。第15行打印变量my_variable的值, 以便您知道其包含的内容。从输出可以知道, 该变量包含了刚被赋给的值8。

第18行将值5赋给my_variable。第21和22行打印两个变量最终的值。变量my_variable的值为5, 其原来的值永远消失了。而变量your_variable仍然包含的是最初的值1000。

3.2.4 使用未初始化的变量

如果使用未初始化的变量, 将会出现什么情况呢? 请看下面的代码片段:

```
int blank_variable;
Console.WriteLine("\nmy_variable contains the value {0}", blank_variable);
```

在上述代码片段中，第二行打印 `blank_variable`。`blank_variable` 的值是什么呢？该变量是在第一行声明的，但没有被初始化。您将无法知道 `blank_variable` 的值，因为编译器将不会编译该程序，程序清单 3.3 证明了这一点。

程序清单 3.3 `blank.cs`: 使用未被初始化的变量

```
1: // blank.cs  C Using unassigned variables.
2: // This listing causes an error!!
3: //-----
4:
5: using System;
6:
7: class blank
8: {
9:     public static void Main()
10:    {
11:        int blank_variable;
12:
13:        Console.WriteLine("\rmy_variable contains the value {0}", blank_variable);
14:    }
15: }
```

分析：该程序不会被编译，编译器将给出如下所示的错误消息：

`blank.cs(13,67): error CS0165: Use of unassigned local variable 'blank_variable'`

C#不允许使用未被初始化的变量。

注意：在诸如C和C++等其他语言中，上述程序清单将被编译。在这种情况下，打印的 `blank_variable` 的值将是无用信息。C#能够防止这种错误发生。

3.3 了解计算机内存

对于知道计算机内存如何工作的读者，可以跳过本节；如果没把握，则请阅读之。本节的内容有助于您理解编程技术。

计算机中的 RAM 有何用途呢？其用途有多种，但对程序员而言，需要关心的只有数据存储。数据是 C# 程序使用的信息。您的程序是否在维护一个联络清单，监控股票市场，规划预算，跟踪商品的价格。正在运行的程序所使用的信息（姓名、股价、开支、价格）被保存在计算机内存的变量中。

计算机运行时，使用随机存储器（RAM）来存储信息。RAM 位于计算机内的集成电路或芯片中。RAM 是易失性的，这意味着需要时，其中的信息将被擦除，并替换为新的信息；而且，只有当计算机开启时，RAM 才具备记忆功能，当计算机关闭后，其中的信息将丢失。

字节是计算机数据存储的基本单位。所有的计算机都安装了一定数量的 RAM。系统中 RAM 的量通常以 MB（兆字节）为单位，如 32MB、64MB、128MB 等。1MB 等于 1024KB（千字节），而 1KB 为 1024 字节。因此，一个具有 8MB 内存的系统实际上有 8192（ 8×1024 ）KB 的 RAM，即 8388608（ 8192×1024 ）字节。表 3.1 列出了存储某些类型的数据所需要的内存量（单位为字节）。

表 3.1 存储数据所需的内存空间

数 据	需要的内存空间 (字节)
字母 x	2
数字 500	2
数字 241.105	4
字符串 Teach Yourself C#	34
打字机中一页的内容	约 4000

计算机中的 RAM 是顺序排列的，一个字节跟着一个字节。每个字节都有用于标识它的唯一性地址——一个用来将它与内存中其他字节区分开来的地址。内存单元被依次指定地址，从 0 开始，直到系统的最大内存。就现在而言，您无需关心地址，地址是自动处理的。

粗略地了解一些关于内存的知识后，我们回到 C# 编程技术，介绍 C# 如何使用内存来存储信息。

3.4 C#数据类型

您已经知道如何声明、初始化变量以及如何修改变量的值。知道可以使用哪些数据类型至关重要。前面介绍过，声明变量时必须指定其数据类型。您已经知道，使用关键字 `int` 声明的变量可以存储整数。整数是不包含小数部分的数字。到目前为止，您声明的只是用来存储整数的变量，如果要存储诸如小数和字符等其他类型的数据，该如何办呢？

3.5 数值变量类型

C# 提供了多种不同的数值变量类型。您需要不同类型的变量，因为不同的数值所需的内存空间不同，对其执行数学运算的难易程度也不同。小型整数（如 1199 和 -8）需要的内存空间较少，计算机对其执行数学运算（如加、乘等）时速度较快；而大型整数以及包含小数部分的数值需要的内存空间较多，对其执行数学运算所需的时间也更长。通过使用合适的变量类型，可以确保程序的运行效率尽可能高。

接下来的几节将数值数据类型分为 4 类：

- 整数；
- 浮点数；
- `Decimal`；
- 布尔型。

本章前面介绍过，变量被存储在内存中；而且，不同类型的信息需要的内存量也不同。存储变量所需的内存空间取决于数据类型。程序清单 3.4 中的一些代码以前没有介绍过，但它让您知道存储一些 C# 数据类型所需要的内存空间。

新术语：编译该程序清单时，必须包含一些提供给编译器的额外信息。这种信息被称为编译器标记 (flag)，可以在命令行中包含它们。具体地说，需要添加 `/unsafe` 标记，如下所示：

```
csc /unsafe sizes.cs
```

如果使用的是集成开发环境，则需要按其文档说明设置 `unsafe` 选项。

程序清单 3.4 size.cs: 各种数据类型所需的内存空间

```

1: // sizes.cs--Program to tell the size of the C# variable types
2: //-----
3:
4: using System;
5:
6: class sizes
7: {
8:     unsafe public static void Main()
9:     {
10:         Console.WriteLine( "\nA byte    is {0} byte(s)", sizeof( byte ));
11:         Console.WriteLine( "An sbyte   is {0} byte(s)", sizeof( sbyte ));
12:         Console.WriteLine( "A char    is {0} byte(s)", sizeof( char ));
13:         Console.WriteLine( "\nA short   is {0} byte(s)", sizeof( short ));
14:         Console.WriteLine( "An ushort is {0} byte(s)", sizeof( ushort ));
15:         Console.WriteLine( "\nAn int    is {0} byte(s)", sizeof( int ));
16:         Console.WriteLine( "An uint   is {0} byte(s)", sizeof( uint ));
17:         Console.WriteLine( "\nA long    is {0} byte(s)", sizeof( long ));
18:         Console.WriteLine( "An ulong  is {0} byte(s)", sizeof( ulong ));
19:         Console.WriteLine( "\nA float   is {0} byte(s)", sizeof( float ));
20:         Console.WriteLine( "A double  is {0} byte(s)", sizeof( double ));
21:         Console.WriteLine( "\nA decimal is{0}byte(s)", sizeof( decimal));
22:         Console.WriteLine( "\nA boolean is {0} byte(s)", sizeof( bool ));
23:     }
24: }

```

该程序清单的输出如下:

```

A byte    is 1 byte(s)
An sbyte   is 1 byte(s)
A char     is 2 byte(s)

A short    is 2 byte(s)
An ushort  is 2 byte(s)

An int     is 4 byte(s)
An uint    is 4 byte(s)

A long     is 8 byte(s)
An ulong   is 8 byte(s)

A float    is 4 byte(s)
A double   is 8 byte(s)

A decimal  is 16 byte(s)

A boolean  is 1 byte(s)

```

分析：虽然还未介绍这些数据类型，但我认为在这里提供该程序清单很有用。在阅读接下来的几节时，可以参考该程序清单及其输出。

该程序清单使用了 C# 关键字 `sizeof`，该关键字指出变量的大小。在该程序清单中，`sizeof` 被用来显示不同数据类型的大小。例如要确定 `int` 的大小，可以使用下面的代码：

```
sizeof(int)
```

如果您声明了一个名为 `x` 的变量，则可以使用下面的代码来确定其大小——实际上是其数据类型的大小：

```
sizeof(x)
```

程序清单 3.4 的输出列出了存储每种 C# 数据类型所需的字节数。对于 `int`，需要 4 个字节的存储空间，而 `short` 需要两个字节。使用的内存量决定了可以存储的数字的大小，接下来的几节将更详细地进行介绍。

关键字 `sizeof` 并不太常用，但对于说明本章的内容很有用。关键字 `sizeof` 进入到内存中，以确定变量或数据类型的大小。在 C# 中，应避免直接进入内存中。第 8 行额外加入了关键字 `unsafe`。如果不加入该关键字，则编译该程序时将出错。就现在而言，只需知道加入关键字 `unsafe` 的原因在于关键字 `sizeof` 直接进入内存。

警告：可以使用 C# 关键字 `sizeof`，但通常应避免使用它。关键字 `sizeof` 有时候会直接访问内存，以确定数据类型的大小。在纯粹的 C# 程序中，应避免直接访问内存。

3.5.1 整型数据类型

到目前为止，您只使用过一种整型数据类型：`int`。整型数据类型用于存储整数。整数指的是任何不包含小数部分的数值。数值 1、1000、56 万亿和 -534 都是整型值。

C# 提供了 9 种整型数据类型，如下所示：

- 整数 (`int` 和 `uint`)；
- 短整数 (`short` 和 `ushort`)；
- 长整数 (`long` 和 `ulong`)；
- `Byte` (`byte` 和 `sbyte`)；
- 字符 (`char`)。

3.5.1.1 整数

从程序清单 3.4 知道，整数占用 4 个字节的内存。这包括数据类型 `int` 和 `uint`。本书前面的很多程序都使用过 `int` 数据类型。虽然您可能还不了解这种数据类型，但它并不能存储任何数字，它只能存储使用 4 个字节（32 位）能够表示的符号整数，即 -2147483648 到 2147483647 之间的任何整数。

`Int` 类型的变量是带符号的，即可以为正，也可以为负。从技术上说，4 个字节可以存储大到 4294967295 的整数，但使用 32 位中的一位表示正负后，可以存储的值便只能大到 2147483647，而小到 -2147483648。

前面介绍过，信息的存储单位叫字节。一个字节实际上是由 8 位组成的，位是计算机最基本的存储单位。位的取值有两个：0 或 1。使用位和二进制，可以将数字存储在多个位中。附录 C 将详细介绍二进制。

要使 `int` 能够存储更大的值，可以让它不带符号。无符号整数只能是正的。这样做的好处显而

易见, `uint` 数据类型用于声明无符号整数, 其存储的值为 0 ~ 4294967295。

如果您试图存储一个更大的数字, 将出现什么情况呢? 如果在 `int` 或 `uint` 变量中存储带小数的值, 情况将如何呢? 如果在 `uint` 变量中存储负数, 情况又将如何呢? 程序清单 3.5 对上述问题做出了回答。

程序清单 3.5 `int_conv.cs`: 执行非法操作

```
1: // int_conv.cs
2: // storing bad values. Program generates errors and won't compile.
3: //-----
4:
5: using System;
6:
7: class int_conv
8: {
9:     public static void Main()
10:    {
11:        int val1, val2;    // declare two integers
12:        uint pos_val;      // declare an unsigned int
13:
14:        val1 = 1.5;
15:        val2 = 9876543210;
16:        pos_val = -123;
17:
18:        Console.WriteLine("val1 is {0}", val1);
19:        Console.WriteLine("val2 is {0}", val2);
20:        Console.WriteLine("pos_val is {0}", pos_val);
21:    }
22: }
```

该程序清单的编译输出如下:

```
int_conv.cs(14,15): error CS0029: Cannot implicitly convert type 'double' to 'int'
int_conv.cs(15,15): error CS0029: Cannot implicitly convert type 'long' to 'int'
int_conv.cs(16,18): error CS0031: Constant value '-123' cannot be converted to a 'uint'
```

警告: 该程序将出现编译错误。

分析: 该程序不能通过编译。正如您看到的, 编译器捕获了所有的错误。第14行试图将包含小数的值赋给一个整数变量; 而第15行试图将一个过大的值赋给整数变量。记住, `int` 变量能够存储的最大值为 2147483647。最后, 第16行试图将一个负数赋给一个无符号整数 (`uint`) 变量。正如输出所表明的, 编译器捕获了所有这些错误, 并阻止程序被编译。

3.5.1.2 短整型

`int` 和 `uint` 变量使用 4 个字节的内存, 通常不需要存储这么大的整数, 例如在记录星期几 (1 ~ 7)、年龄、烘烤蛋糕的温度时, 并不需要这么大的数。

当您需要存储整数, 而又想节省内存时, 可以使用 `short` 和 `ushort` 数据类型。和 `int` 一样, `short` 也用于存储整数; 不同的是, 它只占用 2 个字节, 而不是 4 个字节。从程序清单 3.4 可以知道, 对于 `short` 和 `ushort`, `sizeof` 的返回值都是 2。如果要存储的整数有正有负, 则可以使用 `short`; 如果只有

正数，同时想利用额外的空间，则可以使用 `ushort`。`Short` 能够存储的整数范围为 $-32768 \sim 32767$ ；而 `ushort` 为 $0 \sim 65535$ 。

3.5.1.3 长整型

如果值太大，无法使用 `int` 和 `uint` 存储，则可以使用另一种数据类型 `long`。与 `short` 和 `int` 一样，`long` 数据类型也有一种无符号形式，叫做 `ulong`。从程序清单 3.4 的输出可以知道，`long` 和 `ulong` 占用的内存都为 8 个字节，这使它们能够存储非常大的整数，其中 `long` 可以存储的整数范围为 $-9223372036854775808 \sim 9223372036854775807$ ，而 `ulong` 为 $0 \sim 18446744073709551615$ 。

3.5.1.4 Byte

您可以将整数存储为占用 2、4 或 8 个字节内存的数据类型。当您需要的整数非常小时，可以使用单个字节来存储。为简化问题，使用 1 个字节内存的数据类型被称为 `byte`。和前面的整型数据类型一样，`byte` 也有带符号形式（`sbyte`）和无符号形式（`byte`）。`Sbyte` 的取值范围为 $-128 \sim 127$ ；而 `byte` 为 $0 \sim 255$ 。

3.5.1.5 字符

除了数字外，您还经常需要存储字符。字符指的是字母，如 A、B、C，乃至扩展字符，如笑脸。您可能需要存储的其他字符，如 \$、%、* 等。您甚至还需要存储外语中的字符。

计算机并不能识别字符，而只能识别数字。为此，所有的字符都被存储为数值。为确保所有的人都使用相同的值，制定了一种名叫 Unicode 的标准。在 Unicode 中，每个字符和符号都使用一个整数表示，这也是字符数据类型被视为整型的原因所在。

为指明数字将被看作字符，可以使用数据类型 `char`。`char` 是一个占用两个字节内存的数字，被解释为一个字符。程序清单 3.6 是一个使用 `char` 的程序。

程序清单 3.6 chars.cs: 使用字符

```
1: // chars.cs
2: // A listing to print out a number of characters and their numbers
3: //-----
4:
5: using System;
6:
7: class chars
8: {
9:     public static void Main()
10:    {
11:        int ctr;
12:        char ch;
13:
14:        Console.WriteLine("\nNumber   Value\n");
15:
16:        for( ctr = 60; ctr <= 95; ctr = ctr + 1)
17:        {
18:            ch = (char) ctr;
19:            Console.WriteLine( "{0} is {1}", ctr, ch);
20:        }
21:    }
22: }
```

该程序清单的输出如下所示：

```
Number Value
```

```
60 is <
61 is =
62 is >
63 is ?
64 is @
65 is A
66 is B
67 is C
68 is D
69 is E
70 is F
71 is G
72 is H
73 is I
74 is J
74 is K
76 is L
77 is M
78 is N
79 is O
80 is P
81 is Q
82 is R
83 is S
84 is T
85 is U
86 is V
87 is W
88 is X
89 is Y
90 is Z
91 is [
92 is \
93 is ]
94 is ^
95 is _
```

分析：该程序清单显示一定范围内的数值及其对应的字符。第11行声明了一个名叫ctr的整数变量，用于遍历一系列的整数。第12行声明了一个名为ch的字符变量。第14行为要显示的信息打印标题。

第16行包含了一些新内容。就现在而言，无需完全理解该行代码。第5天的课程将详细介绍这种语句。现在只需要该行代码将ctr的值设置为60，然后运行第18~19行，再将ctr的值加1。这一过程将一直重复下去，直到ctr的值大于95为止。最终结果是，第18~19行将重复运行，而ctr的值将分别为60、61、62...，直到ctr的值为95为止。

第18行将ctr的值（开始为60）赋给字符变量ch。由于ctr是一个整数，因此您需要命令计算

机将其转换为字符，这是通过(char)语句实现的。关于这一点，后面将做更详细的介绍。

第19行打印存储在ctr和ch中的值，正如您看到的，整数变量ctr是作为数字打印的，而ch的值是作为字符（而不是数字）打印的。从该程序清单的输出可以知道，字符A被表示为值65，而字符B对应的值为66。

注意：计算机实际上只能识别（位于比特中的）1和0，它分别将它们看作是开和关（或电压为正和负）。二进制使用0和1来表示数字，附录C对二进制进行了介绍。

字符面值

如何将字符赋给char变量呢？将字符放在单引号中。例如，要将字符a赋给变量my_char，可以使用下面的语句：

```
my_char = 'a';
```

除了常规字符外，您可能需要使用一些扩展字符。在很多程序清单中，已经使用过一个扩展字符：\n，该字符用于换行。表3.2列出了一些最常用的扩展字符，程序清单3.7演示了一些特殊字符的用法。

表3.2

扩展字符

字 符	含 义
\b	空格
\n	换行符
\t	水平制表符
\\	反斜杠
'\'	单引号
'\"'	双引号

注意：表3.3列出的扩展字符常被称为转移字符，因为反斜杠避开了常规文本，指出后面的字符为特殊（扩展）字符。

程序清单 3.7 chars_table.cs: 特殊字符

```
1: // chars_table.cs
2: //-----
3:
4: using System;
5:
6: class chars_table
7: {
8:     public static void Main()
9:     {
10:         char ch1 = 'Z';
11:         char ch2 = 'x';
12:
13:         Console.WriteLine("This is the first line of text");
```

```

14:     Console.WriteLine("\n\nSkipped three lines");
15:     Console.WriteLine("one\ttwo\tthree <-tabbed");
16:     Console.WriteLine(" A quote: \' \ndouble quote: \'"");
17:     Console.WriteLine("\n ch1 = {0}  ch2 = {1}", ch1, ch2);
18: }
19: }

```

该程序清单的输出如下：

This is the first line of text

```

Skipped three lines
one  two   three <-tabbed
 A quote: '
Double quote: "

ch1 = z  ch2 = x

```

分析：该程序清单说明了两个概念。首先，第10和11行演示了如何将字符赋给char变量——只需将字符放在单引号中即可。第13~17行演示了如何使用扩展字符。第13行没有什么特别的地方，第14行打印三个换行符，然后打印一些文本。第15行打印one、two和three，并在它们之间加上制表符。第16行打印一个单引号和一个双引号，该行的结尾处连续出现了两个双引号。最后，第17行打印了ch1和ch2的值。

3.5.2 浮点数

并非所有的数字都是整数。使用带小数的数字时，需要使用其他的数据类型。和存储整数一样，根据要使用的数字的大小以及内存空间的多少，您可以使用不同的数据类型来存储浮点数。其中最主要的两种是float和double。

3.5.2.1 float

float用于存储带小数的数值。例如，计算圆的周长或面积时，得到的结果常常不是整数。存储诸如1.23或3.14159这样的数字时，需要使用非整型数据类型。

float数据类型使用4个字节的内存来存储数字，因此其可以存储的数值范围大约为 1.5×10^{-45} 到 3.4×10^{38} 。

注意： 10^{38} 相当于执行37次 10×10 ，得到的结果为1后面跟38个零；而 10^{-45} 相当于执行46次 $10 \div 10$ ，得到的结果为在小数点和1之间有44个0。

警告：float的精度为7位，因此float的值经常会有些误差。例如，将10.00减去9.90得到的结果可能不是0.10，而是一个接近于0.099999999的值。这种舍入误差通常是微不足道的。

3.5.2.2 double

double类型的变量占用8个字节的内存，这意味着其取值可以比float大得多。Double的取值大约为 5.0×10^{-324} 到 1.7×10^{308} ，其精度通常为15~16位。

注意：C#支持IEEE 754格式的4字节精度（32位）和8字节精度（64位），因此某些数学函数能够返回非常精确的值。如果您执行除以0的运算，结果将为无穷大（可以是正无穷大或负无穷大），

如果将0除以0,得到的结果将为Not-a-Number。0可以为正,也可以为负 更详细的信息请参阅C#文档。

3.5.3 Decimal

C#提供了另一种可用于存储特殊小数的数据类型: decimal。这种数据类型是为存储精度更高的数字而设计的。将数字存储在 float 和 double 变量中时,将引入舍入误差。例如,将 10.00 减去 9.90 的结果存储在 double 变量中时,得到的值将为 0.099999999999999645,而不是 0.10。如果使用 decimal 变量来执行这种数学运算,则得到的结果将为 0.10。

提示: 如果您计算的是金额,或执行的是精度非常重要的金融方面的计算时,则应该使用 decimal,而不是 float 或 double。

decimal 变量占用 16 个字节的内存。与其他数据类型不同的是,decimal 没有无符号形式。decimal 变量可存储的数值范围为 1.0×10^{-28} 到 7.9×10^{28} ,其精度为 28 位。

3.5.4 布尔型

最后一种简单的数据类型是布尔型。有时候,您需要知道某种东西是开还是关、真还是假、是还是否。布尔型数字通常被设置为两个值之一: 0 或 1。

C#中的布尔型数据类型为 bool。从程序清单 3.4 的输出中可以知道, bool 使用 1 个字节的内存。bool 变量的取值只能是 true 或 false, true 或 false 是 C#关键字。这意味着 bool 数据类型只能存储 true 或 false。

警告: “Yes”、“no”、“on”、“off”都不是 C#关键字,这意味着您不能将布尔型变量设置为这些值,而必须设置为 true 或 false。

3.5.5 检查和不检查

通过本章前面的内容,您知道如果将一个过大的值赋给变量,将出错。有时候,您可能不希望出现这种错误。在这种情况下,您可以让编译器不对代码进行检查。这是通过关键字 unchecked 实现的,程序清单 3.8 对此进行了演示。

程序清单 3.8 unchecked.cs: 命令编译器不对代码进行检查

```
1: // unchecked.cs
2: //-----
3:
4: using System;
5:
6: class sizes
7: {
8:     public static void Main()
9:     {
10:         int val1 = 2147483647;
11:         int val2;
12:
13:         unchecked
14:         {
15:             val2 = val1 + 1;
16:         }
17:
```

```
18:      Console.WriteLine( "val1 is {0}", val1);
19:      Console.WriteLine( "val2 is {0}", val2);
20:  }
21: }
```

该程序清单的输出如下：

```
var1 is 2147483647
var2 is -2147483648
```

分析：该程序清单的第13行使用了unchecked，编译器将不检查第14行和第16行的花括号之间的代码，编译该程序清单时，不会出现错误。运行该程序清单时，将得到一个怪异的结果。2147483647是带符号int变量能够存储的最大值，第10行将这个值赋给了var1。第15行（不被检查的行）将var1能够存储的最大值与1相加，并将结果赋给var2。由于该行不会被检查，因此程序将继续运行。结果是，存储在var2中的值为最小的负数。

这类似于汽车中的里程表，当里程达到最大（如 999999）时，再运行 1 公里后，里程表的值便变成 000000 了。此时，汽车并非一辆新车，而是一辆里程超过里程表最大值的汽车。在这种情况下，变量不会变为 0，而是它能够存储的最小值。在上述程序清单中，值变为-2147483648。

请将程序清单的第 13 行修改为如下所示，然后编译并运行它。

```
13:      checked
```

程序将被编译，但能否运行呢？执行该程序时将出错。如果系统询问是否运行调试器，请不要运行。出现的错误与下面类似：

```
Exception occurred: System.OverflowException: An exception of type System.OverflowException
was thrown.
at sizes.Main( )
```

本书后面将介绍如何处理程序中的这种错误。现在您只需记住，如果您认为有可能赋给变量一个非法值时，应启用检查功能。

3.5.6 简单数据类型

前面介绍的 C#数据类型被称为简单数据类型。简单数据类型包括 sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、bool 和 decimal。第 1 天和第 2 天的课程介绍过，C#程序运行在 CLR 上。这些数据类型中的每一种直接对应于 CLR 使用一种数据类型，它们之所以被称为简单数据类型是因为，它们与 CLR 中的类型有直接关系，从而与 .NET 框架中的类型有直接关系。表 3.3 列出了 C#数据类型对应的 .NET 数据类型。

表 3.3 C#和.NET 数据类型

C#数据类型	.NET 数据类型
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16

续表

C#数据类型	.NET 数据类型
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

如果要使用对应的.NET 数据类型声明一个变量（虽然没有理由这样做），则可以使用下面的代码：

```
System.Int32 my_variable = 5;
```

从中可以知道，System.Int32 比使用 int 要复杂得多。程序清单 3.9 演示了如何使用.NET 数据类型。

程序清单 3.9 net_vars.cs: 使用.NET 数据类型

```
1: // net_vars
2: // Using a .NET data declaration
3: //-----
4:
5: using System;
6:
7: class net_vars
8: {
9:     public static void Main()
10:    {
11:
12:        System.Int32 my_variable = 4;
13:        System.Double PI = 3.1459;
14:
15:        Console.WriteLine("\nmy_variable is {0}", my_variable );
16:        Console.WriteLine("\nPI is {0}", PI );
17:    }
18: }
```

该程序清单的输出如下：

```
my_variable is 4
```

```
PI is 3.14159
```

第 12 行和 13 行分别声明了一个 `int` 和 `double` 变量。第 15 和 16 行打印了这些变量的值。`.NET` 类型的所有功能都可以使用更简单的 `C#` 命令实现。但需要明白的是，简单的 `C#` 数据类型将被转换为对应的 `.NET` 类型。您将发现，使用 `.NET` 类型的其他所有编程语言也包含将被转换为 `.NET` 类型的数据类型。

注意：通用类型系统（Common Type System，或称 CTS）是一套 CLR 中的数据类型都必须遵守的规则。`C#` 中的简单数据类型和 `.NET` 数据类型都遵守这些规则。如果某种语言在创建数据类型时遵守了 CTS，则它创建和存储的数据将能够与其他也遵循了 CTS 的编程语言兼容。

3.6 字面值和变量

新术语：在至今为止介绍的范例中，使用了大量并非变量的数字和值。您经常需要在源代码中输入数字或值。在源代码中，**字面值**代表的是其本身。例如，在下面的代码行中，数字 10 和字符串“Bob is a fish”都是字面值。正如您看到的，字面值可以赋给变量。

```
int x = 10;
myStringValue = "Bob is a fish";
```

3.6.1 数值型字面值

前面的很多范例使用过数值型字面值。默认情况下，数值型字面值的数据类型为 `int` 或 `double`；如果是整数，则为 `int` 型，如果是浮点数，则为 `double` 型。例如下面的代码：

```
nbr = 100;
```

这里，100 是一个数值型字面值。默认情况下，其数据类型被视为 `int`，而不管变量 `nbr` 是什么数据类型。现在来看下面的例子：

```
nbr = 99.9;
```

在这个例子中，99.9 也是一个数值型字面值，但其数据类型默认为 `double`，而不管 `nbr` 是什么数据类型。虽然 99.9 可作为 `float` 类型存储。在下面的代码行中，100. 的数据类型是 `int` 还是 `double` 呢？

```
x = 100.;
```

这是一个比较困难的问题。如果您的猜测是 `int`，则是错误的。由于 100. 中包含小数点，因此它是一个 `double`。

3.6.1.1 整型字面值的默认类型

当您使用整型值时，系统将根据其大小将其视为 `int`、`uint`、`long` 或 `ulong`。如果该值可以以 `uint` 或 `int` 保存，则被视为相应的数据类型。否则，将被视为 `long` 或 `ulong` 类型。要指定字面值的数据类型，可以使用字面值后缀。例如，要将数字 10 指定为一个 `long` 型的字面值（有符号或无符号），可以这样编写代码：

```
10L;
```

可以使用 `u` 或 `U` 来指定无符号型的值。如果要指定一个无符号长整型值，可以结合使用两个后缀：`ul`。

注意：如果您使用小写的`l`来声明`long`字面值，微软公司的C#编译器将提出警告：该编译器提出警告是提醒您，小写`l`容易与数字`1`混淆。

3.6.1.2 浮点数字面值的默认类型

正如前面指出的，默认情况下，浮点数字面值被视为 `double` 类型。要将字面值指定为 `float` 类型，可以在数字后面加上 `f` 或 `F`。例如，要将数字 4.4 赋给 `float` 变量 `my_float`，可以使用下面的方法：

```
my_float = 4.4f;
```

要将字面值指定为 `decimal` 类型，可以使用后缀 `m` 或 `M`。例如，下面的代码行声明 `my_decimal` 的值等于 `decimal` 型数值 1.32。

```
my_decimal = 1.32m;
```

3.6.2 布尔型字面值

布尔型字面值已经介绍过了，`true` 和 `false` 都是字面值，同时它们也是 C# 关键字。

3.6.3 字符串字面值

将字符组合在一起，可以形成单词、短语和句子。在编程用语中，一组字符被称为字符串。字符串是可以识别出来的，因为它们位于一对双引号之间。以前的例子中也使用过字符串，例如 `Console.WriteLine` 例程便是用了 一个字符串。字符串字面值指的是一对双引号之间的一组字符。下面都是字符串：

```
"Hello, World!"  
"My Name is Bradley"  
"1234567890"
```

最后一个例子之所以是字符串字面值，而不是数值型字面值，是因为这些数字位于引号之间。

注意：在字符串中可以使用表 3.3 中的任何特殊字符

3.7 常 量

除了使用字面值外，有时候您可能想将一个值存储到变量中，并固定它。例如，如果您声明了一个名为 `PI` 的变量，将其值设置为 3.14159，则您希望其值始终为 3.14159。没有理由再去修改它的值。另外，您还想防止其他人修改它。

要声明一个存储常量的变量，可以使用关键字 `const`。例如，要将 `PI` 声明为一个固定值，可以使用下面的代码：

```
const PI = 3.14159;
```

您可以在程序中使用 `PI`，但不能修改它的值。关键字 `const` 固定了其内容。可以针对任何数据类型的任何变量使用关键字 `const`。

提示：为使常量易于识别，可以将其名称中的字母全部采用大写。这样，将很容易知道某个变量的值是固定的。

3.8 引用类型

至此，已经介绍了很多数据类型。C#提供了两种主要的信息存储方式：按值（byval）和按引用（byref）。前面介绍的基本数据类型都是按值来存储信息的。

当变量按值存储信息时，变量将包含实际的信息。例如，当您将 123 存储到变量 x 中后，x 的值便是 123。变量 x 实际上包含了值 123。

按引用存储信息要稍微复杂些。如果变量按引用存储，而不是存储信息本身，则变量存储的将是信息所在的位置。换句话说，它存储的是到信息的引用。例如，如果 x 是一个“按引用”变量，则它包含的将是关于值 123 所在位置的信息，它并不存储值 123 本身。图 3.1 说明了这种差别。

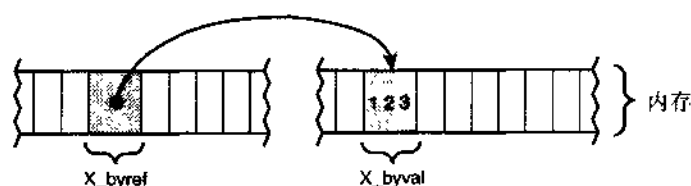


图 3.1 按引用和按值

在 C# 使用的数据类型中，按引用存储的有：

- 类；
- 字符串；
- 接口；
- 数组；
- 代表（Delegate）。

本书后面将详细介绍上述所有数据类型。

3.9 总 结

今天的课程介绍了计算机如何存储信息。重点介绍了按值存储数据的数据类型，包括 int、uint、long、ulong、bool、char、short、ushort、float、double、decimal、byte 和 sbyte。除此之外，还介绍了如何声明和命名变量以及如何设置变量的值，包括使用字面值。表 3.4 列出了这些数据类型及其相关的信息。

表 3.4 C#数据类型

C#数据类型	对应的 .NET 数据类型	占用的字节数	最小值	最大值
sbyte	System.SByte	1	-128	127
byte	System.Byte	1	0	255
short	System.Int16	2	-32768	32767
ushort	System.UInt16	2	0	65535
int	System.Int32	4	-2147483648	2147483647

续表

C#数据类型	对应的.NET 数据类型	占用的字节数	最小值	最大值
uint	System.UInt32	4	0	4294967295
long	System.Int64	8	-9223372036854775808	9223372036854775807
ulong	System.UInt64	8	0	18446744073709551615
char	System.Char	2	0	65535
float	System.Single	4	1.5×10^{-45}	3.4×10^{38}
double	System.Double	8	5.0×10^{-324}	1.7×10^{4932}
bool	System.Boolean	1	false(0)	true(1)
decimal	System.Decimal	16	1.0×10^{-28}	约 7.9×10^{28}

3.10 问与答

问：为何不将所有的数值都声明为较大的数据类型，而是较小的数据类型？

答：虽然使用较大的数据类型好像是合理的，但这样做效率不高。您使用的系统资源（内存）不应该超过需求。

问：将一个负数赋给一个无符号变量将出现什么情况？

答：如果使用的是字面值将出错，编译器指出您不能将负数赋给无符号变量；如果是由于计算而导致无符号变量的值小于0，得到的结果将是错误的。本书后面将介绍如何检查这种错误值。

问：decimal 值的精度高于 float 和 double 值，在不同的数据类型之间进行转换时，将如何进行舍入？

答：将 float、double 或 decimal 转换为某种整型变量类型时，值将被舍入。如果数字过大，无法存储到变量中，将出错。

将 double 数据转换为 float 型时，如果数据过大或过小，则将分别表示为无穷大和0。

将 float 或 double 数值转换为 decimal 型时，数值将被舍入。只有28位小数之后的值才会被舍入，并且只有在必要时才会舍入。如果被转换的值太小，无法表示为 decimal，则转换后的值将为0；如果值太大，无法以 decimal 存储，则将发生错误。

将 decimal 值转换为的 float 或 double 时，将被舍入为与之最接近的 float 或 double 值。记住，decimal 的精度高于 float 或 double，转换后，精度将降低。

问：还有哪些语言也遵循 CLR 中的 CTS？

答：Microsoft Visual Basic.NET（第7版）和 Microsoft Visual C++.NET（第7版）都支持 CTS。另外，许多语言的版本也支持 CTS，包括 Python、COBOL、Perl、Java 等等。有关支持 CTS 的其他语言，请查阅微软公司的网站。

3.11 作业

下面的小测验帮助您巩固所学的知识, 练习则让您实际应用所学的知识。在阅读下一课时之前, 应尽可能理解这些小测验和练习的答案, 答案见附录 A。

3.11.1 小测验

1. C#中的按值数据类型有哪些?
2. 带符号变量和无符号变量之间的区别何在?
3. 要存储数值 55, 可使用的最小数据类型是什么?
4. short 变量能够存储的最大值是多少?
5. 字符 B 对应的数值是什么?
6. 一个字节包含多少位?
7. 哪些字面值可以被赋给布尔型变量?
8. 指出三种引用数据类型。
9. 哪种浮点数据类型的精度最高?
10. 在.NET 中, 与 C#中的 int 数据类型对应的是什么数据类型?

3.11.2 练习

1. 修改程序清单 3.6 的取值范围, 以打印小写字母。
2. 编写一行代码, 声明一个名为 xyz 的 float 变量, 并将值 123.456 赋给该变量。
3. 下述哪些变量名是合法的?
 - a) x
 - b) PI
 - c) 12months
 - d) sizeof
 - e) nine
4. 下面的程序有问题, 请在编辑器中输入该程序, 并编译它。哪一行会导致错误消息?

```

1: //Bug Buster
2: //-----
3: using System;
4:
5: class variables
6: {
7:     public static void Main()
8:     {
9:         double my_double;
10:        decimal my_decimal;
11:
12:        my_double = 3.14;
13:        my_decimal = 3.14;
14:
15:        Console.WriteLine("\nMy Double: {0}", my_double);
16:        Console.WriteLine("\nMy Decimal: {0}", my_decimal);
17:

```



```
18:     }  
19: }
```

5. 选做题: 编写一个程序, 其中对于每种数据类型, 都声明两个变量, 并分别将值 10 和 1.879 赋给它们。

第 4 天课程

使用运算符

知道如何在变量中存储信息后,您可能想使用其中的数据。您很可能需要通过修改来操纵数据。例如,您可能需要使用圆的半径来计算出圆的面积。今天将学习以下内容:

- C#中运算符的类型和分类;
- 使用不同的运算符来操纵信息;
- 使用 if 命令修改程序的流程;
- 了解运算符的优先级;
- 探索按位运算——如果您足够勇敢的话。

4.1 运算符的类型

运算符用于操纵信息。在前几天的范例中,您已经使用过运算符。运算符用于执行诸如加法、乘法、比较等运算。

运算符可以分成很多类:

- 基本的赋值运算符;
- 数学/算术运算符;
- 关系运算符;
- 条件运算符;
- 其他运算符(类型、大小)。

今天将详细介绍各类运算符以及其中的每个运算符。除了运算符类别外,理解运算符语句的结构也很重要。运算符结构有三种:

- 单目;
- 双目;
- 三目。

4.1.1 单目运算符

单目运算符指的是只影响一个变量的运算符。例如,要将 1 变成负值,您输入如下代码:

```
-1
```

如果有一个名为 x 的变量,要将其变成负的,可以使用下面的代码:

-x

负运算符只需要一个变量，因此是单目的。单目运算符的使用格式为下面两种之一，具体使用哪种格式取决于运算符：

```
[operator][variable]
或
[variable][operator]
```

4.1.2 双目运算符

单目运算符只使用一个变量，而双目运算符使用两个变量。例如，加法运算符用于将两个值相加起来。双目运算符的使用格式如下：

```
[variable1][operator][variable2]
```

下面是一些双目运算的例子：

```
5 + 4
3 - 2
100.4 - 92348.67
```

您将发现，大多数运算符都是双目运算符。

4.1.3 三目运算符

三目运算符是使用起来最为复杂的运算符。顾名思义，这种运算符使用三个变量。C#只有一个真正的三目运算符——条件运算符。今天的课程后面将介绍这种运算符。现在，只需知道三目运算符使用三个变量即可。

4.2 标点符号

介绍 C#的各类运算符以及具体的运算符之前，了解标点符号至关重要。标点符号是一种特殊的运算符，帮助您格式化代码，同时执行多个操作，并给编译器提供信息。您需要了解的标点符号有：

- 分号：分号的主要用途是结束每一条 C#语句，分号也可用于一些控制程序流程的 C#语句中。第 5 天的课程将介绍分号在控制语句中的用途。

- 逗号：用于将多个命令放在同一行中。在第 3 天的课程的许多例子中使用过逗号。逗号最常见的用途是用来声明同一类型的多个变量：

```
int var1, var2, var3;
```

- 圆括号：可用于多种地方。在本章后面您将看到使用圆括号来强制改变执行顺序的情况。另外，圆括号还被用于函数中。

- 花括号：花括号用于代码片段分组。前面的很多例子使用了花括号来括住类。您应该注意到了，花括号总是成对出现。

C#中标点符号的工作原理与自然语言中句子中的标点相同。例如，句子以句号或其他标点结束，而在 C#中，代码行以分号或其他标点符号结束。行是一种抽象的概念，因为在源代码清单中，一行

代码可能占用多行。正如第 2 天的课程介绍的，空白和换行符将被忽略。

注意：您将发现，也可以在例程中使用花括号，将代码分块。花括号之间的代码和花括号本身被称为代码块。

4.3 基本的赋值运算符

您首先需要了解的运算符是基本的赋值运算符，它是一个等号。在前面课程的许多例子中已经使用过该运算符。

基本赋值运算符用于赋值。例如，要将 142 赋给变量 `x`，可以使用下面的代码：

```
x = 142;
```

这样，编译器将把赋值运算符右边的值放到左边的变量中。请看下面的代码：

```
x = y = 123;
```

这看起来有些怪异，但却是合法的 C# 代码。这里，最右边的值 123 将被放到变量 `y` 中，然后 `y` 的值将被放到变量 `x` 中。最后，`x` 和 `y` 的值都将为 123。

警告：不能在赋值运算符左边进行运算，例如，不能编写这样的代码：

```
1 + x = y;
```

同时，也不能将宇面值或常量放在赋值运算符的左边。

4.4 数学/算术运算符

数学运算符是最常用的运算之一。在 C# 中，可以执行所有的基本数学运算，包括加、减、乘、除和求模。另外，还有复合运算符，使得执行某些运算更为简洁。

注意：求模运算符也叫求余运算符（*remaindering operator*）。

4.4.1 加减

C# 中的加法运算符用来执行加减运算。对于加法，使用加号；对于减法，则使用减号。使用这些运算符的格式如下：

```
NewVal = Val1 + Val2;  
NewVal = Val1 - Val2;
```

在第一条语句中，`Val2` 和 `Val1` 相加，得到的结果被放到 `NewVal` 中；这种运算执行完毕后，`Val1` 和 `Val2` 的值保持不变。`NewVal` 中原来的值将被相加得到的结果所覆盖。

对于减法语句，则将 `Val1` 减去 `Val2`，并将得到的结果放到 `NewVal` 中。同样，`Val1` 和 `Val2` 的值也将保持不变，而 `NewVal` 中原来的值将被相减得到的结果所覆盖。

`Val1` 和 `Val2` 可以是任何数据类型的值、常量和宇面值，而 `NewVal` 必须是变量，不过可以是与 `Val1` 或 `Val2` 相同的变量。例如，只要 `Var1` 是变量，则下面的语句便是合法的：

```
Var1 = Var1 - Var2
```

上述范例从 `Var1` 中减去 `Var2`，并将结果放到 `Var1` 中，覆盖 `Var1` 原来的值。下面的范例也是

合法的:

```
Var1 = Var1 - Var1;
```

这里, 将 Var1 减去 Var1, 由于这两个值相同, 因此结果为 0。然后将结果 0 放到 Var1 中, 覆盖该变量原来的值。

如果要将某个值翻番, 可以使用下面的语句:

```
Var1 = Var1 + Var1;
```

Var1 与其本身相加, 得到的结果被放回到 Var1 中。最终结果是, Var1 的值翻番了。

4.4.2 乘法运算符

将变量的值翻番的最简单的方法是将其乘以 2。C# 中常用的乘法运算符有三个。

4.4.2.1 乘法运算

执行乘法的运算符是星号 (*), 它将两个值相乘, 格式如下:

```
NewVal = Val1 * Val2;
```

例如, 要将 Val1 的值翻番, 并将得到的结果放回到 Val1 中 (见最后一个关于加法的例子), 可以使用下面的语句:

```
Val1 = Val1 * 2;
```

上述语句与下面的语句等效:

```
Val1 = 2 * Val1;
```

4.4.2.2 除法运算

要执行除法运算, 可使用除法运算符。它是一个斜杠 (/):

```
NewVal = Val1 / Val2;
```

上述范例将 Val1 除以 Val2, 并将得到的结果放到 NewVal 中。要将 2 除以 3, 可使用下面的代码:

```
answer = 2 / 3;
```

4.4.2.3 求余

有时候, 执行除法运算时, 您只想知道余数是多少。例如, 我知道 4 是 3 的一倍多, 但我还可能想知道 4 除以 3 得到的余数是 1。可以使用求余 (也叫求模) 运算符 (百分比符号 %) 来计算余数。例如, 要计算 4 除以 3 得到的余数, 可以使用下面的代码:

```
Val = 4 % 3;
```

结果是, Val 的值为 1。

请看另一个例子。有三个馅饼和 13 个人, 每个馅饼都被切成 6 块, 而每个人吃一块馅饼。请问还余下多少块馅饼。

要解决这种问题, 首先需要知道有多少块馅饼:

```
PiecesOfPie = 3 * 6 //three pies times six pieces
```

显然, PiecesOfPie 的值为 18。现在使用求模运算符来计算余下多少块馅饼:

```
PiecesForMe = PiecesOfPie % 13;
```

上述语句将 `PiecesForMe` 的值设置为 18 除以 13 的余数——5。程序清单 4.1 验证了这一点。

程序清单 4.1 pie.cs: 余下多少块馅饼

```
1: // pie.cs - Using the modulus operators
2: //-----
3: class pie
4: {
5:     static void Main()
6:     {
7:         int PiecesForMe = 0;
8:         int PiecesOfPie = 0;
9:
10:        PiecesOfPie = 3 * 6;
11:
12:        PiecesForMe = PiecesOfPie % 13;
13:
14:        System.Console.WriteLine("Pieces Of Pie = {0}", PiecesOfPie);
15:        System.Console.WriteLine("Pieces For Me = {0}", PiecesForMe);
16:    }
17: }
```

该程序清单的输出如下：

```
Pieces Of Pie = 18
Pieces For Me = 5
```

分析：该程序清单演示了乘法运算符和求模运算符的用法。第10行使用前面列出的公式说明了乘法运算符，而第12行则使用了求模运算符。从第14和15行打印的信息可以知道，结果同预期的完全相同。

4.4.2.4 算术赋值复合运算符

本章前面介绍过基本的赋值运算符，还有其他的赋值运算符——复合赋值运算符（如表 4.1 所示）。

表 4.1 算术赋值复合运算符

运 算 符	描 述	对应的非复合表达式
<code>+=</code>	<code>x += 4</code>	<code>x = x + 4</code>
<code>-=</code>	<code>x -= 4</code>	<code>x = x - 4</code>
<code>*=</code>	<code>x *= 4</code>	<code>x = x * 4</code>
<code>/=</code>	<code>x /= 4</code>	<code>x = x / 4</code>
<code>%=</code>	<code>x %= 4</code>	<code>x = x % 4</code>

复合运算符提供了一种更为简洁的、执行数学运算并进行赋值的方式。如果要将某个变量的值

增加 5，可以使用下面的语句：

```
x = x + 5;
```

也可以使用复合运算符：

```
x += 5;
```

从上面可以知道，复合运算更为简洁。

提示：虽然复合运算符更简洁，但并非总是最容易理解的。使用复合运算符时，必须清晰地指出要做什么，即别忘了对代码进行注释。

4.4.3 执行单目数学运算

到目前为止，您看到的所有算符运算符都是双目的，需要两个操作数。也有很多单目运算符，只需要一个值或变量。单目算术运算符包括递增运算符（++）和递减运算符（--）。

这两个运算符分别将变量的值加 1 和减 1。下面的语句将 x 的值加 1：

```
++x;
```

其功能与下述代码等效：

```
x = x + 1;
```

而下面的语句将 x 的值减 1

```
--x;
```

其功能与下述代码等效：

```
x = x - 1;
```

程序清单 4.2 演示了递增和递减运算符的用法。

提示：当您需逐个遍历大量的值时，使用递增运算符和递减运算符非常方便。

程序清单 4.2 cout.cs: 递增运算符和递减运算符

```
1: // count.cs - Using the increment/decrement operators
2: //-----
3:
4: class count
5: {
6:     static void Main()
7:     {
8:         int Val1 = 0;
9:         int Val2 = 0;
10:
11:         System.Console.WriteLine("Val1 = {0} Val2 = {1}", Val1, Val2);
12:
13:         ++Val1;
14:         --Val2;
15:
16:         System.Console.WriteLine("Val1 = {0} Val2 = {1}", Val1, Val2);
17:
```

```

18:      ++Val1;
19:      --Val2;
20:
21:      System.Console.WriteLine("Val1 = {0} Val2 = {1}", Val1, Val2);
22:  }
23: }

```

该程序清单的输出如下所示:

```

Val1 = 0 Val2 = 0
Val1 = 1 Val2 = -1
Val1 = 2 Val2 = -2

```

分析: 该程序清单并没有完成什么壮观的工作,但说明了递增运算符和递减运算符。在第8行和第9行,两个变量被初始化为0。第11行打印这两个变量的值,以便您知道它们的值为0。第13行和第14行分别将这两个变量的值加1和减1,第16行打印它们的值,以便您能知道结果。第18~21行重复上述操作。

4.4.3.1 使用先递增和后递增运算符

递增运算符和递减运算符有独特的特征,对于新的程序员,这种特征经常会引起一些问题。假设 x 的值为 10,请看下行代码:

```
y = ++x;
```

上述语句执行后, x 和 y 的值分别是多少呢?您可能猜对了, x 和 y 的值都将是 11。现在来看下面的代码,同样假设 x 的当前值为 10:

```
y = x++;
```

上述语句执行后, x 和 y 的值又将分别是多少呢?如果您认为它们的值也将都是 11,则是不正确的。上述代码行执行后, x 的值将为 11,但 y 的值将是 10。糊涂了吧?

很简单,递增运算符可以以先递增和后递增的方式执行运算。先递增时,递增运算将首先被执行;后递增时,递增运算将最后被执行。那么,如何知道是先递增还是后递增呢?很简单,如果运算符位于变量的前面($++x$),则先递增;如果位于变量的后面($x++$),则后递增。递减运算符也是如此。程序清单 4.3 演示了递增运算符和递减运算符的先运算和后运算的情况。

程序清单 4.3 prepost.cs: 使用单目递增运算符

```

1:  // prepost.cs - Using pre- versus post-increment operators
2:  //-----
3:
4:  class prepost
5:  {
6:      static void Main()
7:      {
8:          int Val1 = 0;
9:          int Val2 = 0;
10:
11:          System.Console.WriteLine("Val1 = {0} Val2 = {1}", Val1, Val2);
12:

```



```

13:      System.Console.WriteLine("Val1 (Pre) = {0} Val2 = (Post) {1}",
14:          ++Val1, Val2++);
15:
16:      System.Console.WriteLine("Val1 (Pre) = {0} Val2 = (Post) {1}",
17:          ++Val1, Val2++);
18:
19:      System.Console.WriteLine("Val1 (Pre) = {0} Val2 = (Post) {1}",
20:          ++Val1, Val2++);
21:  }
22: }

```

该程序清单的输出如下：

```

Val1 = 0 Val2 = 0
Val1 (Pre) = 1 Val2 = (Post) 0
Val1 (Pre) = 2 Val2 = (Post) 1
Val1 (Pre) = 3 Val2 = (Post) 2

```

分析：理解程序清单4.3中发生的情况很重要。第8和9行同样将两个变量初始化为0。第11行打印了这些变量的值。从输出中可以知道，Val1和Val2的值都为0。第13和14行再次打印了这两个变量的值，这里打印的分别是++Val1和Val2++，即对Val1执行先递增运算，而对Val2执行后递增运算。从输出中可以知道，Val1的值被加1，然后被打印出来；而Val2首先被打印出来，然后被加1。第16~19行重复执行上述操作两次。

应该	不应该
应使用复合运算符来使数学例程更为简洁	不要混淆了先递增运算符和后递增运算符。前者首先将变量的值加1，而后者最后才将变量的值加1

4.5 关系运算符

生活中总是充满问题。除了提出问题外，对事情进行比较常常也很重要。在编程中，您将对值进行比较，然后根据结果执行相应的代码。关系运算符用于对两个值进行比较。

使用关系运算符，可以确定两个值之间的关系。表4.2列出了关系运算符。

表 4.2

关系运算符

运 算 符	描 述
>	大于
<	小于
==	等于
!=	不等于
>=	大于等于
<=	小于等于

使用关系运算符进行比较时，将得到两个结果之一：真或假。请看下面使用关系运算符进行的比较：

- $5 < 10$: 5 小于 10，因此结果为真；
- $5 > 10$: 5 不大于 10，因此结果为假；
- $5 == 10$: 5 不等于 10，因此结果为假；
- $5 != 10$: 5 不等于 10，因此结果为真。

正如您看到的，这些结果要么为真，要么为假。知道可以检查两个值之间的关系，对于编程很重要。问题是，您如何使用这些关系？

4.5.1 if 语句

关系运算的结果可用于决策，这种决策用于改变程序的执行流程。可以结合使用 if 关键字和关系运算符来改变程序的流程。

If 关键字用于根据两个值的比较结果来决定程序流程。if 命令的标准格式如下：

```
if (val1 [operator] val2)
    statements(s);
```

其中，operator 是关系运算符，val1 和 val2 是变量、常量或字面值；而 statement(s) 是单条语句或包括多条语句的代码块。代码块指的是包含在花括号中的一条或多条语句。

如果 val1 和 val2 的比较结果为真，后面的语句将被执行；否则将不被执行。图 4.1 说明了 if 语句的工作原理。

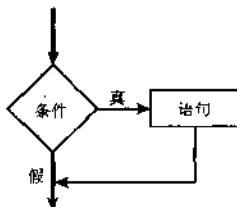


图 4.1 if 命令

在一个范例中使用 if 语句，可以更清楚地进行说明。程序清单 4.4 便是一个说明 if 语句用法的简单范例。

程序清单 4.4 iftest.cs: 使用 if 语句

```
1: // iftest.cs- The if statement
2: //-----
3:
4: class iftest
5: {
6:     static void Main()
7:     {
8:         int Val1 = 1;
9:         int Val2 = 0;
10:
11:         System.Console.WriteLine("Getting ready to do the if...");
12:
```

```

13:     if (Val1 == Val2)
14:     {
15:         System.Console.WriteLine("If condition was true");
16:     }
17:     System.Console.WriteLine("Done with the if statement");
18: }
19: }

```

该程序清单的输出如下：

```

Getting ready to do the if...
Done with the if statement

```

分析：该程序清单的第13行使用if语句比较两个值是否相等。如果相等，则执行第15行的打印语句；否则跳过第15行。由于在第8行和第9行给Val1和Val2指定的值不相等，因此if条件不满足，所以不执行第15行。

将第 13 行作如下修改：

```
if (Val1 !=Val2)
```

则由于 Val1 和 Val2 确实不相等，因此if条件的结果为真。所以程序清单的输出将如下所示：

```

Getting ready to do the if...
If condition was true
Done with the if statement

```

警告：if语句的第一行不包括分号。例如下面的代码是不正确的：

```

if( x != x);
{
    //Statements to do when the if evaluates to true (which will never happen)
}

```

x肯定与其本身相等，因此 $x \neq x$ 的结果将为假，花括号中的语句应该不会被执行。但由于第一行的后面有分号，这表明if语句至此便结束了，这意味着花括号中的语句将被执行。不管if条件的结果是true还是false（这里为false），这些语句都将被执行。一定不要犯这样的错误，即在if语句的第一行中加上分号

4.5.2 条件逻辑运算符

现实世界常常很复杂，很多情况下，需要做多次比较来确定是否执行某个代码块。例如，您可能在某个人的年龄不小于21岁，并且是女性时才执行某些代码。为此，需要在一条if语句中使用另一条if语句，伪代码如下：

```

if ( sex == false)
{
    if (age >=21)
    {
        // This person is a female that is 21 years old or older.
    }
}

```

有一种更简单的方式可以完成上述工作——使用条件逻辑运算符。

条件逻辑运算符让您能够使用关系运算符完成多种比较。您将使用的两个条件逻辑运算符是“和”运算符（&&）和“或”运算符（||）。

4.5.2.1 条件“和”运算符

有时候，需要验证多个条件是否都满足，前面的例子就属于这种情况。逻辑“和”运算符（&&）让您能够验证是否所有条件都满足。可以将上面的例子重新编写，如下所示：

```
if (sex == female && age >= 21)
{
    // This person is a female that is 21 years old or older.
}
```

实际上，可以在 if 语句中包含两个以上的关系。不管进行的比较有多少个，仅当用&&连接起来的所有比较结果都为真时，后面的语句块才会被执行。例如在下面的范例中，仅当 if 语句中的所有三个条件都满足时，statements 代码块才会被执行。如果其中的某个条件不满足，则 statements 代码块将被跳过。

```
if ( x < 5 && y < 10 && z > 10)
{
    // statements
}
```

4.5.2.2 条件“或”运算符

有时候，并不要求所有的条件都满足，而只需要其中的一个满足即可。例如，您可能在提供的日期为星期六或星期天时，执行某些代码。在这种情况下，可以使用条件“或”运算符（||）。下面的范例使用伪代码演示了如何使用该运算符：

```
if ( day equals Sunday OR day equals Saturday)
{
    // do statements
}
```

在这个例子中，当日期为星期六或星期天时，将执行后面的语句块。仅当其中的一个条件满足时，后面的语句块便将执行。程序清单 4.5 演示了逻辑“和”运算符和逻辑“或”运算符的用法。

程序清单 4.5 and.cs: 使用逻辑“和”及逻辑“或”运算符

```
1: // and.cs- Using the conditional AND and OR
2: //-----
3:
4: class andclass
5: {
6:     static void Main()
7:     {
8:         int day = 1;
9:         char sex = 'f';
10:
11:         System.Console.WriteLine("Starting tests... (day:{0},
12:             sex:{1})", day, sex );
13:
14:         if ( day >= 1 && day <= 7 )    //day from 1 to 7?
```

```

15:      {
16:          System.Console.WriteLine("Day is from 1 to 7");
17:      }
18:      if (sex == 'm' || sex == 'f') // Male or female?
19:      {
20:          System.Console.WriteLine("Sex is male or female.");
21:      }
22:
23:      System.Console.WriteLine("Done with the checks.");
24:  }
25: }

```

该程序清单的输出如下：

```

Starting tests...(day:1, sex:f)
Day is from 1 to 7
Sex is male or female.
Done with the checks.

```

分析：该程序清单演示了运算符 `&&` 和 `||` 的用法。第14行使用了“和”运算符 (`&&`)，要使该if语句的条件得以满足，变量`day`的值必须大于等于1，并小于等于7。如果`day`的值为1、2、3、4、5、6或7，则if条件将为真，第16行的代码被执行；否则，if条件为假，第16行将不会被执行。

第18行使用了“或”运算符 (`||`)。在这里，如果变量`sex`包含的是字符`m`或`f`，则第20行将被执行；否则被略过。

警告：请注意第18行的if条件，它检查变量包含的是不是字符`m`或`f`，这些字符是小写的，与相应的大写字母是有区别的。如果在第9行将变量`sex`的值设置为`F`或`M`，则第18行的if条件将不满足。

对第8和第9行的代码进行修改，再运行该程序清单。得到的输出将随您设置的值而异。例如，将其修改为如下所示：

```

8:      int day = 9;
9:      char sex = 'x';

```

则程序的运行结果将如下所示：

```

Starting tests...(day:9, sex:x)
Done with the checks.

```

有时候，需要同时使用运算符 `&&` 和 `||`。例如，您可能希望在某人年龄为不小于21岁，并当其性别为男性或女性时，执行某些代码。这可以通过同时使用运算符“和”与“或”来实现。但这样做时一定要小心。“和”运算符期望其两边的值都为`true`，而“或”运算符期望其两边的值之一为`true`。对于这个例子，您可能输入如下的代码来实现：

```

if ( age >= 21 && gender == male || gender == FEMALE)
    //statements

```

上述代码的运行结果将与您期望的不同。如果这个人的年龄不小于21岁，并为女性，则后面语句将不会执行。在这种情况下，由于`&&`右边的值为`false`，系统不再检查后面的条件，而认为整个if条件为`false`。要避免这种问题，可以使用括号强行改变语句的执行顺序。要得到期望的结果，可将上述代码修改为如下所示：

```
if ( age >= 21 && (gender == male || gender == FEMALE))
    //statements
```

括号最内面的代码总是首先执行。在这里，代码 `gender == male || gender == FEMALE` 将首先被执行，由于使用的是 `||`，因此只要某一边的值 `true`，则结果便为 `true`。然后这部分的结果为 `true`，将执行 `&&` 的左边的部分，检查年龄是否不小于 21 岁。如果年龄不小于 21 岁，则后面的语句将被执行。

提示：应使用括号来确保代码按您希望的次序执行。

应该	不应该
应使用括号来使复杂的数学和关系运算更易于理解	不要将赋值运算符(=)和关系运算符相等(==)混淆了

4.6 逻辑按位运算符

您可能需要用到另外三个逻辑运算符：逻辑按位运算符。虽然按位运算的用法不在本书的范围之内，但本章的最后还是包含了名为“给足够勇敢者”的一节。该节对按位运算、三个逻辑按位运算符和移位运算符进行了介绍。

之所以叫按位运算符，是因为它们是针对各个位进行运算的。位是单个存储单元，其中存储了开关值（0 或 1）。本章的最后一节将介绍如何使用按位运算符对位进行操纵。

4.7 类型运算符

在本书后面开始使用类和接口后，便需要用到类型运算符。不理解接口和类，将难以完全理解类型运算符。就现在而言，您只需知道后面将使用的这种运算符，它们是：

- `typeof`;
- `is`;
- `as`。

4.8 sizeof 运算符

本书前面已经使用过 `sizeof` 运算符，它用于确定值的长度。第 3 天的课程使用过该运算符。

警告：由于 `sizeof` 运算符直接操纵内存，因此应尽可能避免使用它。

4.9 条件运算符

C# 中有一个三目运算符：条件运算符。条件运算符的格式如下：

```
Condition ? if_true_statement : if_false_statement;
```

正如您看到的，两个符号将其中的三部分分开。第一部分是条件，它就像 `if` 语句中的条件一样，可以是结果为 `true` 或 `false` 的任何条件。

之后是一个问号，它将条件和第一条语句分开。如果条件为真，则执行第一条语句；否则执行

第二条语句。第一条语句和第二条语句之间用冒号隔开。程序清单4.6演示了条件运算符的用法。

条件运算符用于创建简洁的代码。如果 if 语句很简单，当其条件为 true 和 false 时，需要执行的语句都很简单，则可以使用条件运算符；否则不应使用。由于它只不过是一种简洁的 if 语句，因此应尽可能使用 if 语句本身。对于阅读您编写的代码的大多数人而言，if 语句的可读性更强，也更易于理解。

程序清单 4.6 cond.cs: 使用条件运算符

```
1: // cond.cs - The conditional operator
2: //-----
3:
4: class cond
5: {
6:     static void Main()
7:     {
8:         int Val1 = 1;
9:         int Val2 = 0;
10:        int result;
11:
12:        result = (Val1 == Val2) ? 1 : 0;
13:
14:        System.Console.WriteLine("The result is {0}", result);
15:    }
16: }
```

该程序清单的输出如下：

The result is 0

分析：该程序清单很简单。在第12行，条件运算符的执行结果被赋给变量result。然后第14行打印该变量的值。条件运算符检查Val1的值是否与Val2相等。由于1不等于0，因此条件不满足。对第8行进行修改，将Val2的值设置为1，再运行该程序清单，您将发现结果为1，而不是0，这是因为1和1相等（条件已满足）。

警告：条件运算符提供了一种实现 if 语句的简短方式。虽然它更为简洁，但并非总是更容易理解。使用条件运算符时，应确保代码不会更难以理解。

4.10 运算符优先级

今天的课程介绍了大量不同的运算符。您经常需要在同一条语句中使用其中的多个运算符。在这种情况下，将出现很多问题。请看下面的语句：

```
answer = 4 * 5 + 6 / 2 - 1;
```

answer 的值将是多少呢？如果您认为是 12，则是不正确的；如果您认为是 44，则也是错误的。正确的答案是 22。

新术语: 不同类型的运算符按一定的次序执行, 这种次序被称为**运算符优先级**。之所以使用“优先级”, 是由于某个运算符的优先级比另一些高。在上面的例子中, 乘除运算的优先级高于加减运算, 这意味着在执行加减运算之前, 首先执行4乘以5和6除以2的运算。

表 4.3 列出了所有的运算符, 位于同一行的所有运算符的优先级都相同。几乎在所有的情况下, 优先级相同的运算符执行的先后次序都对结果没有影响。例如, 对于 $5 * 4 / 10$, 无论是先执行 5 乘以 4 的运算, 还是先执行 4 除以 10 的运算, 结果都相同。

表 4.3 运算符优先级

优 先 级	运算符类型	运 算 符
1	主要运算符	<code>()</code> , <code>...</code> , <code>[]</code> , <code>x++</code> , <code>x--</code> , <code>new</code> , <code>typeof</code> , <code>sizeof</code> , <code>checked</code> , <code>unchecked</code>
2	单目运算符	<code>+</code> , <code>-</code> , <code>!</code> , <code>++x</code> , <code>--x</code>
3	乘除运算符	<code>*</code> , <code>/</code> , <code>%</code>
4	加减运算符	<code>+</code> , <code>-</code>
5	移位运算符	<code><<</code> , <code>>></code>
6	关系运算符	<code><</code> , <code>></code> , <code><=</code> , <code>>=</code> , <code>is</code>
7	相等运算符	<code>==</code> , <code>!=</code>
8	逻辑“和”	<code>&</code>
9	逻辑“异或”	<code>^</code>
10	逻辑“或”	<code> </code>
11	条件“和”	<code>&&</code>
12	条件“或”	<code> </code>
13	条件运算符	<code>?:</code>
14	赋值运算符	<code>=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> , <code>+=</code> , <code>-=</code> , <code><<=</code> , <code>>>=</code> , <code>&=</code> , <code>^=</code> , <code> =</code>

4.10.1 改变优先级次序

本章前面介绍过如何使用括号来改变优先级次序。由于括号的优先级比任何运算符都高, 因此其包含的任何运算符都将比外面的先执行。以上面的范例为例, 可以使用括号来先执行加减运算:

```
answer = 4 * (5 + 6) / (2 - 1);
```

现在, `answer` 的值将是多少呢? 由于括号内的运算先执行, 因此编译器首先将代码解析为:

```
answer = 4 * 11 / 1;
```

最终结果为 44。括号可以嵌套, 例如可以将代码写成如下所示:

```
answer = 4 * ((5 + 6) / (2 - 1));
```

编译器首先将上述代码解析为:


```
answer = 4 * (11 / 1);
```

然后再解析为：

```
answer = 4 * 11;
```

因此，`answer` 的最终值为 44。这里括号并没有导致不同的结果，但有时候确实会导致不同的结果。

4.11 转换数据类型

将某种类型变量的值赋给另一种类型的变量时，必须进行类型转换。另外，对两个不同数据类型的变量执行运算时，也可能需要进行类型转换。类型转换有两种：显式的和隐式的。

新术语：隐式转换自动进行，而不会出现错误。您在本章已经看到过很多这样的情况。但当隐式转换无法进行时，将出现什么情况呢？例如，将`long`变量的值赋给`int`变量时，将出现什么情况呢？

新术语：显式转换指的是强制转换数据的类型。就本章介绍的数据类型而言，最简单的显式转换方式是强制转换（`cast`），即强行将一种数据类型的值转换为另一种数据类型。强制转换的格式如下：

```
ToVariable = (datatype) FromVariable;
```

其中 `datatype` 是要将 `FromVariable` 转换成的数据类型。以将 `long` 变量转换为 `int` 为例，可以使用下面的语句：

```
int IntVariable = 0;
long LongVariable = 1234;
IntVariable = (int) LongVariable;
```

执行强制转换时，程序员负责确保变量能够存储转换后的值。如果目标变量无法保存转换后的值，则将进行裁剪或其他修改。很多情况下需要执行显式转换，表 4.4 列出了这些情况。

注意：就整体而言，显式转换包含隐式转换，即使在隐式转换能够进行的情况下，也可以进行强制转换。

表 4.4 必须执行显式转换的情况

原来的类型	目标类型
sbyte	byte, ushort, uint, ulong, char
byte	sbyte, char
short	sbyte, byte, ushort, uint, ulong, char
ushort	sbyte, byte, short, char
int	sbyte, byte, short, ushort, uint, ulong, char
uint	sbyte, byte, short, ushort, int, char
long	sbyte, byte, short, ushort, int, uint, ulong, char

续表

原来的类型	目标类型
ulong	sbyte, byte, short, ushort, int, uint, long, char
char	sbyte, byte, short
float	sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal
double	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal
decimal	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double

4.12 理解操作数提升

隐式转换还与运算数提升 (operand promotion) —— 自动将操作数从一种类型转换为另一种类型——相关。对两个变量执行基本的算术运算时, 执行运算之前, 它们将被转换为相同的数据类型。例如, 将 byte 变量和 int 变量相加时, byte 变量将首先被提升为整数, 然后执行相加运算。比 int 小的数值型变量将被提升为 int。在 int 之后, 提升的次序如下:

```
int
uint
long
ulong
float
double
decimal
```

4.13 给足够勇敢者

接下来的几节是为那些足够勇敢的读者提供的, 介绍了如何使用按位运算符, 包括移位运算符和逻辑按位运算符。要理解这些运算符的工作原理, 必须先了解变量实际上是如何存储的。

提示: 理解按位运算符和内存的工作方式很有用, 但对于理解 C# 而言并不重要。很多人将它看作是一个高级主题。

4.13.1 在内存中存储变量

要理解按位运算符, 必须首先理解位。在昨天关于数据类型的课程中介绍过, 存储不同的数据类型所需的位数不同。例如, char 占用两个字节, int 占用 4 个字节。另外, 各种数据类型都有可存储的最大值和最小值。

一个字节为 8 位; 两个字节为 16 位——8 的两倍; 4 个字节为 32 位。因此, 最关键的是理解位是什么。

新术语: 位只不过是一个内存存储单位, 就像灯泡一样, 可以是开或关。在磁介质上存储信息时, 位可以是正电荷或负电荷; 使用光介质时, 位可以是凸起或凹陷。无论是哪种情况, 都是一种

状态表示0，而另一种状态表示1。

由于1位只能存储1或0，因此其存储容量很有限。要存储大型值，必须使用一组位。例如，使用两位，可以存储4种不同的值：00、01、10和11。使用三位，则可以存储8种不同的值：000、001、010、011、100、101、110、111；使用4位，则可以存储16种不同的值。实际上， x 位可以存储 2^x 种不同的值，因此一个字节可以存储 2^8 ，即256种不同的值；而两个字节可以存储65536种不同的值。

转换这些0和1只不过是使用二进制而已。附录D详细介绍了如何使用二进制。就现在而言，只需要知道二进制是一种计数法。

您使用十进制来计数。十进制使用10个数字（0-9），而二进制使用两个数字。使用十进制计数时，您使用的是1、10、100、1000等等。例如13包含1个10和3个1；25包含2个10和5个1。

二进制的原理与此相同，只不过使用两个数字（0和1）而已。您使用的不是10和100等，而是1、2、4、8等等，其中的每个值都是2的某次方。1为2的0次方、2为2的1次方、4为2的2次方等等。图4.2对此做了说明。

10^3 ... 1000	10^2 100	10^1 10	10^0 1	} 十进制
2^4 ... 16	2^3 8	2^2 4	2^1 2	
			2^0 1	} 二进制

图 4.2 二进制和十进制的比较

使用二进制表示数字的原理与十进制相同。最右边一位的位置值为1，然后依次为2、4等等。请看下面的数字：

1101

要将二进制数转换为十进制，可以将各位的值和位置值相乘，然后将结果全部相加。例如，最右边的一列为1，然后依次为0、1和1，因此结果为：

$$1 + (0 * 2) + (1 * 4) + (1 * 8)$$

得到的十进制数为13。因此，二进制数1101对应的十进制数为13。也可以用同样的方法将十进制数转换为二进制。随着数字的增大，所需的位数增多。为简化问题，内存实际上被划分为8位的单元——字节。

4.13.2 移位运算符

C#中有两种移位运算符可用于操纵位。顾名思义，这些运算符执行移位操作。使用移位运算符>>可以将位右移，使用<<可以将位左移。这些运算符通过指定距离来移动变量中的位，其格式如下：

```
New_value = Value {shift-operator} number-of-positions;
```

其中，Value是字面值或变量，shift-operator是运算符>>或<<；而number-of-positions是移动距离。例如，如果在一个字节的内存中存储13，则其二进制表示为：

```
00001101
```

对其执行移位运算，可以修改其值。请看下面的例子：

```
00001101 >> 2
```

上述范例右移两位，得到的结果为：

```
00000011
```

其对应的十进制值为 3。也就是说， $13 \gg 2$ 等于 3。请看另一个例子：

```
00001101 << 8
```

上述范例左移 8 位，由于该值使用 1 个字节内存保存的，因此得到的结果为 0。

4.13.3 逻辑运算符

除了移位外，还可以对两个数的对应位进行组合。按位逻辑运算符有四种，如表 4.5 所示。

表 4.5 逻辑按位运算符

运 算 符	描 述
	逻辑“或”按位运算符
&	逻辑“和”按位运算符
^	逻辑“异或”按位运算符
~	逻辑“非”按位运算符

这些运算符用于将两个二进制的对应位组合起来，得到的结果各不相同。

4.13.3.1 逻辑“或”按位运算符

使用逻辑“或”按位运算符（|）将两个值组合起来时，得到的结果如下：

- 如果两个数相应的位都是 0，则结果为 0；
- 如果至少有一个数相应的位为 1，则结果为 1。

下面是对两个 1 字节值执行逻辑“或”运算的例子：

第一个值：00001111

第二个值：11001100

结果：11001111

4.13.3.2 逻辑“与”按位运算符

使用逻辑“与”按位运算符（&）将两个值组合起来时，得到的结果如下：

- 如果两个数相应的位都是 1，则结果为 1；
- 如果至少有一个数相应的位为 0，则结果为 0。

下面是对两个 1 字节值执行逻辑“与”运算的例子：

第一个值：00001111

第二个值：11001100

结果：00001100

4.13.3.3 逻辑“异或”按位运算符

使用逻辑“异或”按位运算符 (^) 将两个值组合起来时，得到的结果如下：

- 如果两个数相应的位相同，则结果为 0；
- 如果两个数相应的位不同，则结果为 1。

下面是对两个 1 字节值执行逻辑“异或”运算的例子：

第一个值：00001111

第二个值：11001100

结果：11000011

程序清单 4.7 演示了这三种逻辑按位运算符的用法。

程序清单 4.7 逻辑按位运算符

```

1: // bitwise.cs - Using the bitwise operators
2: //-----
3:
4: class bitwise
5: {
6:     static void Main()
7:     {
8:         int ValOne = 1;
9:         int ValZero = 0;
10:        int NewVal;
11:
12:        // Bitwise NOT Operator
13:
14:        NewVal = ValZero ^ ValZero;
15:        System.Console.WriteLine("\nThe NOT Operator: \n 0 ^ 0 = {0}", NewVal);
16:
17:        NewVal = ValZero ^ ValOne;
18:        System.Console.WriteLine(" 0 ^ 1 = {0}", NewVal);
19:
20:        NewVal = ValOne ^ ValZero;
21:        System.Console.WriteLine(" 1 ^ 0 = {0}", NewVal);
22:
23:        NewVal = ValOne ^ ValOne;
24:        System.Console.WriteLine(" 1 ^ 1 = {0}", NewVal);
25:
26:        // Bitwise AND Operator
27:
28:        NewVal = ValZero & ValZero;
29:        System.Console.WriteLine("\nThe AND Operator: \n 0 & 0 = {0}", NewVal);
30:
31:        NewVal = ValZero & ValOne;
32:        System.Console.WriteLine(" 0 & 1 = {0}", NewVal);
33:
34:        NewVal = ValOne & ValZero;
35:        System.Console.WriteLine(" 1 & 0 = {0}", NewVal);
36:

```

```
37:     NewVal = ValOne & ValOne;
38:     System.Console.WriteLine(" 1 & 1 = {0}", NewVal);
39:
40:     // Bitwise OR Operator
41:
42:     NewVal = ValZero | ValZero;
43:     System.Console.WriteLine("\nThe OR Operator: \n 0 | 0 = {0}", NewVal);
44:
45:     NewVal = ValZero | ValOne;
46:     System.Console.WriteLine(" 0 | 1 = {0}", NewVal);
47:
48:     NewVal = ValOne | ValZero;
49:     System.Console.WriteLine(" 1 | 0 = {0}", NewVal);
50:
51:     NewVal = ValOne | ValOne;
52:     System.Console.WriteLine(" 1 | 1 = {0}", NewVal);
53: }
54: }
```

该程序清单的运行结果如下:

The NOT Operator:

```
0 ^ 0 = 1
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
```

The AND Operator:

```
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
```

The OR Operator:

```
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1
```

分析: 该程序清单较长, 但对逻辑按位运算符进行了总结。第8行和第9行声明了两个变量, 并将其值分别设置为1和0, 然后使用它们来执行按位运算。每当执行完一种按位运算后, 便将结果显示到控制台。查看输出可以知道, 结果与前面各节描述的完全相同。

4.14 总 结

今天的课程介绍了大量关于运算符及其用法的知识。您应该知道了运算符类型(包括算术运算符、关系运算符、逻辑运算符和条件运算符)以及运算符的优先级次序。最后介绍了按位运算和按位运算符。

4.15 问与答

问：理解运算符和运算符优先级有多重要？

答：几乎每一个应用程序都要用到运算符，理解运算符优先级至关重要。如果不了解运算符的优先级，得到的结果将可能出乎您的意料。

问：今天的课程简要地介绍了二进制。理解二进制很重要吗？其他计数法是否也很重要？

答：虽然理解二进制并非生死攸关，但却很重要。在当今的计算机中，信息是以二进制格式存储的。不管表示方式是正电荷和负电荷、凸起和凹陷还是其他方式，所有的数据最终都是以二进制方式存储的。对您来说，了解二进制的原理后，理解这些实际存储的值将更容易。

除了二进制外，许多程序员还使用八进制和十六进制。八进制是一种基数为 8 的计数法；而十六进制是一种基数为 16 的计数法。附录 D 更详细地介绍了这些计数法。

4.16 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

4.16.1 小测验

1. 哪个字符用于执行乘法运算？
2. $10 \% 3$ 的结果为多少？
3. $10 + 3 * 2$ 的结果是多少？
4. 条件运算符是什么？
5. 哪个 C# 关键字可用于改变程序的流程？
6. 单目运算符和双目运算符的区别何在？
7. 显式数据类型转换和隐式数据类型转换之间有何区别？
8. 可以将 long 值转换为 int 值吗？
9. 条件运算有哪些可能的结果？
10. 位移运算符有何功能？

4.16.2 练习

请注意，并非对所有的练习，附录 A 都提供了答案。下面的练习帮助您应用今天所学的知识。

1. 下述运算的结果是多少？

$$2 + 6 * 3 + 5 - 2 * 4$$

2. 下述运算的结果是多少？

$$4 * (8 - 3 * 2) * (0 + 1) / 2$$

3. 编写一个程序，检查变量的值是否大于 65。如果是，则打印文本“The value is greater than 65!”。
4. 编写一个程序，检查一个字符变量的值是否为 t 或 T。
5. 编写一行代码，将 long 变量 MyLong 的值赋给 short 变量 MyShort。
6. 下面的程序有问题。在编辑器中输入该程序，然后编译之。哪些行会生成错误消息？是么

错误?

```
1: class exercise
2: {
3:     static void Main()
4:     {
5:         int value = 1,
6:
7:         if ( value > 100 );
8:         {
9:             System.Console.WriteLine("Number is greater than 100");
10:        }
11:    }
12: }
```

7. 编写一行代码，将 `int` 变量 `IntVal` 的值赋给 `short` 变量 `ShortVal`。
8. 编写一行代码，将 `decimal` 变量 `DecVal` 的值赋给 `long` 变量 `LongVal`。
9. 编写一行代码，将 `int` 变量 `ch` 的值赋给字符变量 `charVal`。

第 5 天课程

控制语句

前面 4 天的课程介绍了大量的内容, 包括如何存储信息、如何执行运算以及如何使用 `if` 语句来避免执行某些命令等。还介绍了一些使用 `if` 语句来控制流程的知识, 但您常常需要对程序流程有更大的控制权。本章包含以下内容:

- 用于控制流程的其他命令;
- 使用 `if` 语句完成更多的任务;
- 在多个选项之间进行切换;
- 重复执行语句块多次;
- 中断代码的重复执行。

5.1 控制程序流程

通过控制程序流程, 能够实现一些很有用的功能。编程时, 您可能需要以许多方式来改变流程。您将需要重复执行代码块多次、跳过某个代码块或选择执行多个不同代码块中的一个。无论您以何种方式改变程序的流程, `C#` 都有相应的选项供您完成相应的任务。改变流程的方法可以分为两类:

- 选择语句;
- 循环语句。

5.2 使用选择语句

选择语句让您能够根据条件的结果执行特定的代码块。本书前面介绍过的 `if` 语句就是一种选择语句, `switch` 语句也是。

5.2.1 再谈 `if` 语句

前面已经介绍过 `if` 语句, 但有必要重温一下。请看下面的范例:

```
if( gender == 'm' || gender == 'f' )
{
    System.Console.WriteLine("The gender is valid");
}
if( gender != 'm' && gender != 'f' )
```

```

{
    System.Console.WriteLine("The gender value, {0} is not valid", gender);
}

```

本例使用一个字符变量 `gender`。第一个 `if` 语句检查 `gender` 是否为 `m` 或 `f`，使用了前一天课程中介绍的“或”运算符。第二个 `if` 语句在 `gender` 既不为 `m` 也不为 `f` 时，打印一条错误消息，这个 `if` 语句是一个确保变量的值有效的例子，如果变量的值既不为 `m` 也不为 `f`，则打印一条错误消息。

查看这两条语句，可以发现它们并不是最佳的。和其他语言一样，C# 也提供了另一个用于 `if` 语句的关键字 `else`。`else` 关键字是专门用于 `if` 语句的，`if...else` 语句的格式如下：

```

if ( condition )
{
    // If condition is true, do these lines
}
else
{
    // If condition is false, do these lines
}
// code after if... statement

```

`else` 语句让您能够加入 `if` 语句的条件不满足时执行的代码。`if` 后面的语句块和 `else` 后面的语句块总有一个被执行，但不会都执行。其中的语句块之一执行后，程序将跳到 `if...else` 语句后的第 1 行代码处。

程序清单 5.1 使用 `if...else` 命令对上述检查 `gender` 值的代码做了修改。从中可以知道，其效率更高，更容易理解。

程序清单 5.1 ifelse.cs: 使用 `if...else` 命令

```

1: // ifelse.cs - Using the if...else statement
2: //-----
3:
4: class ifelse
5: {
6:     static void Main()
7:     {
8:         char gender = 'x';
9:
10:        if( gender == 'm' || gender == 'f' )
11:        {
12:            System.Console.WriteLine("The gender is valid");
13:        }
14:        else
15:        {
16:            System.Console.WriteLine("The gender value, {0}, is not valid", gender);
17:        }
18:        System.Console.WriteLine("The if statement is now over!");
18:    }
19: }

```

该程序清单的输出如下：

```
The gender value, x, is not valid
The if statement is now over!
```

分析：第8行声明了一个char类型的变量gender，并将其值设置为x。if语句是从第10行开始的第10行检查gender的值是否为m或f。如果是，则在第12行打印一条消息，指出gender的值是有效的；否则if条件不满足，控制权转到第14行的else语句。在这个例子中，gender的值为x，因此else命令被执行，即打印一条消息，指出gender的值是无效的。然后，控制权被转到if...else语句后的第一行，即第18行。

修改第8行，将gender的值设置为m或f，重新编译并执行该程序，将得到如下所示的输出：

```
The gender is valid
The if statement is now over!
```

警告：如果将gender的值设置为M或F，结果将如何呢？别忘了，C#是区分大小写的，大写字母与小写字母是不同的。

5.2.1.1 if语句的嵌套

新术语：嵌套指的是一个语句中包含另一个语句。几乎所有的C#流程命令都可以嵌套。

要嵌套if语句，只需将一个if语句放在另一个内面即可。可以在if部分进行嵌套，也可以在else部分进行嵌套。以检查gender值的范例为例，可以按下面的方式使语句的效率稍高些（其中粗体部分是被嵌套的if语句）：

```
if( gender == 'm' )
{
    // it is a male
}
else
{
    if ( gender == 'f' )
    {
        // it is a female
    }
    else
    {
        //neither a male or a female
    }
}
```

一个if...else语句完全被嵌套在另一个if语句的else部分中。代码的运行方式与您预想的完全相同。如果gender的值不为m，则进入到第一个else语句。该else语句中包含另一个if语句。该if语句检查gender的值是否为f，如果不是，则进入相应的else语句。此时，您知道gender的值既不是m也不是f，因此添加了合适的代码逻辑。

嵌套使得实现某些功能更为容易，同时您也可以堆积if语句。就检查gender值的范例而言，层叠式if语句是一种更佳解决方案。

5.2.1.2 If语句的层叠

if语句的层叠指的是将else与另一个if组合在一起。理解层叠的最佳方法是看看检查gender值

的范例的层叠式解决方案，如程序清单 5.2 所示。

程序清单 5.2 stacking.cs: if 语句的层叠

```

1: // stacked.cs - Using the if...else statement
2: //-----
3:
4: class stacked
5: {
6:     static void Main()
7:     {
8:         char gender = 'x';
9:
10:        if( gender == 'm' )
11:        {
12:            System.Console.WriteLine("The gender is male");
13:        }
14:        else if ( gender == 'f' )
15:        {
16:            System.Console.WriteLine("The gender is female");
17:        }
18:        else
19:        {
20:            System.Console.WriteLine("The gender value, {0}, is not valid", gender);
21:        }
22:        System.Console.WriteLine("The if statement is now over!");
23:    }
24: }
```

该程序清单的输出如下：

```

The gender value, x, is not valid
The if statement is now over!
```

上述代码与嵌套 if 语句范例极其类似。主要区别在第 14 行，else 语句后面紧跟着 if，而没有花括号（语句块）。层叠式 if 语句的格式如下：

```

if ( condition 1 )
{
    // do something about condition 1
}
else if ( condition 2 )
{
    // do something about condition 2
}
else if ( condition 3 )
{
    // do something about condition 3
}
else if ( condition x )
{
    // do something about condition x
}
```

```

else
{
    // All previous conditions failed
}

```

上述格式较容易理解。就检查 gender 值的范例而言，只有两个条件需要检查。有时候，需要检查的条件多于两个。例如，您可以创建一个计算机程序来检查骰子的点数，并根据点数执行不同的操作。每个被层叠的 if 语句检查不同的点数（1~6），而最后的 else 语句打印错误消息，因为骰子的点数只能是 1~6。相应的代码如下：

```

if (roll == 1)
    // roll is 1
else if (roll == 2)
    // roll is 2
else if (roll == 3)
    // roll is 3
else if (roll == 4)
    // roll is 4
else if (roll == 5)
    // roll is 5
else if (roll == 6)
    // roll is 6
else
    // it isn't a number from 1 to 6

```

上述代码理解起来相对容易，因此很容易知道，对骰子 6 种可能的点数都进行了检查。如果点数不属于这 6 种情况之一，则最后的 else 语句负责处理错误逻辑或“重掷”逻辑。

注意：从上述代码可知，没有使用花括号将 if 语句括起。如果 if 或 else 内只有一条语句，则无需使用花括号将其括起。仅当有多条语句时，才需要这样做。

5.2.2 switch 语句

在根据变量的值改变程序流程方面，C# 提供了一种更容易的方式：switch 语句。switch 语句的格式如下：

```

switch ( value )
{
    case result_1 :
        // do stuff for result_1
        break;
    case result_2 :
        // do stuff for result_2
        break;
    ...
    case result_n :
        // do stuff for result_x
        break;
    default:
        // do stuff for default case
        break;
}

```

```
}
```

从中可以知道，其中不包含 condition，而是使用了 value。value 可以是表达式，也可以是变量。value 将与每一个 case 语句中的值进行比较，直到找到匹配的值为止。如果没有找到匹配的值，则进入到 default。如果没有 default，则进入 switch 语句后的第一条语句。

找到匹配的值后，相应的 case 语句将被执行。执行完该 case 语句后，switch 语句也就执行完毕。因此，最多只会执行其中的一个 case 语句，然后将执行 switch 语句后的第一条语句。程序清单 5.3 演示了 switch 语句的用法，这里以掷骰子为例。

程序清单 5.3 roll.cs: 使用 switch 语句处理掷骰子的情况

```
1: // roll.cs- Using the switch statement.
2: //-----
3:
4: class roll
5: {
6:     public static void Main()
7:     {
8:         int roll = 0;
9:
10:        //The next two lines set the roll to a random number from 1 to 6
11:        System.Random rnd = new System.Random();
12:        roll = (int) rnd.Next(1,7);
13:
14:        System.Console.WriteLine("Starting the switch... ");
15:
16:        switch (roll)
17:        {
18:            case 1:
19:                System.Console.WriteLine("Roll is 1");
20:                break;
21:            case 2:
22:                System.Console.WriteLine("Roll is 2");
23:                break;
24:            case 3:
25:                System.Console.WriteLine("Roll is 3");
26:                break;
27:            case 4:
28:                System.Console.WriteLine("Roll is 4");
29:                break;
30:            case 5:
31:                System.Console.WriteLine("Roll is 5");
32:                break;
33:            case 6:
34:                System.Console.WriteLine("Roll is 6");
35:                break;
36:            default:
37:                System.Console.WriteLine("Roll is not 1 through 6");
38:                break;
```

```
39:     }
40:     System.Console.WriteLine("The switch statement is now over!");
41: }
42: }
```

该程序清单的输出如下所示：

```
Starting the switch...
Roll is 1
The switch statement is now over!
```

注意：您在执行上述程序时，输出的点数可能不为1。

分析：该程序清单比以前的都稍长些，但功能也强大些。首先需要注意的是第16~39行，这些代码行包含switch语句，需要重点讨论。switch语句使用存储在roll变量中的值，根据roll的值，其中的某条case语句将被执行。如果点数不为1~6，则从第39行开始的default语句将被执行。如果为1~6，则相应的case语句将被执行。

您应该注意到，每条case语句都以break命令结束，这是必须的。它指明了case语句的结束。如果不包含break命令，将出现编译错误。

为使该程序清单更有趣些，加入了第11和12行的代码，第11行您可能不熟悉，它创建了一个名叫rnd变量，该变量是一个存储随机数的对象。明天的课程将再次探讨这行代码，并详细介绍其功能。就现在而言，您只需知道它创建了一个用于存储随机数的变量即可。

第12行的代码也将在接下来的几天介绍，其中的命令(int) rnd.Next(1,7)生成一个1~6之间的随机数。

提示：可以使用第11和12行的代码来生成任何范围内的随机数，只需将其中的1和7修改为要指定的范围的上限和下限即可。第一个数是将返回的最小值，第二个数比将返回的最大值大1。例如，要生成90~100之间的随机数，可以将第12行修改为如下所示：

```
Roll = (int) rnd.Next(90, 101);
```

5.2.2.1 多种情况使用同一种解决方案

有时候，可能对于多个不同的值，都执行相同的代码。例如，对于掷骰子的情况，您可能在点数为奇数时执行某种操作，而在点数为偶数时执行另一种操作。在这种情况下，可以将多个case语句作为一组，相应的switch语句如下：

```
switch (roll)
{
    case 1:
    case 3:
    case 5:
        System.Console.WriteLine("Roll is odd");
        break;
    case 2:
    case 4:
    case 6:
        System.Console.WriteLine("Roll is even");
        break;
    default:
```

```

        System.Console.WriteLine("Roll is not 1 through 6");
        break;
    }
}

```

点数为 1、3 或 5 时，执行的代码相同。另外，点数为 2、4 或 6 时，执行的代码也相同。

警告：在 C++ 中，可以通过省略 break 命令，来使多个 case 语句执行相同的代码。省略 break 命令，可以跳过两个 case 语句之间的代码。但在 C# 中，这样做是非法的。Case 语句之间的代码不能被跳过，也就是说，如果要 case 语句分组，则 case 语句之间不能有任何代码，而只能将代码放在每组中最后一个 case 语句的后面。

5.2.2.2 执行多个 case 语句

有时候，需要执行同一个 switch 语句中的多个 case 语句。在 C# 中，可以使用 goto 命令来实现这种功能。可以在 switch 语句中使用 goto 命令来跳到某个 case 语句或 default 命令。下面的代码片段对上一节的 switch 语句做了修改，它使用了 goto 命令，而不是简单地跳过后面的 case 语句。

```

switch (roll)
{
    case 1:
        goto case 5;
        break;
    case 2:
        goto case 6;
        break;
    case 3:
        goto case 5;
        break;
    case 4:
        goto case 6;
        break;
    case 5:
        System.Console.WriteLine("Roll is odd");
        break;
    case 6:
        System.Console.WriteLine("Roll is even");
        break;
    default:
        System.Console.WriteLine("Roll is not 1 through 6");
        break;
}

```

虽然这个例子使用 goto 命令来演示，但采用前一节将多个 case 语句分组的方法将会简单多。您将发现，有时候 goto 命令确实能够提供您所需要的解决方案。

5.2.3 switch 语句的控制类型

只有某些数据类型能用于 switch 语句中，switch 语句的数据类型——或控制类型 (governing type)——指的是 switch 语句的表达式将被解析成的类型。数据类型 sbyte、byte、short、ushort、int、uint、long、ulong、char 或文本字符串是控制类型；另一种数据类型 enum 也是合法的控制类型。数据类型 enum 将在第 8 天介绍。

如果表达式的数据类型不是上述类型,则该数据类型必有唯一的一种隐式转换方式可以将其转换为 sbyte、byte、short、ushort、int、uint、long、ulong 或字符串。如果没有这样的隐式转换方式,或者这样的方式不止一种,则编译程序时将发生错误。

注意: 如果忘记了隐式转换是什么,可参阅第3天的课程。

应该	不应该
检查变量的多种不同的值时,应该使用 switch 语句	不要在 switch 或 if 语句的条件后面加上分号: if (condition);

5.3 使用循环语句

除了使用选择语句改变程序流程外,有时候可能需要重复执行某个代码片段多次。为了能够重复执行代码,C#提供了多种循环语句。循环语句可以执行代码块 0 或更多次。每执行一次代码,便是一次循环。

C#中的循环语句包括:

- while;
- do;
- for;
- foreach。

5.3.1 while 语句

while 命令用于条件为真时,不断重复执行代码块,其格式如下:

```
while (condition)
    statement(s)
```

图 5.1 对此做了说明。

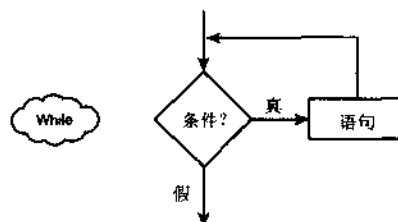


图 5.1 while 命令

从上图可以知道, while 语句使用了一个条件语句,如果条件满足,则语句将被执行,否则不执行,程序流程进入 while 语句后面的语句。程序清单 5.4 使用 while 语句打印 10 个 1~10 之间的随机数及其平均值。

程序清单 5.4 average.cs: 使用 while 命令

```
1: // average.cs Using the while statement.
2: // print the average of 10 random numbers that are from 1 to 10.
3: //-----
```

```
4:
5: class average
6: {
7:     public static void Main()
8:     {
9:         int ttl = 0; // variable to store the running total
10:        int nbr = 0; // variable for individual numbers
11:        int ctr = 0; // counter
12:
13:        System.Random rnd = new System.Random(); // random number
14:
15:        while ( ctr < 10 )
16:        {
17:            //Get random number
18:            nbr = (int) rnd.Next(1,11);
19:
20:            System.Console.WriteLine("Number {0} is {1}", (ctr + 1), nbr);
21:
22:            ttl += nbr;        //add nbr to total
23:            ctr++;            //increment counter
24:        }
25:
26:        System.Console.WriteLine("\nThe total of the {0} numbers is {1}", ctr, ttl);
27:        System.Console.WriteLine("\nThe average of the numbers is {0}", ttl/ctr );
28:    }
29: }
```

注意：您执行该程序时，得到的输出将与这里的不同。因为这里使用的是随机数，这些数字在每次执行程序时都不同。

该程序清单的输出如下：

```
Number 1 is 2
Number 2 is 5
Number 3 is 4
Number 4 is 1
Number 4 is 1
Number 6 is 5
Number 7 is 2
Number 8 is 5
Number 9 is 10
Number 10 is 2
```

```
The total of the 10 numbers is 37
```

```
The average of the numbers is 3
```

分析：这里使用了今天介绍的生成随机数的代码。只不过生成的随机数不在1~6之间，而是1~10之间。这是在第18行完成的，这里将随机数的范围扩大到10。在以这种方式使用随机变量之前，第13行对其进行了初始化。

while 语句是从第 15 行开始的。该 while 语句的条件是检查计数器是否小于 10。由于该计数器在第 11 行被初始化为 0，因此条件是满足的，所以 while 内的语句将被执行。while 语句取得一个 1 ~ 10 之间的随机数（第 18 行），并将其加入到累计计数器 nl 中（第 22 行）。第 23 行将计数器变量 ctr 加 1，然后便是 while 语句的结尾处（24 行）。程序流程将自动回到第 15 行的 while 条件处，对条件重新进行评估，看其是否仍然满足。如果满足，则再次执行语句。这一过程将一直持续下去，直到 while 条件不满足为止。就这个程序而言，当 ctr 的值为 10 时，条件便不满足了。此时程序流程将进入第 25 行，即紧跟在 while 语句后的一行。

While 语句后面的代码打印 10 个随机数的总和及平均值。然后结束程序。

警告：为确保 while 语句最终得以结束，必须在语句中修改某些将影响条件的因素。如果条件永远不会不满足，则 while 语句将成为死循环。还有另一种终止 while 循环的方法：使用 break 语句。这将在下一节进行介绍。

5.3.1.1 中断和继续 while 语句

可以在条件不满足之前中断 while 语句，也可以在执行完毕所有的语句之前中断 while 语句的一次循环。

要中断 while 语句，提早结束它，可以使用 break 命令。Break 语句将使程序流程立刻转到 while 语句后面的第一条语句。

也可以使 while 语句立刻进入到下一次循环，这是通过 continue 语句实现的。Continue 语句导致程序流程进入到 while 的条件语句。程序清单 5.5 演示如何在 while 语句中使用 break 和 continue 语句。

程序清单 5.5 even.cs: 使用 break 和 continue 语句

```
1: // even.cs- Using the while with the break and continue commands.
2: //-----
3:
4: class even
5: {
6:     public static void Main()
7:     {
8:         int ctr = 0;
9:
10:        while (true)
11:        {
12:            ctr++;
13:
14:            if (ctr > 10 )
15:            {
16:                break;
17:            }
18:            else if ( (ctr % 2) == 1 )
19:            {
20:                continue;
21:            }
22:            else
23:            {
24:                System.Console.WriteLine("...{0}...", ctr);
25:            }
```

```

26:     }
27:     System.Console.WriteLine("Done!");
28: }
29: }

```

该程序清单的输出如下所示:

```

...2...
...4...
...6...
...8...
...10...
Done!

```

分析: 这个程序清单打印偶数, 而忽略奇数。当计数器的值大于10后, 通过break语句结束while语句。

第8行声明了一个计数器变量ctr, 并将其值设置为0。从第10行开始为while语句, 由于使用break语句来结束循环, 因此第10行的条件被简单地设置为真。这实际上创建了一个死循环, 因此需要使用break语句来结束while语句的循环。在while语句中, 首先将ctr的值加1, 如第12行所示。然后检查ctr的值是否大于10, 如第14行所示。如果ctr大于10, 则执行第16行的break语句, 终止while循环, 进入第27行。

如果ctr小于10, 则执行第18行的else语句。该else语句与一个if语句层叠起来, 后者检查当前的数字是否为奇数, 这是使用求模运算符实现的。如果计数器为偶数, 则其除以2的余数将为0, 如果为奇数, 则余数将为1。当计数器为奇数时, 将执行第20行的continue语句, 即回到while语句的开始位置, 重新检查条件。由于条件总是为真, 因此将再次执行while语句。这将首先将计数器加1, 如第12行所示, 然后执行其他检查。

如果计数器不是奇数, 则第22行的else语句将被执行。该else语句只包含一条语句, 它调用WriteLine, 打印计数器的值。

5.3.2 do 语句

如果首次检查时, while语句的条件便不满足, 则while语句一次也不会执行。有时候, 希望语句至少执行一次。在这种情况下, do语句将是一种更好的解决方案。

do语句的格式如下:

```

do
    statement(s)
while (condition);

```

图5.2说明了这种格式。

从上图可以知道, do语句首先执行其包含的语句, 然后对while语句中的条件进行检查。这里的while语句及其条件的运行方式与前面介绍的while语句完全相同。如果条件为真, 则返回到前面的语句; 否则, 进入do...while语句后的第一行。程序清单5.6演示了do语句的用法。

注意: 由于do语句中要使用while语句, 因此do语句常被称为do...while语句。

程序清单 5.6 do_it.cs: 使用do命令

```

1: // do_it.cs Using the do statement.

```

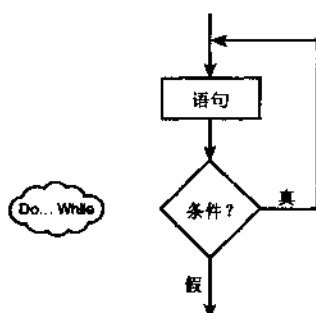


图 5.2 do 命令

```

2: // Get random numbers (from 1 to 10) until a 5 is reached.
3: //-----
4:
5: class do_it
6: {
7:     public static void Main()
8:     {
9:         int ttl = 0; // variable to store the running total
10:        int nbr = 0; // variable for individual numbers
11:        int ctr = 0; // counter
12:
13:        System.Random rnd = new System.Random(); // random number
14:
15:        do
16:        {
17:            //Get random number
18:            nbr = (int) rnd.Next(1,11);
19:
20:            ctr++;           //number of numbers counted
21:            ttl += nbr;      //add nbr to total of numbers
22:
23:            System.Console.WriteLine("Number {0} is {1}", ctr, nbr);
24:
25:        } while ( nbr != 5.);
26:
27:        System.Console.WriteLine("\n{0} numbers were read", ctr);
28:        System.Console.WriteLine("The total of the numbers is {0}", ttl);
29:        System.Console.WriteLine("The average of the numbers is {0}", ttl/ctr);
30:    }
31: }

```

该程序清单的输出如下所示:

```

Number 1 is 1
Number 2 is 6
Number 3 is 5

```

```

3 numbers were read

```

```
The total of the numbers is 12
The average of the numbers is 4
```

由于该程序清单使用的是随机数，因此您执行该程序得到的结果很可能不同于这里显示的。您将得到一系列的数字，但最后一个为 5。

在该程序中，您希望至少执行操作（获取一个随机数）一次。然后一直执行这种操作，直到条件（获取的随机数为 5）满足为止。这种情况非常适合使用 `do` 语句。该程序清单与程序清单 5.4 极其类似。第 9~11 行声明了几个变量来记录总数和计数器。第 13 行声明了一个用于获取随机数的变量。

`do` 语句是从第 15 行开始的。`do` 语句体（第 16~24 行）被执行。首先获取下一个随机数，同样这里将 1~10 之间的随机数赋给变量 `nbr`。第 20 行在每读取一个随机数后将 `ctr` 变量的值加 1，来记录读取了多少个随机数。第 21 行将取得的随机数加入到累积值中。记住下面的代码：

```
ttl += nbr;
```

与下述代码是等效的：

```
ttl = ttl + nbr;
```

第 23 行将取得的随机数及其编号打印到屏幕上。

第 25 行是 `do` 语句的条件部分。这里的条件是 `nbr` 不等于 5。只要取得的随机数 `nbr` 不等于 5，`do` 语句体便继续执行。当取得 5 时，循环将终止。从程序的输出可以知道，其中总是只有一个“5”，并且是最后一个数。

第 27~29 行打印关于取得的随机数的统计信息。

5.3.3 for 语句

虽然 `do...while` 和 `while` 语句提供了控制代码循环所需的所有功能，但还有其他的语句也可用于控制代码的循环。介绍 `for` 语句之前，请先看一下下面的代码片段：

```
ctr = 1
while (ctr < 10)
{
    //do some stuff
    ctr ++;
}
```

从中可以知道，使用了一个计数器来循环执行 `while` 语句。其流程如下：

1. 将计数器的值设置为 1。
2. 检查计数器是否小于 10。如果不小于 10（条件不满足），则结束循环。
3. 执行某些操作。
4. 将计数器加 1。
5. 回到第 2 步。

在循环中，这些步骤很通用。由于这些步骤是通用的，因此 `for` 语句也使用这些步骤，不过格式更为简单：

```
for (initializer; condition; incrementor)
    statement(s)
```

从上述 for 语句的格式可以知道，括号内包含三部分：initializer、condition 和 incrementor，各部分之间用分号隔开。省略其中的某部分时，分号仍需保留。

for 语句执行时，首先执行 initializer，它只在开始时执行一次，以后再也不执行。

执行 initializer 后，对 condition 进行评估。和 while 语句中的条件一样，这里的条件也只能为真或假。如果为真，则 statement(s) 将被执行。

语句或语句块执行后，将返回到 for 语句的开始位置，执行 incrementor。Incrementor 可以是任何合法的 C# 表达式，但通常用于增加计数器的值。

Incrementor 被执行后，将再次对条件进行评估。只要条件仍为真，语句块便会被执行，然后执行 incrementor。这一过程将一直重复下去，直到条件为假为止。图 5.3 说明了 for 语句的流程。

介绍程序清单之前，我们来将本节开始的 while 语句修改为 for 语句：

```
for ( ctr = 1; ctr < 10; ctr++)
{
    //do some stuff
}
```

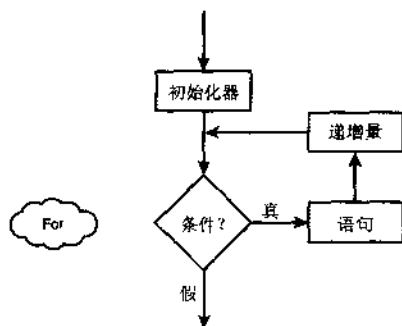


图 5.3 for 命令

上述 for 语句比前面使用 while 的代码简单得多。

该 for 语句的执行步骤如下：

1. 将计数器的值设置为 1。
2. 检查计数器是否小于 10。如果不小于 10（条件不满足），则结束 for 语句。
3. 执行某些操作。
4. 将计数器加 1。
5. 回到第 2 步。

其步骤与前面的 while 语句完全相同，区别在于 for 更简洁，更易于理解。程序清单 5.7 演示了一个更强大的 for 语句。实际上，该程序清单的功能与程序清单 5.4 完全相同，只不过更为简洁而已。

程序清单 5.7 foravg.cs: 使用 for 语句

```
1: // foravg.cs Using the for statement.
2: // print the average of 10 random numbers that are from 1 to 10.
3: //-----
4:
5: class average
```

```
6: {
7:     public static void Main()
8:     {
9:         int ttl = 0; // variable to store the running total
10:        int nbr = 0; // variable for individual numbers
11:        int ctr = 0; // counter
12:
13:        System.Random rnd = new System.Random(); // random number
14:
15:        for ( ctr = 1; ctr <= 10; ctr++ )
16:        {
17:            //Get random number
18:            nbr = (int) rnd.Next(1,11);
19:
20:            System.Console.WriteLine("Number {0} is {1}", (ctr), nbr);
21:
22:            ttl += nbr;        //add nbr to total
23:        }
24:
25:        System.Console.WriteLine("\nThe total of the 10 numbers is {0}", ttl);
26:        System.Console.WriteLine("\nThe average of the numbers is {0}", ttl/10 );
27:    }
28: }
```

该程序清单的输出如下:

```
Number 1 is 10
Number 2 is 3
Number 3 is 6
Number 4 is 5
Number 4 is 7
Number 6 is 8
Number 7 is 7
Number 8 is 1
Number 9 is 4
Number 10 is 3
```

```
The total of the 10 numbers is 54
```

```
The average of the numbers is 5
```

分析: 该程序清单的大部分内容都与程序清单5.4相同。但也有一些区别。第15行使用了for语句。计数器被初始化为1, 这使在第20行使用WriteLine例程来显示值更为容易。另外, for语句中的条件语句也做了相应的调整。

程序执行到for语句处时, 将出现什么情况呢? 简单地说, 计数器被设置为1, 然后验证条件。这时候, 计数器小于等于10, 因此for语句体被执行。for语句体(第16~23行)执行完毕后, 回到第15行中for语句的incrementor处。这里将计数器的值加1。然后再次对条件进行检查, 如果为真, 则再次执行for语句体。这一过程将一直重复下去, 直到条件为假为止。在这个程序中, 当计数器

的值为 11 时，条件便为假。

5.3.3.1 for 语句表达式

可以在 `initializer`、`condition` 和 `incrementor` 中完成很多工作。可以将表达式放在这些地方，甚至可以放置多个表达式。

如果在 `for` 语句的某部分中使用多个表达式，则需要分开它们。分隔符是逗号。作为一个例子，下面的 `for` 语句初始化两个变量，并递增它们的值。

```
for ( x = 1, y = 1; x + y < 100; x++, y++)  
    //Do something...
```

除了可以包含多个表达式外，还可以省略某些部分。在下面的范例中，所有的工作都是在 `for` 语句的控制结构中完成的，`for` 语句体只有一条空语句——一个分号。

```
for (x = 0; ++x <= 10; System.Console.WriteLine("{0}", x))
```

上述简单的代码行实际上完成了很多任务，它打印数字 1 ~ 10。本章后面的一个练习要求您将
其改写为一个完整的程序清单。

警告：不要在 `for` 语句的控制结构中完成太多的工作，要确保它不要过于复杂，以至难以理解。

5.3.4 foreach 语句

`foreach` 语句以类似于 `for` 语句的方式执行循环，但有一种特殊用途，即可用于遍历诸如数组等集合。`foreach` 语句、集合和数组将在第 8 天的课程中介绍。

5.3.5 再谈 break 和 continue 语句

在前面关于 `while` 语句的一节中已经介绍过 `break` 和 `continue`，另外还在 `switch` 语句中使用过 `break` 命令。这两个命令也可以用于其他的程序流程语句中。

在 `do...while` 语句中，`break` 和 `continue` 的运行方式与在 `while` 语句中完全相同。`continue` 命令返回到条件语句处；而 `break` 命令转到 `do...while` 后面的语句。

在 `for` 语句中，`continue` 语句转到 `incrementor` 语句处，然后对条件进行检查，如果为真，则继续执行 `for` 语句。`break` 语句转到 `for` 语句之后的语句。

`break` 命令退出当前循环；而 `continue` 命令开始下一次循环。

5.4 使用 goto

不管您使用的是哪种编程语言，`goto` 语句都充满争议。由于 `goto` 语句能够无条件地改变程序流程，因此非常强大。但权力和义务总是并存的。许多开发人员避免使用 `goto` 语句，因为它容易导致代码难以理解。

`goto` 语句的使用方式有三种。在前面您已经看到了，在 `switch` 语句中，可以以两种方式使用 `goto` 语句：`goto case` 和 `goto default`。在前面讨论 `switch` 语句时，以这两种方式使用过 `goto` 语句。

第三种 `goto` 语句的使用格式如下：

```
goto label;
```

使用这种格式的 `goto` 语句，可以转到一条标签语句。

5.4.1 标签语句

标签语句只不过是一个指示位置的命令。标签的格式如下：

label_name:

注意，它后面跟的是冒号，而不是分号。程序清单 5.8 演示了如何使用带标签的 goto 语句。

程序清单 5.8 Score.cs: 使用带标签的 goto 语句

```
1: // score.cs Using the goto and label statements.
2: // Disclaimer: This program shows the use of goto and label
3: //           This is not a good use; however, it illustrates
4: //           the functionality of these keywords.
5: //-----
6:
7: class score
8: {
9:     public static void Main()
10:    {
11:        int score = 0;
12:        int ctr = 0;
13:
14:        System.Random rnd = new System.Random();
15:
16:        Start:
17:
18:        ctr++;
19:
20:        if (ctr > 10)
21:            goto EndThis;
22:        else
23:            score = (int) rnd.Next(60, 101);
24:
25:        System.Console.WriteLine("{0} - You received a score of {1}",
26:                                ctr, score);
27:
28:        goto Start;
29:
30:    EndThis:
31:
32:        System.Console.WriteLine("Done with scores!");
33:    }
34: }
```

该程序清单的输出如下所示：

```
1 - You received a score of 83
2 - You received a score of 99
3 - You received a score of 72
4 - You received a score of 67
5 - You received a score of 80
6 - You received a score of 98
```

```
7 - You received a score of 64
8 - You received a score of 91
9 - You received a score of 79
10 - You received a score of 76
Done with scores!
```

分析：该程序清单的目的相对简单，它通过取得10个60~100之间的随机数来打印10个分数。这里使用随机数的方式与前面介绍的相同，只是做了细微的修改。在第23行，指定的随机数下限为60，而不是1；另外，由于要获取60~100的随机数，因此上限被设置为101。通过将101设置为上限，取得的随机数将小于101。

该程序清单的重点在第16、21、28和30行。第16行包含一个名为Start的标签，由于它是一个标签，因此程序执行时将略过这一行，进入第18行，将计数器的值加1。然后，检查第20行的if语句中的条件。如果计数器的值大于10，则第21行的goto语句被执行，即跳到第30行的标签End。此处。由于此时计数器不大于10，因此第22行的else语句被执行。该else语句取得一个随机的分数（如第23行所示，这在前面已经介绍过了）。第25行打印取得的分数，然后进入到第28行，该行无条件地跳到Start标签处。由于Start标签位于第16行，因此将跳到第16行。

该程序清单完成的循环与while、do或for语句类似。在很多情况下，可以使用其他编程元素来代替goto语句。如果有其他的方法可供选择，则应该使用这些方法。

提示：应尽可能避免使用goto语句，它会导致被称为意大利面条式代码（spaghetti code）的情形。意大利面条式代码指的是所有代码纠缠在一起，因此难以知道代码从哪里开始，到哪里结束。

5.5 程序流程命令的嵌套

今天介绍的所有程序流程命令都可以被嵌套。将程序流程命令嵌套时，应确保命令正确结束。如果嵌套不正确，可能导致逻辑错误，甚至语法错误。

应该	不应该
一定要对代码进行注释，以清楚地说明程序的功能和流程	除非绝对必要，否则不要使用goto语句

5.6 总 结

今天介绍的内容很多，而这些知识几乎在所有的C#应用程序中都要用到。

今天的课程再次对C#语句的一些基本结构进行了介绍。首先通过介绍else语句，扩展了您关于if语句的知识。今天还介绍了另一种选择语句：switch语句。接着讨论了控制程序循环流程的语句，包括while、do和for语句。另外，还简要介绍了另一个命令foreach，该命令将在第8天详细介绍。您除了学会了如何使用命令外，还知道这些命令可以嵌套。最后，介绍了goto语句以及如何将其与case、default和标签结合起来使用。

5.7 问与答

问：还有其他类型的控制语句吗？

答：有 throw、try、catch 和 finally。后面的课程将介绍这些语句。

问：可以在 switch 语句中使用文本字符串吗？

答：可以。字符串是 switch 语句的一种“控制类型”，这意味着可以在 switch 语句中使用字符串变量，进而在 case 语句中使用字符串值。记住，字符串只不过是使用双引号标示的文本而已。在后面的一个练习中，您将创建一条使用字符串的 switch 语句。

问：为何人们对 goto 语句的评价如此差？

答：goto 语句臭名昭著。如果以结构化、有组织的方式偶尔使用，goto 语句可以帮助解决一些编程方面的问题，goto case 和 goto default 就是绝佳的例子。goto 语句之所以臭名昭著，主要在于它经常被滥用。程序员以非结构化的方式使用它，快速地从代码的一个地方跳到另一个地方。在面向对象的编程语言中，程序的结构化程度越高，程序越好，维护起来越容易。

5.8 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

5.8.1 小测验

1. 为重复执行代码行多次，C#提供了哪些命令？
2. while 语句中的语句将至少执行多少次？
3. do 语句中的语句将至少执行多少次？
4. 下面的 for 语句中，哪一部分是条件语句？

```
for ( x = 1; x == 1; x++)
```
5. 上述 for 语句的 incrementor 语句是什么？
6. 什么语句用于结束 switch 语句中的 case 表达式？
7. 标签中使用的是什么标点？
8. 在 for 语句中，使用哪个标点符号来分隔表达式？
9. 嵌套指的是什么？
10. 什么命令用于跳到下一次循环。

5.8.2 练习

1. 编写一条 if 语句，检查变量 file-type 的值是 s、m 还是 j，并根据 file-type 的值打印下面的消息：

```
s: The filer is single
m: The filer is married filing at the single rate
j: The filer is married filing at the joint rate
```

2. 下面的 if 语句合法吗？这些代码执行后，x 的值将为多少？

```
int x = 2;
int y = 3;
if( x == 2) if (y > 3) x=5; else x =9;
```

3. 编写一个从 99 数到 1 的 while 语句。

4. 将练习 3 的 while 语句改为 for 语句。

5. 下面的程序清单正确吗？如果正确，其功能是什么？如果不正确，错在哪里？

```
// Ex5-5.cs, Exercise 5 for Day 5
//-----

class score
{
    public static void Main()
    {
        int score = 99;

        if ( score == 100 );
        {
            System.Console.WriteLine("You got a perfect score!");
        }
        else
            System.Console.WriteLine("Bummer, you were not perfect!");
    }
}
```

6. 编写一个这样的 for 循环，即在其 initializer、condition 和 incrementor 部分完成打印数字 1 ~ 10 的工作，for 语句体是一条空语句。

7. 编写一条根据变量 name 的值执行不同操作的 switch 语句。如果该变量的值为 Robert，则打印消息“Hi Bob”；如果为 Richard，则打印消息“Hi Rich”；如果为 Barbara，则打印消息“Hi Barb”；如果为 Kalee，则打印“You Go Girl!”。对于其他情况，则打印“Hi x”，其中 x 为变量 name 的值。

8. 编写一个掷骰子 100 次的程序，并打印出各种点数的出现次数。

第6天课程

类

正如第2天介绍的,对于面向对象语言而言,类至关重要,对于C#来说也是如此。在本书前面介绍的每一个程序中都使用了类。今天将介绍以下内容:

- 面向对象编程涉及的概念;
- 如何声明类;
- 如何定义类;
- 类成员;
- 创建自己的数据成员;
- 在类中实现属性;
- 名称空间。

6.1 再谈面向对象编程

第2天介绍过,C#是一种面向对象语言。要充分利用C#,必须理解面向对象语言的概念。接下来的几节将简要地重温一下第2天介绍的一些概念,然后介绍如何将这些概念用于实际的C#程序中。

第2天介绍过,面向对象语言的关键特征包括:

- 封装;
- 多态;
- 继承;
- 重用。

6.1.1 封装

封装指的是建立类(包),其中包含所需的所有东西。在面向对象编程中,这意味着您可以创建一个类,其中存储了您需要的所有变量以及操纵这些数据的例程。可以创建一个 circle 类,来存储关于圆的信息。这可能包括圆心的位置、半径以及对所有的圆都通用的例程。这些例程可能包括计算圆面积和周长、改变圆心位置和半径。

通过封装圆,可以让用户无需关心圆的工作原理,而只要知道如何与其交互即可。它将圆的内部工作方式隔离开来,即类中的变量可以被修改,但对用户而言,是不可见的。例如,可以存储

圆的直径，而不是半径。如果封装了功能和数据，则上述修改只会影响类。对于使用该类的任何程序，都无需进行修改。今天和明天的课程中都包括直接使用 Circle 类的程序。

注意：封装常被称为“黑盒化 (black boxing)” 黑盒化指的是隐藏功能或内部处理方式。对于圆而言，传入半径可以获得其面积。您无需关心这是如何得到的，只要知道得到的答案是正确的。

6.1.2 多态

多态指的是能够呈现多种形式的功能，即程序能够使用您发送给它的东西进行工作。例如，在前几天使用过的 WriteLine() 中，您可以使用 {} 来创建参数字段。该字段打印什么值呢？它可以打印任何类型的变量或字符串。至于如何打印它们，则是由 WriteLine() 例程负责的。该例程能够接受您传递给它的大部分数据类型，因此是多态的。

以圆为例，您可能想调用圆对象来取得其面积。为此，可以传递圆周上的三个点或圆心和半径。无论采用哪种方式，得到的结果都是相同的。另外，圆是一种形状，因此圆的一种多态特征是能够理解这种形状，并以这种形状的方式做出反应。

6.1.3 继承

正如第2天介绍过的，继承是最复杂的面向对象概念。继承在一个类（对象）是另一个类的扩展时发生。

在许多关于面向对象编程的书籍中，使用动物方面的类比来说明继承。这种类比首先使用了动物是一种生物的概念。

爬行动物属于动物，但是变温的。爬行动物具备动物的所有特征，但同时有自己特有的特征。而蛇属于爬行动物，但修长且没有脚。它具备爬行动物的所有特征，但同时有自己特有的特征。可以这么说，蛇继承了爬行动物的特征，而爬行动物继承了动物的特征。

在第11天的课程中，您将知道如何将这种概念用于类和编程中。

6.1.4 重用

创建类后，可以重复使用它来创建大量的对象。通过使用继承以及前面介绍的其他一些特性，可以创建出能够在许多程序中以许多不同的方式重复使用的例程。通过封装功能，可以创建出经过测试并被证明能够正确运行的例程。用户无需详细测试功能，而只要正确使用它即可。这使得重用这些例程简单而快速。

6.1.5 对象和类

第2天使用蛋糕刀和蛋糕来说明了类和对象。现在有了这些概念后，可以使用它们来编写一些代码了。

注意：接下来的三天中，将介绍关于类的知识，首先介绍一些非常简单的范例，然后对其进行扩展。

6.2 定义类

为简化起见，关键字 `class` 被用来定义类。类的基本结构如下。

```
class identifier
{
    class-body;
```

```
}
```

其中 `identifier` 是给类指定的名称, `class-body` 是组成类的代码。

类名和其他变量名一样, 可以指定。应该给类指定一个有意义的名称, 即对类的功能进行描述。

Microsoft .NET 框架中包含大量内置类, 本书一开始便使用了其中的一个: `Console` 类。`Console` 类包含多个数据成员和例程, 前面已经使用过其中的几个例程, 包括 `Write` 和 `WriteLine`。这个类的名称 (标识符) 为 `Console`。`Console` 的类体中包含了例程 `Write` 和 `WriteLine` 的代码。阅读完明天的课程后, 您便能够创建并命名自己的、包含与 `Console` 类中类似的例程的类。

6.3 类声明

定义类后, 便可以使用它来创建对象。类只是一种用于创建对象的定义。类本身并不能存储信息或执行例程, 而只是用于声明对象。对象可以按类定义的方式来存储信息和执行例程。

注意: 声明对象常被称为实例化。换句话说, 对象是类的一个实例。

使用类来声明对象的格式如下:

```
class_name object_identifier = new class_name( );
```

其中 `class_name` 是类的名称, 而 `object_identifier` 是要声明的对象的名称。例如, 如果有一个名为 `point` 的类, 则可以使用下面的代码来创建一个名为 `startingPoint` 的对象:

```
point startingPoint = new point( );
```

类名为 `point`, 声明的对象的名称为 `startingPoint`。由于 `startingPoint` 是一个对象, 因此它包含 `point` 类声明的数据和例程。

您可能会问, 上述声明代码行中的其他项是干什么的? 这里最重要的是以前未介绍过的关键字 `new`。

顾名思义, 关键字 `new` 用于创建一个新东西, 这里创建一个新的点。由于 `point` 是一个类, 因此创建的是一个对象。关键字 `new` 指示要创建一个新的实例, 这里创建的新实例是一个 `point` 对象。

使用类来声明对象时, 必须在赋值运算符右边的类名后面加上圆括号, 这样可以使类来构造一个新对象。

警告: 不加上构造代码 `new classname`, 将声明一个类, 但编译器不会构造其内部结构。一定要将 `new classname` 赋给声明的对象名, 以确保所有的东西都将构造好。明天将更详细地介绍这种初始化结构。

再来看一下上述语句:

```
point startingPoint = new point( );
```

下面分别介绍其中各部分的功能。

```
point startingPoint
```

使用 `point` 类来声明一个名为 `startingPoint` 的对象。上述代码与声明诸如 `int` 和 `decimal` 等其他数据类型类似。

```
startingPoint =
```


将赋值运算符右边的结果赋给左边的变量。这里变量是一个对象，一个名称为 `startingPoint`、类型为 `point` 的对象。

```
new point( )
```

这部分代码构造 `point` 对象。带圆括号的类名表示要构造（创建）一个这种类的对象。`New` 指示为该新对象预留一些内存空间。记住，类只是一种定义，它不存储任何东西。对象需要存储信息，因此需要为它预留内存。关键字 `new` 用于预留内存。

和所有的语句一样，该声明以分号结束，分号指示语句结束。

6.3.1 类成员

知道类的整体结构以及如何使用类来创建对象后，我们来看一下类中可以包含哪些东西。类体中可包含的东西主要有两种：数据成员和函数成员。

数据成员包括变量和常量，这包括第3天介绍的任何数据类型的变量以及后面将介绍的高级数据类型的数据成员。数据成员甚至可以是其他类。

类体中的另一种元素是函数成员。函数成员指的是执行操作的例程。这种操作可以简单到取得一个值，也可以复杂到使用不同数目的值，将一行文本写到屏幕上，就像 `Write` 和 `WriteLine` 那样。`Write` 和 `WriteLine` 是 `Console` 类的成员函数。明天将介绍如何创建和使用自己的成员函数。接下来我们来介绍数据成员。

6.4 数据成员（字段）

新术语：变量也被称为**字段**。正如前面指出的，类中的数据成员是一个属于类成员的变量。

对于前面介绍的 `point` 类，您希望它包含用于存储点的 `x` 坐标和 `y` 坐标的数据成员。这些坐标值可以是很多种数据类型中的一种，如果是 `int` 型，则可以使用下面的方式来定义 `point` 类：

```
class point
{
    int x;
    int y;
}
```

这实际上就是一个非常简单的 `point` 类的代码。现在需要加入另一个元素，即访问限定符 `public`。不加入单词 `public`，将无法在 `point` 的外面访问 `x` 和 `y`。除非特别说明，否则只能在声明变量的代码块内访问该变量。这里的语句块是 `point` 类的定义。

注意：代码块是位于花括号内的代码片段，类体就是一个代码块。

对 `point` 类所做的修改比较简单，加入访问限定符 `public` 后，`point` 类将变成如下所示：

```
class point
{
    public int x;
    public int y;
}
```

虽然 `point` 类包含的是两个整数，但您可以在该类中使用任何数据类型。例如，您可以创建一个名为 `FullName` 的类，它包含三个字符串，分别存储姓、名和中名。也可以创建一个名为 `Address`

类，它包含 `FullName` 类和用于存储地址的其他字符串。还可以创建一个 `customer` 类，它包含一个用于存储客户编号的 `long` 变量、一个 `Address` 类、一个用于存储账户结余的 `decimal` 变量和一个指定该账户是否有效的布尔型变量等。

6.4.1 访问数据成员

声明数据成员后，需要取得它们的值。正如前面介绍的，访问限定符 `public` 让您能够从类的外面访问数据成员。

在类的外面，不能仅仅使用数据成员的名称来访问它。例如，在程序中声明了一个名为 `startingPoint` 的 `point` 对象后，好像可以使用 `x` 和 `y`（在 `point` 类中的名称）来取得该点的坐标。但如果您在同一个程序中声明了 `point` 对象 `startingPoint` 和 `endingPoint`，将会出现什么情况呢？如果您使用 `x`，访问的将是哪个点的 `x` 坐标呢？

要访问数据成员，需要同时指定对象和数据成员的名称，并在它们之间加上成员运算符（句点）。因此，要访问 `startingPoint` 的坐标，可以使用下面的代码：

```
startingPoint.x
startingPoint.y
```

对于 `endingPoint`，则可以使用下面的代码：

```
endingPoint.x
endingPoint.y
```

现在，您具备了尝试编写一个程序所需的基本知识。程序清单 6.1 声明 `point` 类，并使用该类声明了两个对象：`startingPoint` 和 `endingPoint`。

程序清单 6.1 point.cs: 声明包含数据成员的类

```
1: // point.cs- A class with two data members
2: //-----
3:
4: class point
5: {
6:     public int x;
7:     public int y;
8: }
9:
10: class pointApp
11: {
12:     public static void Main()
13:     {
14:         point starting = new point();
15:         point ending = new point();
16:
17:         starting.x = 1;
18:         starting.y = 4;
19:         ending.x = 10;
20:         ending.y = 11;
21:
22:         System.Console.WriteLine("Point 1: ({0},{1})",
```

```

23:             starting.x, starting.y);
24:         System.Console.WriteLine("Point 2: ({0},{1})",
25:             ending.x, ending.y);
26:     }
27: }

```

该程序清单的输出如下：

```

Point 1: {1,4}
Point 2: {10,11}

```

分析：第4~8行声明了一个名为point的简单类，其结构与前面介绍的相同。第4行使用了关键字class，后面跟类名point。第5行和第8行是包围类体的花括号。在类体中，声明了两个int变量x和y。它们都被声明为公有的，以便能够在类的外面使用它们。

从第10行开始，是应用程序的主要部分，有意思的是这这也是一个类！关于这一点，后面将做更详细的介绍。

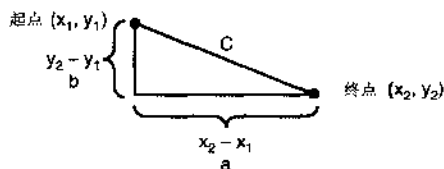
从第12行开始是一个例程，现在您应该对它很熟悉。第14和15行使用point类创建了两个对象，创建的格式与前面介绍的相同。第17~20行设置了每个point对象的数据成员的值。第17行将starting对象的数据成员x的值设置为1。使用成员运算符将成员名和对象名分隔开来。第18、19、20行的格式与此相同。

第22行包含一个WriteLine例程，该例程在本书前面已经介绍过了。但这里很特别，因为打印的是存储在对象starting中的值。这些值被存储在starting.x和starting.y中，而不是x和y中。第24行打印ending对象的值。

6.4.2 使用数据成员

程序清单6.1演示了如何给数据成员赋值以及如何取得数据成员的值。如果要执行比赋值和显示更复杂的操作时，该如何办呢？

类的数据成员和其他任何类型的变量一样，您可以将其用于运算、控制语句或其他可以使用常规变量的任何地方。程序清单6.2扩展了point类的用途，计算了两点之间的距离。如果您忘记如何计算，请参考图6.1，它说明了如何计算两点之间的距离。



$$c^2 = a^2 + b^2$$

或

$$c = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

图6.1 计算两点之间的距离

程序清单 6.2 point2.cs: 使用数据成员

```

1: // line.cs- Calculate the length of a line.
2: //-----
3:

```

```

4: class point
5: {
6:     public int x;
7:     public int y;
8: }
9:
10: class lineApp
11: {
12:     public static void Main()
13:     {
14:         point starting = new point();
15:         point ending = new point();
16:         double line;
17:
18:         starting.x = 1;
19:         starting.y = 4;
20:         ending.x = 10;
21:         ending.y = 11;
22:
23:         line = System.Math.Sqrt( (ending.x - starting.x)*(ending.x -
starting.x) +
24:                                 (ending.y - starting.y)*(ending.y - starting.y) );
25:
26:         System.Console.WriteLine("Point 1: ({0},{1})",
27:                                 starting.x, starting.y);
28:         System.Console.WriteLine("Point 2: ({0},{1})",
29:                                 ending.x, ending.y);
30:         System.Console.WriteLine("Length of line from Point 1 to Point 2: {0}",
31:                                 line);
32:     }
33: }

```

该程序清单的输出如下所示:

```

Point 1: (1,4)
Point 2: (10,11)
Length of line from Point1 to Point 2: 11.4017542509914

```

分析: 该程序清单与程序清单6.1极其类似。最大的不同在于增加了一个变量, 并做了一些计算来确定线段的长度。第16行声明了一个新的名为line的double变量, 用于存储两点之间的直线距离。

第23和24行实际上是一条语句, 该语句看起来很复杂, 其实并非如此。除了 `System.Math.Sqrt` 外, 其他的内容您应该能够理解。Sqrt 是 `System.Math` 对象的一个例程, 它计算平方根。将该公式与图 6.1 进行比较, 您将发现这是正确的。最后得到的结果是线段的长度。需要注意的重要一点是, 在计算中, 数据成员的使用方式与其他变量相同, 唯一的差别在于命名模式。

6.4.3 将类用作数据成员

前面介绍过, 可以在一个类中嵌套另一个类。类也是一种数据类型, 因此使用类声明的对象只不过是一种更复杂的变量而已, 可以在能够使用其他变量的任何地方使用它。程序清单 6.3 包含一

个 line 类，这个类由两个 point 对象（starting 和 ending）组成。

程序清单 6.3 line2.cs: 嵌套类

```

1: // line2.cs- A class with two data members
2: //-----
3:
4: class point
5: {
6:     public int x;
7:     public int y;
8: }
9:
10: class line
11: {
12:     public point starting = new point();
13:     public point ending = new point();
14: }
15:
16: class lineApp
17: {
18:     public static void Main()
19:     {
20:         line myLine = new line();
21:
22:         myLine.starting.x = 1;
23:         myLine.starting.y = 4;
24:         myLine.ending.x = 10;
25:         myLine.ending.y = 11;
26:
27:         System.Console.WriteLine("Point 1: ({0},{1})",
28:                                   myLine.starting.x, myLine.starting.y);
29:         System.Console.WriteLine("Point 2: ({0},{1})",
30:                                   myLine.ending.x, myLine.ending.y);
31:     }
32: }

```

该程序清单 的输出如下：

```

Point 1: (1,4)
Point 2: (10,11)

```

分析：该程序清单与前面的几个极其类似。第4~8行定义了point类，这与前面清单没有什么不同。但第10~14行定义了另一个类：line，它由两个point类变量（starting和ending）组成。当使用line类声明一个对象时，line类将创建两个point对象。

接下来，第20行创建了一个新的line对象myLine，这里创建对象的格式与前面介绍的相同。

第22~25行访问line类的数据成员，并给它们赋值。乍一看，这要复杂些，但视觉是有欺骗性的。将其分解后，您将发现这些代码很直观。第22行将1赋给变量myLine.starting.x，换句话说，是将myLine的starting成员的x成员设置为1，也可以说是将1赋给了myLine对象的starting成员的

x 成员。这就像是一棵树。图 6.2 说明了 myLine 类的成员及其名称。

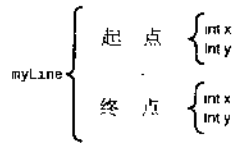


图 6.2 myLine 类的数据成员

6.4.4 类型嵌套

第 3 天介绍了可以使用的各种标准数据类型。正如您从程序清单 6.3 看到的，使用类创建的对象可以使用的地方与其他任何数据类型的变量相同。

就类本身而言，它并没有做任何工作，而只是一种描述。例如，在程序清单 6.3 中，第 4~8 行的 point 类只是一种描述，它没有声明任何东西，也没有使用内存。该描述定义了一种类型，这里是类，具体地说是 point 类。

可以将这种类型嵌套到其他类中。如果只在 line 中使用 point，则可以将 point 类的定义放在 line 类中。这样便可以在 line 类中使用 point 类。

嵌套 point 类的代码如下：

```

class line
{
    public class point
    {
        public int x;
        public int y;
    }
    public point starting = new point();
    public point ending = new point();
}
  
```

这里有些不同的地方是，point 类也必须定义为公有的。否则，将出错。这是有道理的，如果 point 本身都不是公有的，其组成部分如何能是公有的呢？

6.5 静态变量

有时候，您希望使用同一个类声明的对象可以共享一个值。例如，您可能声明许多起点都相同的 line 对象，如果其中的一个 line 对象修改了起点，则其他所有 line 对象的起点也将相应修改。

要在属于同一种类的对象之间共享数据值，可以使用限定符 static。程序清单 6.4 再次使用了 line 类，但所有的 line 类的起点都将相同。

程序清单 6.4 statline.cs：将限定符 static 用于数据成员

```

1: // statline.cs- A class with two data members
2: //-----
3:
4: class point
5: {
6:     public int x;
  
```

```
7:     public int y;
8: }
9:
10: class line
11: {
12:     static public point origin= new point();
13:     public point ending = new point();
14: }
15:
16: class lineApp
17: {
18:     public static void Main()
19:     {
20:         line line1 = new line();
21:         line line2 = new line();
22:
23:         // set line origin
24:         line.origin.x = 1;
25:         line.origin.y = 2;
26:
27:
28:         // set line1's ending values
29:         line1.ending.x = 3;
30:         line1.ending.y = 4;
31:
32:         // set line2's ending values
33:         line2.ending.x = 7;
34:         line2.ending.y = 8;
35:
36:         // print the values...
37:         System.Console.WriteLine("Line 1 start: ({0},{1})",
38:                                   line.origin.x, line.origin.y);
39:         System.Console.WriteLine("line 1 end:  ({0},{1})",
40:                                   line1.ending.x, line1.ending.y);
41:         System.Console.WriteLine("Line 2 start: ({0},{1})",
42:                                   line2.ending.x, line2.ending.y);
43:         System.Console.WriteLine("line 2 end:  ({0},{1})\n",
44:                                   line2.ending.x, line2.ending.y);
45:
46:         // change value of line2's starting point
47:         line2.origin.x = 939;
48:         line2.origin.y = 747;
49:
50:         // and the values again...
51:
52:         System.Console.WriteLine("Line 1 start: ({0},{1})",
53:                                   line2.origin.x, line2.origin.y);
54:         System.Console.WriteLine("line 1 end:  ({0},{1})",
55:                                   line1.ending.x, line1.ending.y);
56:         System.Console.WriteLine("Line 2 start: ({0},{1})",
```

```

57:             line.origin.x, line.origin.y);
58:     System.Console.WriteLine("line 2 end:  ({0},{1})",
59:                               line2.ending.x, line2.ending.y);
60: }
61: }

```

该程序清单的输出如下:

```

Line 1 start: (1,2)
Line 1 end: (3,4)
Line 2 start: (1,2)
Line 2 end: (7,8)
Line 1 start: (939,747)
Line 1 end: (3,4)
Line 2 start: (939,747)
Line 2 end: (7,8)

```

警告: 如果访问静态数据成员时使用对象名 (如 `line1`) , 将出错。访问静态数据成员时, 必须使用类名。

分析: 程序清单6.4与前面的几个差别不大, 最大的差别在第12行, 它除了将 `point` 声明为公有的外, 还将其声明为静态的。 `static` 关键字使 `line` 类有天壤之别, 这样使用 `line` 类声明的所有对象都共享同一个起点, 而不是各有一个。

从第18行起是 `Main` 例程, 第20和21行声明了两个名为 `line1` 和 `line2` 的 `line` 对象。第28~29行设置了 `line` 的终点, 而第33~34行则设置了 `line` 的起点。第24和25行包含一些与以前不同的内容, 它们设置了 `line` 类的起点坐标, 而不是设置 `line1` 或 `line2` 的起点坐标。这很重要, 如果试图设置 `line1` 或 `line2` 的起点坐标, 则编译时将出错。换句话说, 下面的代码是不正确的:

```
line1.origin.x = 1;
```

由于 `origin` 对象被声明为静态的, 因此它将在所有的 `line` 对象之间共享。由于 `line1` 和 `line2` 都不拥有该对象, 因此不能使用它们来直接设置该对象的值, 而必须使用类名。记住, 类中被声明为静态的变量属于这个类, 而不是实例化的各个对象。

第37~44行打印 `line1` 和 `line2` 的起点和终点坐标, 同样在打印起点坐标时, 使用的是类名, 而不是对象名。第47~48行修改起点的坐标, 而后面的代码再次打印各线段的起点和终点坐标。

注意: 静态数据成员的一种常见用途是用作计数器, 每当某个对象执行某种操作时, 便可以将所有对象的计数器加1。

6.6 应用程序类

您可能没有发现, 在所有的应用程序中都需要使用应用程序类, 对此我们还没有讨论过。程序清单6.4的第16行包含如下代码:

```
class LineApp
```

您将发现, 在本书的每个应用程序中都包含一个与此类似的类。C#是一种面向对象的语言, 这意味着所有的一切都是对象, 应用程序也是。要创建对象, 需要一个类来定义它。程序清单6.4的

应用程序类是 `lineApp`。当您执行该程序时，`lineApp` 类将被实例化，创建一个 `lineApp` 对象，它正是您的程序！

正如您在前面看到的，应用程序类声明数据成员。在程序清单 6.4 中，`lineApp` 类的数据成员是两个类：`line1` 和 `line2`。当然应用程序类还有其他功能，在明天的课程中您将知道，您的类中也可能包含这些功能。

6.7 属 性

前面指出过，面向对象程序的优点之一是能够控制数据的内部表示以及对数据的访问。在本章前面的范例中，所有的数据成员都被声明为公有的，因此可以在任何代码中访问这些数据成员。

在面向对象的程序中，您希望对哪些人可以访问（不可以访问）数据有更大的控制权。通常您不希望类的外面可以直接访问数据成员，如果允许这样做，您将陷于无法改变数据成员的数据类型的境地。

C# 提供了一种被称为属性（`properties`）的概念，让您能够在类中创建面向对象的字段（`field`）。属性使用关键字 `get` 和 `set` 来取得和设置变量的值。程序清单 6.5 演示了如何将 `get` 和 `set` 用于 `point` 类中。

程序清单 6.5 `prop.cs`：使用属性

```

1: // prop.cs- Using Properties
2: //-----
3:
4: class point
5: {
6:     int my_X; // my_X is private
7:     int my_Y; // my_Y is private
8:
9:     public int x
10:    {
11:        get
12:        {
13:            return my_X;
14:        }
15:        set
16:        {
17:            my_X = value;
18:        }
19:    }
20:    public int y
21:    {
22:        get
23:        {
24:            return my_Y;
25:        }
26:        set
27:        {
28:            my_Y = value;

```

```

29:     }
30: }
31: }
32:
33: class MyApp
34: {
35:     public static void Main()
36:     {
37:         point starting = new point();
38:         point ending = new point();
39:
40:         starting.x = 1;
41:         starting.y = 4;
42:         ending.x = 10;
43:         ending.y = 11;
44:
45:         System.Console.WriteLine("Point 1: ({0},{1})",
46:                                   starting.x, starting.y);
47:         System.Console.WriteLine("Point 2: ({0},{1})",
48:                                   ending.x, ending.y);
49:     }
50: }

```

该程序清单的输出如下:

```

Point 1: (1,4)
Point 2: (10,11)

```

分析: 程序清单6.5为point类的x和y坐标分别创建了属性。Point类是在第4~31行定义的, 这些代码行都是point的定义。第6~7行创建了两个数据成员my_x和my_y, 因为它们没有被声明为公有的, 因此在类的外面无法访问它们, 所有被视为私有变量。第8天的课程将更详细地介绍如何使某些东西是私有的。

第9~19行和第20~30行的类似, 只是前者使用的是变量my_x, 而后者使用的是变量my_y。这些代码分别为my_x和my_y创建属性。

第9行看起来就像是声明另一个数据成员, 事实上确实如此。这一行声明了一个名为x的int变量, 注意该行的结尾没有分号, 因此该成员变量声明还没有结束, 它还包含第10~19行的代码块。该代码块中使用了两个命令, 第11行是一条get语句, 每当程序试图取得正在声明的数据成员(这里为x)的值时, 该语句便被调用。例如, 如果您将x的值赋给其他变量, 则首先需要取得x的值, 然后将其放在该变量中。每当您设置变量x的值时, 第15行的set语句都将被调用。例如将x的值设置为10, 将把10放在my_x中。

当程序要取得x的值时, 第11行的get属性将被调用, 这将执行get内的代码, 即第13行。第13行返回my_x的值, 而my_x是point类的私有变量。

当程序设置x的值时, 第15行的set属性将被调用。这将执行set内的代码, 即第17行。第17行将value的值赋给point类的私有变量my_x。value是x的值。例如, 执行下面的语句后, value的值将为10:

```
x = 10;
```

该语句将 `x` 的值设置为 10，而 `x` 内的 `set` 属性将把这个值赋给 `my_x`。

从应用程序的主要部分（第 33 ~ 50 行）可以知道，`x` 的使用方式与以前相同。在使用 `point` 类方面，与以前没有任何差别。差别在于，可以修改 `point` 类，以不同的方式来存储 `my_x` 和 `my_y`，而不会对程序产生任何影响。

虽然第 9 ~ 30 行的代码很简单，但不一定非得如此。在 `get` 和 `set` 中，可以包含任何代码，也可以执行任何操作。您甚至不一定非得将值写入到另一个数据成员中。

应该

进入第 6 天的课程之前，务必要理解今天介绍的关于数据成员和类的知识

创建程序时，务必使用属性存取器（accessor）来访问类的数据成员

6.8 名称空间

开始学习类后，必须知道存在大量可以完成各种功能的类。NET 框架提供了大量的基类供您使用。另外，您还可以获得第三方的类。

注意：第 16 天的课程将专门介绍如何使用许多重要的 .NET 基类。

随着您往下阅读本书，将了解很多重要的类。您实际上已经使用过几个基类。前面提到过，`Console` 是一个基类。您还知道，`Console` 有很多例程，其中的两个叫做 `Write` 和 `WriteLine`。例如，下面的代码将作者的姓名显示到控制台中：

```
System.Console.WriteLine("Bradley L. Jones");
```

您知道，“Bradley L. Jones”是一个字符串值，而 `WriteLine` 是 `Console` 类的一个例程。您甚至还知道 `Console` 是一个使用类声明的对象。您唯一不了解的是 `System`。

由于类的数量非常大，因此以一定的方式对其进行组织至关重要。可以将类集中在一起，形成名称空间。名称空间是一组被命名的类。`Console` 类属于 `System` 名称空间。

`System.Console.WriteLine` 是一个全限定名称。使用全限定名称，可以直接指定代码所在的位置。C# 提供一种使用类和方法的简捷方式，无需总是指定全限定名称。这是通过使用关键字 `using` 实现的。

关键字 `using` 让您能够在程序中包含名称空间，这样程序便知道到哪个名称空间中去搜索其中使用的类和例程。包含名称空间的格式如下：

```
using namespace_name
```

其中 `namespace_name` 是名称空间或嵌套名称空间的名称。例如，要包含名称空间 `System`，可以在程序清单的开始位置包含下面的代码：

```
using System;
```

包含上述代码后，在使用 `System` 名称空间中的类和例程时，便可以省略 `System`。程序清单 6.6 使用 `using` 语句来包含 `System` 名称空间。

程序清单 6.6 namesp.cs: 使用 using 和名称空间

```
1: // namesp.cs- Namespaces and the using keyword
```

```
2: //-----
3:
4: using System;
5:
6: class name
7: {
8:     public string first;
9:     public string last;
10: }
11:
12: class NameApp
13: {
14:     public static void Main()
15:     {
16:         // Create a name object
17:         name you = new name();
18:
19:         Console.Write("Enter your first name and press enter: ");
20:         you.first = Console.ReadLine();
21:         System.Console.Write("\n{0}, enter your last name and press enter: ",
22:                               you.first);
23:         you.last = System.Console.ReadLine();
24:
25:         Console.WriteLine("\nData has been entered.....");
26:         System.Console.WriteLine("You claim to be {0} {1}",
27:                                   you.first, you.last);
28:     }
29: }
```

该程序清单的输出如下:

```
Enter your first name and press enter: Bradley

Bradley, enter your last name and press enter: Jones

Data has been entered.....
You claim to be Bradley Jones
```

注意: 输出中的粗体文本是作者运行程序时输入的, 您可以输入任何文本。建议您输入自己的姓名, 而不是作者的。

分析: 第4行是该程序的重点。关键字 `using` 将名称空间 `System` 包含进来, 这样当您使用 `Console` 类中的函数时, 便无需全限定其名称了, 如第19、20和25行所示。使用 `using` 关键字后, 仍可以使用全限定名称, 如第21、23和26行所示。但没有必要这样做, 因为名称空间已经被包含进来了。

该程序使用了 `Console` 类的另一个例程 `ReadLine`, 通过运行这个程序可以知道, `ReadLine` 读取用户在按 `Enter` 键之前输入的信息, 并返回用户输入的信息。在这个程序中, 用户输入的信息被赋给 `name` 类的一个数据成员。

6.8.1 嵌套名称空间

多个名称空间可以存在一起，即作为一个名称存储。如果一个名称空间包含其他名称空间，则可以通过限定名称，或在 `using` 语句中使用子名称空间来包含它们。例如，`System` 名称空间包含其他几个名称空间，如 `Drawing`、`Data` 和 `Windows.Forms`，要使用这些名称空间中的类，可以使用限定名称，或使用 `using` 语句来包含它们。要使用 `using` 语句来包含 `System` 名称空间中的 `Data` 名称空间，可以使用下面的代码：

```
using System.Data;
```

6.9 总 结

今天和明天的课程是本书最重要的内容之一。类是面向对象编程语言的关键所在，因此也是 C# 的关键所在。今天重温了封装、多态、继承和重用等概念。然后介绍了如何定义类的基本结构以及如何创建类的数据成员。学习如何使用存取器 `set` 和 `get` 创建属性时，您学会了最重要的、封装程序的方法之一。今天最后介绍了名称空间和 `using` 语句。

6.10 问与答

问：您是否会创建只包含数据成员的类？

答：通常不会。类和面向对象编程的价值在于能够将功能和数据封装到一个包中。今天只介绍了类的数据成员，明天将介绍如何给类添加功能。

问：是否应该将所有的数据成员都声明为公有的，以便其他人能够访问它们？

答：完全不是这样。虽然在今天的课程中，很多数据成员都被声明为公有的，但由于某些原因，有时候您不希望其他人能够访问您的数据。原因之一是以便能够修改数据的存储方式。

问：前面指出过，有很多已有的类，如何了解这些类？

答：微软公司提供了大量的、被称为 .NET 基类的类，还提供了关于这些类的功能的文档。这些类是以名称空间的方式组织的，编写本书时，了解这些类的唯一途径是通过在线帮助。微软公司提供了关于这些基类的完全参考。第 19 天的课程将更详细地介绍这些基类。

6.11 作 业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

6.11.1 小测验

1. 面向对象程序的四个特征是什么？
2. 类中可以存储哪两种重要的东西？
3. 被声明为公有的数据成员和没有被声明为公有的数据成员之间的差别何在？
4. 给数据成员加上关键字 `static` 有何作用？
5. 程序清单 6.2 的应用程序类的名称是什么？
6. 哪些命令被用于实现属性？

7. `value` 有何用途?
8. `Console` 是类、数据成员、名称空间、例程还是数据类型?
9. `System` 是类、数据成员、名称空间、例程还是数据类型?
10. 使用哪个关键字来包含名称空间?

6.11.2 练习

1. 创建一个类来保存圆心和半径。
2. 在练习 1 创建的 `Circle` 类中添加一个属性。
3. 创建一个类，它存储一个 `int` 数据成员 `MyNumber`，并给该数据成员创建属性，当该数据成员被存储时，将其乘以 100；当其被读取时，将其除以 100。
4. 下面的程序有问题，请在编辑器中输入该程序，并编译它。哪一行导致错误消息?

```

1: // A bug buster program
2: // Is something wrong? Or not?
3: //-----
4: using System;
5: using System.Console;
6:
7: class name
8: {
9:     public string first;
10: }
11:
12: class NameApp
13: {
14:     public static void Main()
15:     {
16:         // Create a name object
17:         name you = new name();
18:
19:         Write("Enter your first name and press enter: ");
20:         you.first = ReadLine();
21:         Write("\nHello {0}!", you.first);
22:     }
23: }

```

5. 编写一个 `die` 类，它存储骰子的面数 (`sides`) 以及当前掷骰子得到的点数 (`value`)。
6. 在程序中使用练习 5 的类声明两个骰子对象，并设置这些对象的 `sides` 数据成员，然后将骰子的点数设置为一个随机值。

第7天课程

类方法和成员函数

昨天介绍过，类有几个组成部分。最重要的是，类能够定义用于存储数据和例程的对象。昨天介绍了如何存储数据，今天将介绍如何在类中创建、存储和使用例程。例程让对象能够实现您所需的功能。虽然存储信息也很重要，但操作数据使程序具有生命力。今天将介绍以下内容：

- 创建自己的方法；
- 使用参数将信息传递给例程；
- 值和引用的概念；
- 调用方法的概念；
- 构造函数；
- 释放类。

7.1 方法初步

前几天介绍了如何存储和操纵数据以及如何控制程序流程。现在需要将其其中的一些功能封装到可重用的例程中。另外，还需要将这些例程和类的数据成员关联起来。

在C#中，例程被称为函数或方法，这两个术语之间并没有什么差别，可以交替使用。

注意：大多数C++和C#开发人员将例程称为方法，而大多数C程序员将其称为函数。不管您管它们叫什么，指的都是同一回事。

新术语：方法是一个带名称的、独立的、以可重用的方式放置的代码片段。方法无需应用程序其他部分的干预便能运行，如果创建的正确，则能够执行其名称指示的特定任务。

通过今天的课程，您将知道方法可以返回一个值。另外，许多方法还允许您传递信息给它们。

7.2 使用方法

在本书的前面，您已经使用过几个方法：Write、WriteLine 和 ReadLine。另外，在每一个程序中，您都使用了 Main 方法。程序清单 7.1 包含了前面介绍过的 Circle 类，不过这里将计算面积和周长的例程作为方法加入到了这个类中。

程序清单 7.1 circle.cs: 一个包含成员方法的类

```
1: // circle.cs - A simple circle class with methods
2: //-----
3:
4: class Circle
5: {
6:     public int x;
7:     public int y;
8:     public double radius;
9:
10:    public double area()
11:    {
12:        double theArea;
13:        theArea = 3.14159 * radius * radius;
14:        return theArea;
15:    }
16:
17:    public double circumference()
18:    {
19:        double theCirc;
20:        theCirc = 2 * 3.14159 * radius;
21:        return theCirc;
22:    }
23: }
24:
25: class CircleApp
26: {
27:     public static void Main()
28:     {
29:         Circle first = new Circle();
30:         Circle second = new Circle();
31:
32:         double area;
33:         double circ;
34:         first.x = 10;
35:         first.y = 14;
36:         first.radius = 3;
37:
38:         second.x = 10;
39:         second.y = 11;
40:         second.radius = 4;
41:
42:         System.Console.WriteLine("Circle 1: Center = ({0},{1})",
43:                                   first.x, first.y);
44:         System.Console.WriteLine("        Radius = {0}", first.radius);
45:         System.Console.WriteLine("        Area   = {0}", first.area());
46:         System.Console.WriteLine("        Circum = {0}", first.circumference());
47:
48:         area = second.area();
```



```
49:     circ = second.circumference();
50:
51:     System.Console.WriteLine("\nCircle 2: Center = ({0},{1})",
52:                               second.x, second.y);
53:     System.Console.WriteLine("        Radius = {0}", second.radius);
54:     System.Console.WriteLine("        Area   = {0}", area);
55:     System.Console.WriteLine("        Circum = {0}", circ);
56: }
57: }
```

该程序清单的输出如下:

```
Circle 1: Center = (10,14)
        Radius = 3
        Area   = 28.27431
        Circum = 18.84954

Circle 2: Center = (10,11)
        Radius = 4
        Area   = 50.26544
        Circum = 25.13272
```

分析: 该程序清单的大部分内容对您来说都是熟悉的, 阅读完今天的课程后, 那些不熟悉的部分便会变得熟悉了。

来看一下该程序清单, 从第4行开始是 Circle 类的定义, 第6~8行声明的三个数据成员与前一天的范例相同, 它们是用于存储圆心坐标的 x 和 y 以及用于存储半径的变量 radius。数据成员声明之后的内容仍属于类定义。

第10~15行是第一个成员方法的定义, 对其工作原理的详细解释见接下来的几节。但现在您知道该方法的名称为 area。第12~14行是该方法的代码, 它们计算圆的面积并将其返回给调用程序。第12~13行您应该熟悉, 关于第14行今天课程的后面将更详细地介绍。第17~22行是第二个方法, 该方法名为 circumference, 它计算圆的周长, 并将其返回给调用程序。

从第25行开始是该程序清单的应用程序类。第27行包含 Main 方法, 每个应用程序的入口都是 Main 方法。该例程创建两个 Circle 对象(第29和30行), 然后给其数据成员赋值(第34~40行)。第42和43行打印第一个圆的数据成员。而第45和46行包含了以前使用过的 Console.WriteLine 方法, 区别在于这里打印的是您传递的值。在第45行, 您传递的是 first.area, 这将调用第一个圆的成员方法 area, 该方法是在第10~15行定义的。第48行调用第二个圆的 area 方法, 并将结果赋给变量 area。

提示: 在程序清单中, 您知道 area 是一个成员方法, 而不是数据成员, 因为调用它时, 后面跟有圆括号。后面将对此做更详细的介绍。

如果您还没有执行该程序清单, 现在应该执行它, 以看看发生的情况。接下来的几节将详细介绍如何定义自己的方法以及方法的工作方式。另外, 还将介绍如何给方法传递值以及接收方法返回的值。

7.3 包含方法的程序的流程

前面介绍过，方法是一个独立的代码片段，被封装和命名，以便能够在程序中调用它。方法被调用时，程序流程将切换到方法，执行方法的代码，然后返回。图 7.1 说明了程序清单 7.1 的流程。方法被调用时，程序流程将切换到方法，执行方法的代码，然后返回到调用例程。另外，方法也可以调用其他的方法，流程与上面介绍的相同。

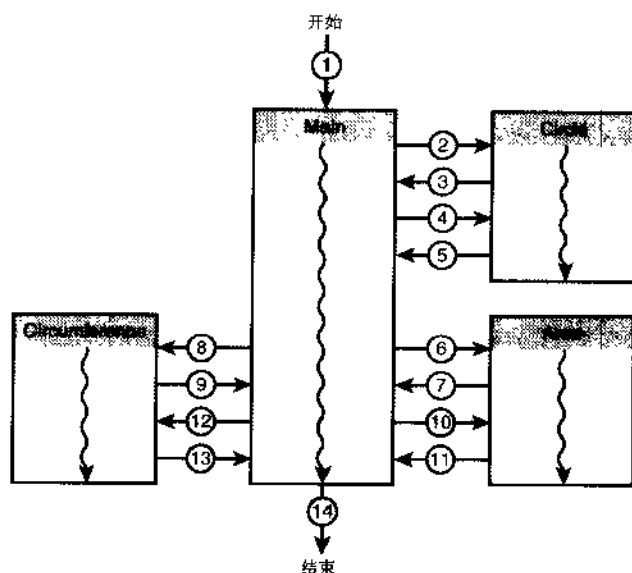


图 7.1 程序清单 7.1 的程序流程

7.4 方法的格式

了解方法的格式很重要。程序清单 7.1 以说明了调用方法的格式和步骤。方法的基本格式如下：

```

Method header
{
    Method body
}
  
```

7.4.1 方法头

方法头定义了关于方法的几项内容：

- 对方法的访问权限；
- 返回的数据类型；
- 传递给方法的值；
- 方法的名称。

从程序清单 7.1 的第 10 行可以知道，area 的方法头如下：

```
public double area( )
```

该方法被声明为公有的，即可以被类外面的程序访问，另外，该方法的返回数据类型为 `double`，即可以给调用程序返回一个 `double` 类型的值。该方法的名称为 `area`。最后，由于圆括号内为空，因此不用给 `area` 方法传递任何值，它使用同一个类的数据成员的值。稍后，将介绍如何给方法传递信息。

警告：方法头不以分号结束，如果在方法头的结尾处加上分号，将出错。

7.4.2 方法的返回数据类型

可以声明方法的返回类型，这是在方法头中指定的。方法的返回数据类型可以是任何合法的数据类型。

在方法体中，必须将一个返回数据类型的值返回给调用该方法的程序。要从方法返回一个值，可以使用关键字 `return`，后面跟一个数据类型为方法头中指定的值或变量。例如，在程序清单 7.1 中，`area` 方法的返回数据类型被声明为 `double`，而第 14 行使用关键字 `return` 返回了一个 `double` 类型的变量。`area` 方法将一个 `double` 值返回给调用程序。

如果方法不需要返回值，怎么办呢？此时应使用哪种数据类型呢？如果方法不返回值，则使用关键字 `void`，该关键字指示方法不返回值。

7.4.3 给方法命名

给方法指定合适的名称很重要。在给方法命名方面，有多种理论。您需要确定哪种理论最适合您或您的单位。一种放之四海皆准的规则是，给方法取一个有意义的名称。如果方法计算并返回面积，则使用名称 `area`、`GetArea`、`CalculateArea` 和 `CalcArea` 等是有意义的。诸如 `routine1` 和 `myRoutine` 等的含义不明确。

对于给方法命名，一种流行的准则是，总是结合使用一个动词和一个名词。因为方法执行某种操作，因此总是可以组合使用一个动词和一个名词。按照这种准则，则 `area` 的含义不太明确，而 `CalculateArea` 和 `CalcArea` 等是更佳的选择。

7.4.4 方法体

方法体包含方法被调用时所执行的代码。这些代码以左花括号开始，以右花括号结束。花括号之间的代码可以包括本书前面介绍的任何编程元素。通常代码只修改其所属类的数据成员或传递给方法的数据。

如果方法头指示了返回数据类型，则方法返回的值的类型必须与之相同。您使用关键字 `return` 来返回值，关键字 `return` 的后面跟要返回的值。再来看一下程序清单 7.1 中的 `area()` 方法，其方法体为第 11~15 行，它计算圆的面积，并将结果赋给 `double` 变量 `theArea`，然后在第 14 行使用 `return` 语句将这个值返回。

警告：方法返回的变量的数据类型必须与方法头中指定的数据类型相同。

7.4.4.1 使用方法

要使用方法，调用它即可。调用方法的方式与调用数据成员相同，即使用对象名和方法名，并在它们之间加上句点。调用方法和调用数据成员的区别在于，调用方法必须包含括号，还可能包含参数。在程序清单 7.1 中，使用下面的代码调用了第一个对象的 `area` 方法：

```
first.area( )
```

和变量一样，如果方法有返回值，则它是在方法被调用时返回的。例如，方法 `area` 将圆的面积

作为 `double` 值返回。在程序清单 7.1 的第 45 行，这个值被作为参数返回给另一个方法：`Console.WriteLine`。在第 48 行，第二个对象的 `area` 方法的返回值被赋给另一个变量：`area`。

7.4.4.2 在方法中使用数据成员

在程序清单 7.1 中，方法 `area` 使用了数据成员 `radius`，而未指定类名或对象名。该方法的代码如下：

```
public double area()
{
    double theArea;
    theArea = 3.14159 * radius * radius;
    return theArea;
}
```

以前，使用数据成员时，需要包含对象名，而这里使用 `radius` 时并没有包含对象名。例程为何可以省略对象名呢？仔细想想，则答案很简单。

当 `area` 方法被调用时，将使用特定的对象进行调用。当您使用 `circle1` 调用该方法时，`circle1` 中的所有常规数据成员和其他方法都将是可用的。之所以不需要使用对象名，是因为已经位于特定的对象的成员方法中。

另外，在类的成员方法中，还可以声明其他变量。这些变量只在方法运行时有效，它们被称为方法的局部变量。在 `area` 方法中，创建并使用了一个 `double` 变量 `theArea`。方法运行完毕后，存储在变量 `theArea` 中的值以及 `theArea` 变量本身都将不复存在。

程序清单 7.2 演示了局部变量的用法和程序流程。

程序清单 7.2 `locals.cs`: 使用局部变量和类变量

```
1: // locals.cs - Local variables
2: //-----
3:
4: using System;
5:
6: class loco
7: {
8:     public int x;
9:
10:    public void count_x()
11:    {
12:        int x;
13:
14:        Console.WriteLine("In count_x method. Printing X values...");
15:        for ( x = 0; x <= 10; x++)
16:        {
17:            Console.Write("{0} - ", x);
18:        }
19:        Console.WriteLine("\nAt the end of count_x method. x = {0}", x);
20:    }
21: }
22:
23: class CircleApp
```

```

24: {
25:     public static void Main()
26:     {
27:         loco Locals = new loco();
28:
29:         int x = 999;
30:         Locals.x = 555;
31:
32:         Console.WriteLine("\nIn Main(), x = {0}", x);
33:         Console.WriteLine("Locals.x = {0}", Locals.x);
34:         Console.WriteLine("Calling Method");
35:         Locals.count_x();
36:         Console.WriteLine("\nBack From Method");
37:         Console.WriteLine("Locals.x = {0}", Locals.x);
38:         Console.WriteLine("In Main(), x = {0}", x);
39:     }
40: }

```

该程序清单的输出如下：

```

In Main(), x = 999
Locals.x = 555
Calling Method
In count_x method. Printing X values...
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -
At the end of count_x method. x = 11

Back From Method
Locals.x = 555
In Main(), x = 999

```

分析：该程序清单使用的各种名称的含义不明显，但说明了一些要点。该程序清单中声明了几个名为`x`的变量，这包括第8行声明的公有的`int`变量`x`（属于`loco`类的数据成员）、第12行声明的局部`int`变量`x`（属于方法`count_x`）和第29行声明的变量`x`（属于`Main`方法）。虽然三个变量的名称相同，但它们是不同的变量。

第一个`x`变量（属于`loco`类）最容易理解，它属于一个类。前面介绍过，要在`loco`类的外面使用该变量，必须指定对象名，如第30行所示，这里使用名为`Locals`的对象来将该对象的数据成员`x`的值设置为555。在第29行，`Main`例程的`x`变量的值被设置为999。在第32~33行，这两个`x`变量包含各自的值，它们之间易于区分。

在第35行，`Main`方法调用了第27行声明的`loco`对象`Locals`的`count_x`方法。进入该方法后，首先声明了一个名为`x`的变量（第12行），该变量将以前声明的所有`x`变量（包括类中声明的`x`变量）屏蔽掉。该方法的余下部分使用变量`x`进行循环并打印数字，循环结束后，将最后一次打印`x`的值，然后方法便执行完毕。

该方法执行完毕后，将返回到`Main`方法，再次打印变量`x`的值。`Locals`的数据成员`x`以及`Main`内的局部变量`x`的值均未受影响，它们的值仍然分别是555和999。这些`x`变量都是相互独立的。

如果要在`count_x`方法中使用类数据成员`x`，该如何办呢？前面介绍过，在类的方法中使用数

据成员时，无需指定对象名。事实上，也无法使用对象名，因为对象名是不确定的。那么，当方法中包含局部变量 `x` 时，如何使用数据成员 `x` 呢？程序清单 7.3 对 7.2 稍做了修改。

程序清单 7.3 locals2.cs: 在方法中使用数据成员

```

1: // locals.cs - Local variables
2: //-----
3:
4: using System;
5:
6: class loco
7: {
8:     public int x;
9:
10:    public void count_x()
11:    {
12:        int x;
13:
14:        Console.WriteLine("In count_x method. Printing X values...");
15:        for ( x = 0; x <= 10; x++)
16:        {
17:            Console.Write("{0} - ", x);
18:        }
19:
20:        Console.WriteLine("\nDone looping. x = {0}", x);
21:        Console.WriteLine("The data member x's value: {0}", this.x);
22:        Console.WriteLine("At the end of count_x method.");
23:    }
24: }
25:
26: class CircleApp
27: {
28:     public static void Main()
29:     {
30:         loco Locals = new loco();
31:
32:         int x = 999;
33:         Locals.x = 555;
34:
35:         Console.WriteLine("\nIn Main(), x = {0}", x);
36:         Console.WriteLine("Locals.x = {0}", Locals.x);
37:         Console.WriteLine("Calling Method");
38:         Locals.count_x();
39:         Console.WriteLine("\nBack From Method");
40:         Console.WriteLine("Locals.x = {0}", Locals.x);
41:         Console.WriteLine("In Main(), x = {0}", x);
42:     }
43: }

```

该程序清单的输出如下：

```

In Main(), x = 999
Locals.x = 555
Calling Method
In count_x method. Printing X values...
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -
Done looping. x = 11
The data member x's value: 555
At the end of count_x method.

Back From Method
Locals.x = 555
In Main(), x = 999

```

分析：该程序清单独特的地方是第21行，它打印一个名为`this.x`的值。关键字`this`指的是当前使用的对象。在这里，`this`指的是其`count_x`方法被调用的`Locals`对象。由于`this`指的是当前对象，因此`x`指的是该对象的数据成员`x`，而不是局部变量`x`。因此，要在方法中访问其所属类的数据成员，可以使用关键字`this`。

那么类的方法如何访问调用方法中的变量`x`（在程序清单 7.3 的 `Main` 方法中声明的局部变量 `x`，如第 32 行所示）呢？除非将其作为参数传递给该方法，否则无法访问。

7.5 给方法传递值

您已经知道了如何访问方法、如何在方法中声明局部变量以及如何使用方法所属类的数据成员。如果要使用其他类或另一个方法的一个值或多个值，该如何办呢？例如，假设您需要一个接受两个数字，将其相乘并返回结果的方法。您已经知道如何返回单个值，但如何在方法中获得这两个数字呢？

方法被调用时，可以接受值。为此，必须在方法头中定义参数。包含参数的方法头的格式如下：

```
Modifiers Return Type Name( Parameters)
```

参数是在括号内传递的，参数是可选的，如果没有参数，则括号内将为空，就像前面的范例那样。

传递参数的基本格式如下：

```
[Attribute] Type ArgumentName
```

其中 `Type` 是要传递的参数的数据类型，`ArgumentName` 是要传递的变量的名称。另外还可以指定属性，这将在本章后面介绍。属性是可选的。程序清单 7.4 中的程序包含一个简单的方法，它接受两个数字，将其相乘，然后将结果返回。

程序清单 7.4 Mult.cs: 传递值

```

1: // mult.cs - Passing values
2: //-----
3:
4: using System;
5:

```

```

6: class Multiply
7: {
8:     static long multi( long nbr1, long nbr2 )
9:     {
10:         return (nbr1 * nbr2);
11:     }
12:
13:     public static void Main()
14:     {
15:         long x = 1234;
16:         long y = 5678;
17:         long a = 6789;
18:         long b = 9876;
19:
20:         long result;
21:
22:         result = multi( x, y);
23:         Console.WriteLine("x * y : {0} * {1} = {2}", x, y, result);
24:
25:         result = multi(a, b);
26:         Console.WriteLine("a * b : {0} * {1} = {2}", a, b, result);
27:
28:         result = multi( 555L, 1000L );
29:         Console.WriteLine("With Long values passed, the result was {0}", result);
30:     }
31: }

```

该程序清单的输出如下:

```

x * y : 1234 * 5678 = 7006652
a * b : 6789 * 9876 = 67048164
With Long values passed, the result was 555000

```

分析: 该程序清单说明传递两个值的要点, 同时说明了其他东西。首先来看一下第8~11行的方法定义, 该方法名为Multi, 接受两个参数。这两个参数都是long型的, 分别名为nbr1和nbr2, 它们是该方法内的局部变量。第10行将这两个值相乘, 并将结果返回给调用者。在第8行的方法头中, 方法multi的返回值被声明为long, 因此它能够返回一个long值。

第22、25和28行使用不同的值分别调用了multi方法。从中可以知道, 您可以传递变量(如第22和25行所示), 也可以传递字面值(如第28行所示)。当multi方法被调用时, 传递的值将被发送给该方法, 并使用方法头中的变量名来引用它们。因此, 对于第22行, x和y分别被传递给nbr1和nbr2; 在第25行, a和b分别被传递给nbr1和nbr2; 在第28行, 555和1000分别被传递给nbr1和nbr2。然后, 方法将使用这些传递来的值。

注意, 传递给方法的参数数目必须与定义的相同。就multi方法而言, 您必须传递两个值, 否则将出错。

注意: 对于使用像C++这样的语言编写过程序的读者, 需要注意的是, 在C#中, 参数没有默认值。

7.5.1 静态方法

前面介绍过, `static` 限定符使数据成员与类而不是特定的对象关联在一起。程序清单 7.4 中使用了一个静态方法, 该程序清单只包含一个类, 其中不但包含 `Main` 方法, 还包含 `multi` 方法。

也可以像前面那样, 将 `multi` 方法放在另一个类中。如果这样做, 则需要声明该类的一个对象, 然后使用对象名调用 `multi` 方法。由于这里方法 `multi` 和 `Main` 位于同一个类中, 因此将它声明为静态的, 这样在 `Multiply` 类的任何地方都可以调用它, 当然在 `Main()` 方法中也能使用它。

7.5.2 不同的参数访问属性

在前一个范例中, 您给方法传递的是值。方法将获得这些值的一个拷贝, 然后使用这些拷贝, 当方法运行完毕后, 这些拷贝将被丢弃。按值传递只是其中的一种传递方式, 参数的访问属性有三类:

- 值;
- 引用;
- 输出 (Out)。

7.5.2.1 以值的方式访问参数

正如前面指出的, 以值的方式访问参数时, 参数的一个拷贝将被传递给方法, 然后方法使用这些拷贝, 而原来的值不受影响。

7.5.2.2 以引用的方式访问参数

有时候, 需要修改原来的变量中存储的值。在这种情况下, 可以传递到变量的引用, 而不是变量的值。引用是一个变量, 它可以访问原来的变量。修改引用, 将修改原来变量的值。

从技术上说, 引用变量指向内存中的一个位置, 其中存储了数据。请看图 7.2, 变量的值存储在内存中, 可以创建一个引用, 它指向变量在内存中的位置。当引用被修改时, 修改的是内存中的值, 因此变量的值也将被修改。

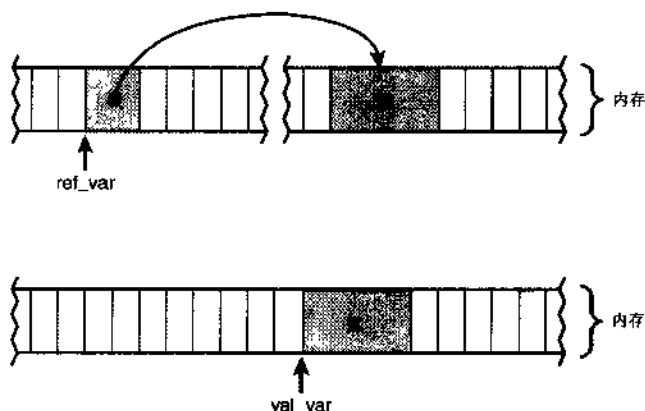


图 7.2 引用变量和值变量

由于引用变量可以指向不同的位置, 因此它不像常规变量那样, 总是对应于内存中的特定位置。当方法被调用时, 如果该方法包含属性为引用的参数, 则该参数将指向被传递给方法的相应变量。由于指向的是变量在内存中的位置, 因此对参数变量的修改, 也将导致原来的变量被相应修改。

声明参数时, 默认情况下, 其属性为其数据类型的属性。对于基本数据类型, 为按值。要使基本数据类型 (如 `int`) 参数按引用传递, 需要在方法头中的数据类型前面加上关键字 `ref`。程序清单

7.5 和 7.6 演示了如何使用关键字 `ref` 以及值参数和引用参数之间的区别。在程序清单 7.6 中，一个 `double` 参数是按引用传递的。从其输出可以知道这两个程序清单之间的区别。

注意：正如第3天的课程介绍的，默认情况下，基本数据类型为值类型。这意味着，当您创建一个变量时，系统将从内存中分配特定的单元，供它存储自己的值。诸如类等数据类型默认为引用类型，这意味着类名包含的是类数据的存储地址，而不是数据本身。

程序清单 7.5 `refnot.cs`: 按值传递参数

```

1: // refnot.cs - without reference variables (by value)
2: //-----
3:
4: using System;
5:
6: class nbr
7: {
8:     public double square( double x )
9:     {
10:         x = x * x;
11:         return x;
12:     }
13: }
14:
15: class TestApp
16: {
17:     public static void Main()
18:     {
19:         nbr doit = new nbr();
20:
21:         double nbr1 = 3;
22:         double retVal = 0;
23:
24:         Console.WriteLine("Before square -> nbr1 = {0}, retVal = {1}",
25:                             nbr1, retVal);
26:
27:         retVal = doit.square( nbr1 );
28:
29:         Console.WriteLine("After square -> nbr1 = {0}, retVal = {1}",
30:                             nbr1, retVal);
31:     }
32: }
```

该程序清单的输出如下：

```

Before square -> nbr1 = 3, retVal = 0
After square -> nbr1 = 3, retVal = 9
```

程序清单 7.6 `refers.cs`: 按引用传递参数

```

1: // refers.cs - Reference variables
2: //-----
```

```
3:
4: using System;
5:
6: class nbr
7: {
8:     public double square( ref double x )
9:     {
10:         x = x * x;
11:         return x;
12:     }
13: }
14:
15: class TestApp
16: {
17:     public static void Main()
18:     {
19:         nbr doit = new nbr();
20:
21:         double nbr1 = 3;
22:         double retVal = 0;
23:
24:         Console.WriteLine("Before square -> nbr1 = {0}, retVal = {1}",
25:             nbr1, retVal);
26:
27:         retVal = doit.square( ref nbr1 );
28:
29:         Console.WriteLine("After square -> nbr1 = {0}, retVal = {1}",
30:             nbr1, retVal);
31:     }
32: }
```

该程序清单的输出如下:

```
Before square -> nbr1 = 3, retVal = 0
After square -> nbr1 = 9, retVal = 9
```

分析: 这两个程序清单的输出说明了发生的情况, 程序清单7.5使用的不是引用, 因此传递给square方法的变量的值没变, 在方法调用前后, 该变量的值都是3。在程序清单7.6中, 传递的是一个引用, 因此传递给square的变量的值被修改, 如该程序清单的输出所示。

在程序清单7.6中, 在方法头中声明参数时, 加上了关键字 `ref`。正常情况下, `double` 是一种值数据类型, 因此, 要将其作为引用传递给方法, 必须在被传入的变量前面加上关键字 `ref`, 如第27行所示。

7.5.2.3 以输出方式访问参数

通过指定返回类型, 可以从方法返回一个值, 但有时候, 需要返回多个值。虽然可以使用引用变量来完成这样的工作, 但 C# 提供了一种特殊的属性类型, 专门用于从方法返回数据。

可以在方法头中加入专门用于返回值的参数。这种参数的声明方式与其他参数相同, 只是需要将其属性设置为关键字 `out`。该关键字指示从方法返回一个值, 但不需要传入。调用包含输出参数的

方法时，一定要包含用于存储返回值的变量。程序清单 7.7 演示了如何使用 `out` 属性。

程序清单 7.7 Outer.cs: 使用 out 属性

```

1: // outter.cs - Using output variables
2: //-----
3:
4: using System;
5:
6: class nbr
7: {
8:     public void math_routines( double x,
9:                               out double half,
10:                              out double squared,
11:                              out double cubed )
12:     {
13:         half = x / 2;
14:         squared = x * x;
15:         cubed = x * x * x;
16:     }
17: }
18:
19: class TestApp
20: {
21:     public static void Main()
22:     {
23:         nbr doit = new nbr();
24:
25:         double nbr = 600;
26:         double Half_nbr = 0;
27:         double Squared_nbr = 0;
28:         double Cubed_nbr = 0;
29:
30:         Console.WriteLine("Before method -> nbr = {0}", nbr);
31:         Console.WriteLine("        Half_nbr = {0}", Half_nbr);
32:         Console.WriteLine("        Squared_nbr = {0}", Squared_nbr);
33:         Console.WriteLine("        Cubed_nbr = {0}\n", Cubed_nbr);
34:
35:         doit.math_routines( nbr, out Half_nbr, out Squared_nbr, out Cubed_nbr );
36:
37:         Console.WriteLine("After method -> nbr = {0}", nbr);
38:         Console.WriteLine("        Half_nbr = {0}", Half_nbr);
39:         Console.WriteLine("        Squared_nbr = {0}", Squared_nbr);
40:         Console.WriteLine("        Cubed_nbr = {0}", Cubed_nbr);
41:     }
42: }

```

该程序清单的输出如下：

```

Before method -> nbr = 600
        Half_nbr = 0

```

```

Squared_nbr = 0
Cubed_nbr = 0

After method -> nbr = 600
Half_nbr = 300
Squared_nbr = 360000
Cubed_nbr = 216000000

```

分析：该程序清单中需要说明的有两部分。首先是第8~11行的方法头。还记得吗，可以使用空白来提供代码的可读性。该方法被格式化成每个参数占据一行。第一个参数是一个常规的双精度变量，其他三个参数的属性被声明为out。这意味着，这三个参数不是将值传入到方法内，而是从方法中传值。

需要说明的第2部分代码是第35行，它调用方法 `math_routines`。调用方法时，传递的参数的属性也必须指定为out，如果省略关键字out，将出错。

就整体而言，该程序清单还是相对比较简单。nbr类中的 `math_routines` 使用一个数字执行三种运算。第30~33行打印用来存储数学计算结果的变量的值。第37~40行打印这些变量在 `math_routines` 方法中被填充后的值。

如果不给输出参数设置值，将出错。必须给方法的所有输出参数赋值。例如，请将程序清单的第14行注释掉：

```
14:         squared = x * x;
```

这样输出变量 `squared` 将不被赋值。重新编译该程序清单时，将出现如下所示的错误：

```
outterz.cs(8,17): error CS0177: The out parameter 'squared' must be assign to before control
leaves the current method
```

从上面可以知道，如果定义了一个输出参数，则必须给它赋值。

注意：将变量用作输出变量时，在调用方法前，可以不初始化。例如，可以删除第26~28行的初始化部分，变成如下所示：

```

26:         double Half_nbr;
27:         double Squared_nbr;
28:         double Cubed_nbr;

```

当然，在这种情况下，还需要删除第31~33行。重新编译该程序清单，第35行的方法调用将不会出现任何问题，虽然这三个变量没有被初始化。

应该	不应该
应给类和方法指定易懂的、描述性的名称	不要将值变量和引用变量混淆，按值传递变量将创建变量的一个拷贝，按引用传递变量将可以操纵原来变量的值

7.6 类方法的类型

前面介绍了使用方法的基本知识，您还需了解一些特殊的方法：

- 属性存取器方法；

- 构造函数;
- 析构函数。

7.6.1 属性存取器方法

您在昨天实际上使用过属性存取器方法: set 和 get, 这些方法让您能够使数据成员为私有的。

7.6.2 构造函数

有时候, 在对象被创建时, 您想做一些设置。有一种特殊的方法专门用于完成对象初始设置或构造, 这种方法叫做构造函数。实际上存在两种构造函数: 实例构造函数和静态构造函数, 前者在每个实例或对象创建时被调用, 后者在类的第一个对象被创建之前被调用。

7.6.2.1 实例构造函数

实例构造函数是一个方法, 每当实例化一个对象时, 该方法都将被调用。这种构造函数可以包含常规函数能包含的任何代码。构造函数通常用于完成对象的初始化设置, 可以包含诸如初始化变量等功能。

构造函数的格式如下:

```
modifiers classname( )
{
    // Constructor body
}
```

构造函数的名称与其所属的类相同, 其中限定符与用于其他方法的限定符相同。通常只使用 public, 不指定返回数据类型。

需要注意的是, 每个类都有一个默认的构造函数, 即使您没有创建任何构造函数。通过创建构造函数, 可以控制某些设置。

每当您创建一个对象时, 相应类的构造函数都将被调用。程序清单 7.8 演示了如何使用构造函数。

程序清单 7.8 Constr.cs: 使用构造函数

```
1: // constr.cs - constructors
2: //-----
3:
4: using System;
5:
6: public class myClass
7: {
8:     static public int sctr = 0;
9:     public int ctr = 0;
10:
11:     public void routine()
12:     {
13:         Console.WriteLine("In the routine - ctr = {0} / sctr = {1}\n",
14:                             ctr, sctr );
15:     }
16:
17:     public myClass()
18:     {
```

```
19:         ctr++;
20:         sctr++;
21:         Console.WriteLine("In Constructor- ctr = {0} / sctr = {1}\n",
22:                             ctr, sctr );
23:     }
24: }
25:
26: class TestApp
27: {
28:     public static void Main()
29:     {
30:         Console.WriteLine("Start of Main method...");
31:
32:         Console.WriteLine("Creating first object...");
33:         myClass first = new myClass();
34:         Console.WriteLine("Creating second object...");
35:         myClass second = new myClass();
36:
37:         Console.WriteLine("Calling first routine...");
38:         first.routine();
39:
40:         Console.WriteLine("Creating third object...");
41:         myClass third = new myClass();
42:         Console.WriteLine("Calling third routine...");
43:         third.routine();
44:
45:         Console.WriteLine("Calling second routine...");
46:         second.routine();
47:
48:         Console.WriteLine("End of Main method");
49:     }
50: }
```

该程序清单的输出如下所示:

```
Start of Main method...
Creating first object...
In Constructor- ctr = 1 / sctr = 1

Creating second object...
In Constructor- ctr = 1 / sctr = 2

Calling first routine...
In the routine - ctr = 1 / sctr = 2

Creating third object...
In Constructor- ctr = 1 / sctr = 3

Calling third routine...
In the routine - ctr = 1 / sctr = 3
```

```

Calling second routine...
In the routine - ctr = 1 / sctr = 3

End of Main method

```

分析：该程序清单演示一个非常简单的构造函数，如第17~23行所示，它还再次说明了静态数据成员和常规数据成员之间的差别。第6~24行定义了一个名为myClass的类，这个类包含两个被用作计数器的数据成员，第一个数据成员被声明为静态的，名称为sctr；第二个不是静态的，名为ctr（不带s）。

注意：类将创建一个在所有对象之间共享的静态数据成员拷贝，而对于常规数据成员，每个对象都有自己的一个拷贝。

myClass 类包含两个方法。第一个叫 routine，它打印一行文本，其中包含两个计数器的当前值；第二个方法的名称与类名相同，即为 myClass，通过这一点，您知道它是一个构造函数。每当对象被创建时，该方法都将被调用。该方法完成了几项工作。首先将每个计数器的值加 1。ctr 是首次被加 1，因为它是特定对象的 ctr 拷贝。sctr 则不同，因为它是该类的所有对象共享的。因此，每创建该类的一个对象，sctr 的值都将被加 1。最后，第 21 行打印一条消息，显示 ctr 和 sctr 的值。

从第 26 行开始是该程序的应用程序类，它打印消息并实例化 myClass 对象，仅此而已。查看输出中的消息，您将发现，这与程序清单是相符的。需要注意的有趣的一点是，在第 46 行调用第二个对象的 routine 方法时，sctr 的值为 3，而不是 2。这是因为 sctr 是在所有的对象之间共享的，当您打印该消息时，您已经调用了构造函数三次。

最后需要注意的是，不要在程序清单中调用构造函数，请看第 33 行：

```
33:      myClass first = new myClass();
```

这行代码创建一个对象。虽然构造函数是自动调用的，但请注意这行中的 myClass 调用！

注意：明天的课程将介绍如何给构造函数传递参数。

7.6.2.2 静态构造函数

与数据成员和方法一样，也可以创建静态构造函数。使用限定符 static 声明的构造函数将在第一个对象被创建之前被调用，它只被调用一次，以后便不再被使用。程序清单 7.9 对程序清单 7.8 做了一些修改。在该程序清单中，在 test 类中添加了一个静态构造函数。

注意，这个构造函数同其他构造函数的名字相同。因为静态构造函数的名字包括 Static，编译器会将其与普通的构造函数区分开。

程序清单 7.9 statcon.cs: 使用静态构造函数

```

1: // statcon.cs - static constructors
2: //-----
3:
4: using System;
5:
6: public class test
7: {
8:     static public int sctr;
9:     public int ctr;

```



```
10:
11:     public void routine()
12:     {
13:         Console.WriteLine("In the routine - ctr = {0} / sctr = {1}\n",
14:             ctr, sctr );
15:     }
16:
17:     static test()
18:     {
19:         sctr = 100;
20:         Console.WriteLine("In Static Constructor - sctr = {0}\n", sctr );
21:     }
22:
23:     public test()
24:     {
25:         ctr++;
26:         sctr++;
27:         Console.WriteLine("In Constructor- ctr = {0} / sctr = {1}\n",
28:             ctr, sctr );
29:     }
30: }
31:
32: class TestApp
33: {
34:     public static void Main()
35:     {
36:         Console.WriteLine("Start of Main method...");
37:
38:         Console.WriteLine("Creating first object...");
39:         test first = new test();
40:         Console.WriteLine("Creating second object...");
41:         test second = new test();
42:
43:         Console.WriteLine("Calling first routine...");
44:         first.routine();
45:
46:         Console.WriteLine("Creating third object...");
47:         test third = new test();
48:         Console.WriteLine("Calling third routine...");
49:         third.routine();
50:
51:         Console.WriteLine("Calling second routine...");
52:         second.routine();
53:
54:         Console.WriteLine("End of Main method");
55:     }
56: }
```

该程序清单的输出如下：

```
Start of Main method...
Creating first object...
In Static Constructor - sctr = 100

In Constructor- ctr = 1 / sctr = 101

Creating second object...
In Constructor- ctr = 1 / sctr = 102

Calling first routine...
In the routine - ctr = 1 / sctr = 102

Creating third object...
In Constructor- ctr = 1 / sctr = 103

Calling third routine...
In the routine - ctr = 1 / sctr = 103

Calling second routine...
In the routine - ctr = 1 / sctr = 103

End of Main method
```

分析：该程序清单的输出和程序清单7.8没有什么本质的不同。输出中的第三行来自静态构造函数。该构造函数（第17~21行）将静态数据成员sctr初始化为100，然后打印一条消息。程序其他部分的运行方式与程序清单7.8完全相同。输出与程序清单7.8稍有不同的原因在于，变量sctr的初始值为100，而不是0。

7.6.3 析构函数

可以在对象被释放时执行某些操作，这是在析构函数中实现的。

当程序不再使用对象后的某一时刻，析构函数将自动执行。所谓“某一时刻”是不是有些不明确？这是故意的，析构操作可以在对象不再被使用到程序结束之前的任何时候进行。事实上，程序结束时也可能不调用析构函数，这意味着析构函数永远不会被调用。至于何时执行析构函数，您并没有真正的控制权，因此析构函数的价值有限。

警告：在诸如C++这样的语言中，可以调用析构函数，程序员能够控制何时执行析构函数。而在C#中，不是这样。

从技术的角度

析构函数通常是由 C#运行阶段环境在对象不再被使用时调用的。C#运行阶段环境通常是在检查是否有内存可供释放（被称为无用单元收集）前调用析构函数。如果在对象不再使用之后到程序结束之前，C#运行阶段环境没有执行这种内存检查，则析构函数将永远不会被调用。可以强行进行无用单元收集，但这还不如尽量少用析构函数。

7.6.3.1 使用析构函数

在C#中，析构函数是使用后跟类名和空括号的代字号(~)来定义的。例如，类xyz的析构函

数为:

```
~xyz( )
{
    // Destructor body
}
```

析构函数不使用限定符或其他关键字。程序清单 7.10 对 7.8 进行了简化, 并添加了一个析构函数。

程序清单 7.10: destr.cs: 使用析构函数

```
1: // destr.cs - constructors
2: //-----
3:
4: using System;
5:
6: public class test
7: {
8:     static public int sctr = 0;
9:     public int ctr = 0;
10:
11:     public void routine()
12:     {
13:         Console.WriteLine("In the routine - ctr = {0} / sctr = {1}",
14:                             ctr, sctr);
15:     }
16:
17:     public test()
18:     {
19:         ctr++;
20:         sctr++;
21:         Console.WriteLine("In Constructor");
22:     }
23:
24:     ~test()
25:     {
26:         Console.WriteLine("In Destructor");
27:     }
28: }
29:
30: class TestApp
31: {
32:     public static void Main()
33:     {
34:         Console.WriteLine("Start of Main method");
35:
36:         test first = new test();
37:         test second = new test();
38:
39:         first.routine();
```

```
40:
41:     test third = new test();
42:     third.routine();
43:
44:     second.routine();           // calling second routine last
45:
46:     Console.WriteLine("End of Main method");
47: }
48: }
```

该程序清单的输出如下:

```
Start of Main method
In Constructor
In Constructor
In the routine - ctr = 1 / sctr = 2
In Constructor
In the routine - ctr = 1 / sctr = 3
In the routine - ctr = 1 / sctr = 3
End of Main method
In Destructor
In Destructor
In Destructor
```

从输出可以知道, 在每个对象被释放后, 析构函数被调用。在这里, 是在 `Main()` 方法结束时; 但也有可能析构函数永远不被调用。

注意: 需要重申的是, 析构函数不一定被调用。当您执行程序清单 7.10 时, 便可能发现它们没有被执行。

7.6.3.2 析构函数和扫尾工作

由于内部发生的一些事情, 析构函数常被称为结束器 (finalizer)。析构函数与一个叫作 `Finalize` 的方法有关。为完成扫尾工作 (finalization), 编译器将把构造函数转换为正确的代码。

7.7 总 结

今天的课程介绍的主题不多, 但这些主题对于使用 C# 和面向对象语言编程举足轻重。昨天介绍了如何在类中添加数据成员, 而今天介绍了如何以方法的方式给类添加功能。您知道了方法、函数和例程虽然是不同的术语, 但指的是同一回事。

介绍方法的基本知识后, 重温了“按值”和“按引用”之间的差别, 介绍了如何以参数 (argument) 的方式将信息传递给方法头中指定的参量 (parameter)。通过使用诸如 `ref` 和 `out` 等关键字, 可以改变方法处理被传递的数据的方式。

最后, 介绍了一些特殊的方法, 包括构造函数和析构函数。

本周复习的后面是明天的课程。在第 8 天的课程中, 将扩充您今天学到关于方法的知识, 将介绍重载、代表以及方法的许多其他的高级特性。

7.8 问与答

问：实参（argument）和形参（parameter）之间的区别何在？

答：形参定义了应将什么传递给方法，它位于方法头的方法定义中。实参是传递给方法的值，传递给方法的实参必须与方法定义中指定的形参匹配。

问：可以在类外创建方法吗？

答：虽然在其他语言中，可以在类的外面创建方法，但在 C# 中不可以。C# 是面向对象的，因此所有的代码都必须位于类框架之内。

问：在 C# 中，方法和类的运行方式与在诸如 C++ 和 Java 等语言中相同吗？

答：在很大程度上是类似的，但随语言而异。详细介绍这种差别超出了本书的范围。其中的差别之一是，C# 不允许方法的参数有默认值。在诸如 C++ 等语言中，如果调用方法时没有给参数提供值，则该参数可以有默认的特定值。而 C# 则不是这样。当然还有其他差别。

问：如果不打算指望析构函数，如何编写扫尾代码？

答：建议创建自己的代码来完成扫尾工作，并在您知道不会再使用某个对象时，显式地调用这些代码。例如，如果有一个创建文件对象的类，则在使用完这种对象后，应该关闭文件。由于析构函数可能不会被调用，或者很长一段时间后才会被调用，因此应该创建自己的关闭文件的方法。您确实不希望使用完文件后，还让它处于打开状态。

7.9 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

7.9.1 小测验

1. 方法的两个重要组成部分是什么？
2. 函数和方法之间有何区别？
3. 方法可以返回多少个值？
4. 使用哪个关键字从方法返回一个值？
5. 方法可以返回哪些数据类型？
6. 访问类的成员方法使用什么样的格式？例如，如果使用 `myClass` 类实例化了一个名为 `myObject` 的对象，而该类包含一个名为 `myMethod` 的方法，下面哪种访问该方法的方式是正确的？
 - a. `myClass.myMethod`
 - b. `myObject.myMethod`
 - c. `myMethod`
 - d. `myClass.myObject.myMethod`
7. 按引用传递变量和按值传递变量之间的区别何在？
8. 构造函数在何时被调用？
9. 没有参数的析构函数的语法是什么样的？
10. 析构函数在何时被调用？

7.9.2 练习

1. 编写公有方法 xyz 的方法头，该方法接受两个参数，不返回任何值。
2. 为方法 myMethod 编写方法头。该方法接受三个参数。第一个名为 myVal，其数据类型为 double，并按值传递；第二个是一个输出变量，名为 myOutput；第三个是按引用传递的，数据类型为 int，名为 myReference。另外该方法是公有的，其返回类型为 byte。
3. 使用程序清单 7.1 中的 Circle 类，添加一个构造函数，将默认的圆心坐标设置为(5,5)，半径为 1。在程序中使用这个类，该程序打印缺省的值和您设置的值。不要在 Main() 方法中打印关于圆的信息，而是创建一个打印圆信息的方法。
4. 下面的代码片段有问题，哪一行将导致错误消息？

```
public void myMethod()
{
    System.Console.WriteLine("I'm a little teapot short and stout");
    System.Console.WriteLine("Down came the rain and washed the spider out");

    return 0;
}
```

5. 使用几天前介绍的骰子类，创建一个新的程序。该程序包含一个骰子类，后者有三个数据成员，即骰子的面数、骰子的点数以及包含随机数类（在前面的范例中，被定义为 rnd）的静态数据成员。为这个类声明一个名为 roll() 的成员方法，它以随机的方式返回下一次掷骰子得到的点数。

第1周复习

恭喜您！您完成了学习 C# 的第 1 周课程。这一周为编写 C# 应用程序打下了基础。您学习了如何存储基础数据、控制程序流程、重复执行代码片段、创建包含数据和方法的类以及许多其他的知识。

本周介绍的大部分清单和程序都是重点介绍某个概念。下面的代码将本周学习的各种概念综合在一起，正如您看到的，程序清单将更长一些。

执行该程序清单时，屏幕上将出现一个菜单，供您选择其中的选项。系统将根据您的选择来创建类并执行一些代码。

注意：该程序清单没有包含任何您还未学习过的东西。在接下来的两周中，您将学习一些对该程序清单进行改进的办法。这包括以更佳的方式实现某些功能、其他一些取得并转换来自用户的数据的方式，等等。

程序 WR01.cs

请输入、编译并执行程序清单 WR01.cs。该清单中包含 XML 注释，这意味着您可以通过包含第 2 天介绍的编译器开关/doc 来生成 XML 文档。

注意：虽然作者认为，学习的最佳方式是输入程序清单，并犯一些错误，但可以从 Sam 出版社的网站下载本书中各个程序清单的源代码。

程序清单 WR1.1 WR01.cs: 第 1 周复习

```
1: // File: wr01.cs
2: // Desc: Week One In Review
3: //      This program presents a menu and lets the user select a c
4: //      a choice. Based on this choice, the program then executes
5: //      a set of code that either manipulates a shape or exits the
6: //      program
7: //-----
8:
9: using System;
10:
11: class WR01App
```

```
12: {
13:     /// <summary>
14:     /// Main() routine that starts the application
15:     /// </summary>
16:     public static void Main()
17:     {
18:         int menuChoice = 99;
19:
20:         do
21:         {
22:             menuChoice = GetMenuChoice();
23:
24:             switch( menuChoice )
25:             {
26:                 case 0: break;
27:                 case 1: WorkWithLine();
28:                     break;
29:                 case 2: WorkWithCircle();
30:                     break;
31:                 case 3: WorkWithSquare();
32:                     break;
33:                 case 4: WorkWithTriangle();
34:                     break;
35:                 default: Console.WriteLine("\n\nError... Invalid menu option.");
36:                     break;
37:             }
38:
39:             if ( menuChoice != 0 )
40:             {
41:                 Console.Write("\nPress <ENTER> to continue...");
42:                 Console.ReadLine();
43:             }
44:
45:         } while ( menuChoice != 0 );
46:     }
47:
48:     /// <summary>
49:     /// Displays a menu of choices.
50:     /// </summary>
51:     static void DisplayMenu()
52:     {
53:         Console.WriteLine("\n          Menu");
54:         Console.WriteLine("===== \n");
55:         Console.WriteLine(" A - Working with Lines");
56:         Console.WriteLine(" B - Working with Circles");
57:         Console.WriteLine(" C - Working with Squares");
58:         Console.WriteLine(" D - Working with Triangles");
59:         Console.WriteLine(" Q - Quit\n");
60:         Console.WriteLine("===== \n");
61:     }
```



```

62:
63:  /// <summary>
64:  /// Gets a choice from the user and verifies that it is valid.
65:  /// Returns a numeric value to indicate which selection was made.
66:  /// </summary>
67:  static int GetMenuChoice()
68:  {
69:      int option = 0;
70:      bool cont = true;
71:      string buf;
72:
73:      while( cont == true )
74:      {
75:          DisplayMenu();
76:          Console.Write(" Enter Choice: ");
77:          buf = Console.ReadLine();
78:
79:          switch( buf )
80:          {
81:              case "a":
82:              case "A": option = 1;
83:                  cont = false;
84:                  break;
85:              case "b":
86:              case "B": option = 2;
87:                  cont = false;
88:                  break;
89:              case "c":
90:              case "C": option = 3;
91:                  cont = false;
92:                  break;
93:              case "d":
94:              case "D": option = 4;
95:                  cont = false;
96:                  break;
97:              case "q":
98:              case "Q": option = 0;
99:                  cont = false;
100:                  break;
101:
102:              default: Console.WriteLine("\n\n--> {0} is not valid <-- \n\n", buf);
103:                  break;
104:          }
105:      }
106:      return option;
107:  }
108:
109:  /// <summary>
110:  /// Method to perform code for Working with Line.
111:  /// </summary>

```

```
112: static void WorkWithLine()
113: {
114:     line myLine = new line();
115:
116:     myLine.start.x = 0;
117:     myLine.start.y = 0,
118:     myLine.end.x = 3;
119:     myLine.end.y = 3;
120:
121:     myLine.DisplayInfo();
122: }
123:
124: /// <summary>
125: /// Method to perform code for Working with Circles.
126: /// </summary>
127: static void WorkWithCircle()
128: {
129:     circle myCircle = new circle();
130:
131:     myCircle.center.x = 1;
132:     myCircle.center.y = 1;
133:     myCircle.radius = 10;
134:
135:     myCircle.DisplayInfo();
136: }
137:
138: /// <summary>
139: /// Method to perform code for Working with Squares.
140: /// </summary>
141: static void WorkWithSquare()
142: {
143:     square mySquare = new square();
144:
145:     mySquare.width.start.x = 1;
146:     mySquare.width.end.x = 10;
147:     mySquare.width.start.y = 0;
148:     mySquare.width.end.y = 0;
149:     mySquare.height.start.y = 2;
150:     mySquare.height.end.y = 8;
151:     mySquare.height.start.x = 0;
152:     mySquare.height.end.x = 0;
153:
154:     mySquare.DisplayInfo();
155: }
156:
157: /// <summary>
158: /// Method to perform code for Working with Triangles.
159: /// </summary>
160: static void WorkWithTriangle()
161: {
```

第1周复习

```
162:     Console.WriteLine("\n\nDo Triangle Stuff\n\n");
163:     // This section left for you to do
164: }
165: }
166:
167: //-----
168: /// <summary>
169: /// This is a point class. It is for storing and
170: /// working with an (x,y) value.
171: /// </summary>
172: class point
173: {
174:     private int point_x;
175:     private int point_y;
176:
177:     public int x {
178:         get { return point_x; }
179:         set { if ( value < 0 )
180:             point_x = 0;
181:             else
182:                 point_x = value; }
183:     }
184:     public int y {
185:         get { return point_y; }
186:         set { if ( value < 0 )
187:             point_y = 0;
188:             else
189:                 point_y = value; }
190:     }
191:
192:     public point()
193:     {
194:         x = 0;
195:         y = 0;
196:     }
197: }
198:
199: //-----
200: /// <summary>
201: /// This class encapsulates line functionality
202: /// <see>point</see>
203: /// </summary>
204: class line
205: {
206:     private point lineStart;
207:     private point lineEnd;
208:
209:     public point start {
210:         get { return lineStart; }
211:         set { if ( value.x < 0 )
```

```
212:         lineStart.x = 0;
213:     else
214:         lineStart.x = value.x;
215:     if ( value.y < 0 )
216:         lineStart.y = 0;
217:     else
218:         lineStart.y = value.y;
219:     }
220: }
221: public point end {
222:     get { return lineEnd; }
223:     set { if ( value.x < 0 )
224:         lineEnd.x = 0;
225:     else
226:         lineEnd.x = value.x;
227:     if ( value.y < 0 )
228:         lineEnd.y = 0;
229:     else
230:         lineEnd.y = value.y;
231:     }
232: }
233:
234: public double length()
235: {
236:     int x_diff;
237:     int y_diff;
238:     double length;
239:
240:     x_diff = end.x - start.x;
241:     y_diff = end.y - start.y;
242:
243:     length = (double) Math.Sqrt((x_diff * x_diff) + (y_diff * y_diff));
244:     return (length);
245: }
246:
247: public void DisplayInfo()
248: {
249:     Console.WriteLine("\n\n-----");
250:     Console.WriteLine("        Line stats:");
251:     Console.WriteLine("-----");
252:     Console.WriteLine(" Length:      {0:f3}", length());
253:     Console.WriteLine(" Start Point: ({0},{1})", start.x, start.y);
254:     Console.WriteLine(" End Point:   ({0},{1})", end.x, end.y);
255:     Console.WriteLine("-----\n");
256: }
257:
258: public line()
259: {
260:     lineStart = new point();
261:     lineEnd = new point();
```

第1周复习

```
262:     }
263: }
264:
265: //-----
266: /// <summary>
267: /// This class encapsulates square functionality
268: /// <see>line</see>
269: /// </summary>
270: class square
271: {
272:     private line squareHeight;
273:     private line squareWidth;
274:
275:     public line height {
276:         get { return squareHeight; }
277:         set { squareHeight.start.y = value.start.y;
278:             squareHeight.end.y = value.end.y;
279:         }
280:     }
281:     public line width {
282:         get { return squareWidth; }
283:         set { squareWidth.start.x = value.start.x;
284:             squareWidth.end.x = value.end.x;
285:         }
286:     }
287:
288:     public double area()
289:     {
290:         double total;
291:
292:         total = (width.length() * height.length());
293:         return (total);
294:     }
295:
296:     public double border()
297:     {
298:         double total;
299:
300:         total = ((2 * width.length()) + (2 * (height.length())));
301:         return (total);
302:     }
303:
304:     public void DisplayInfo()
305:     {
306:         Console.WriteLine("\n\n-----");
307:         Console.WriteLine("    Square stats:");
308:         Console.WriteLine("-----");
309:         Console.WriteLine(" Area:          {0:f3}", area());
310:         Console.WriteLine(" Border:        {0:f3}", border());
311:         Console.WriteLine(" WIDTH Points: ({0},{1}) to ({2},{3})",
```

```

312:             width.start.x, width.start.y, width.end.x, width.end.y);
313:         Console.WriteLine("        Length: {0:f3}", width.length());
314:         Console.WriteLine(" HEIGHT Points: ({0},{1}) to ({2},{3})",
315:             height.start.x, height.start.y, height.end.x,
height.end.y);
316:         Console.WriteLine("        Length: {0:f3}", height.length());
317:
318:         Console.WriteLine("-----\n");
319:     }
320:
321:     public square()
322:     {
323:         squareHeight = new line();
324:         squareWidth = new line();
325:
326:         width.start.x = 0;
327:         width.start.y = 0;
328:         width.end.x = 0;
329:         width.end.y = 0;
330:         height.start.x = 0;
331:         height.start.y = 0;
332:         height.end.x = 0;
333:         height.end.y = 0;
334:     }
335: }
336:
337: //-----
338: /// <summary>
339: /// This class encapsulates circle functionality
340: /// <see>line</see>
341: /// </summary>
342: class circle
343: {
344:     private point circleCenter;
345:     private long circleRadius;
346:
347:     public point center {
348:         get { return circleCenter; }
349:         set { circleCenter.x = value.x;
350:             circleCenter.y = value.y;
351:         }
352:     }
353:     public long radius {
354:         get { return circleRadius; }
355:         set { circleRadius = value; }
356:     }
357:
358:     public double area()
359:     {
360:         double total;

```

```

361:
362:     total = 3.14159 * radius * radius;
363:     return (total);
364: }
365:
366: public double circumference()
367: {
368:     double total;
369:
370:     total = 2 * 3.14159 * radius;
371:     return (total);
372: }
373:
374: public void DisplayInfo()
375: {
376:     Console.WriteLine("\n\n-----");
377:     Console.WriteLine("    Circle stats:");
378:     Console.WriteLine("-----");
379:     Console.WriteLine(" Area:           {0:f3}", area());
380:     Console.WriteLine(" Circumference: {0:f3}", circumference());
381:     Console.WriteLine(" Center Points: ({0},{1})", center.x, center.y);
382:     Console.WriteLine(" Radius:         {0:f3}", radius);
383:     Console.WriteLine("-----\n");
384: }
385:
386: public circle()
387: {
388:     circleCenter = new point();
389:
390:     center.x = 0;
391:     center.y = 0;
392:     radius = 0;
393: }
394: }
395: //----- End of Listing -----

```

运行该程序时，将出现如下所示的菜单：

```

Menu
=====
A - Working with Lines
B - Working with Circles
C - Working with Squares
D - Working with Triangles
Q - Quit
=====

```

Enter Choice:

如果您输入的不是上述菜单中指定的字母，则将出现下面的消息：

```

Menu
=====

A - Working with Lines
B - Working with Circles
C - Working with Squares
D - Working with Triangles
Q - Quit
=====

```

Enter Choice: g

g is not valid <--

然后再次显示菜单，选择有效的选项将得到与下面类似的结果（这是输入 c 得到的结果）：

```

Enter Choice: c
-----
Square Stats:
-----
Area:          54.0 00
Border:        3 0.00 0
WIDTH Points:  (1, 0) to (10, 0)

Length: 9.0 00
HEIGHT Points: (0, 2) to (0, 8)
Length: 6.000
-----

```

Press <ENTER> to continue...

XML 文档

正如前面指出的，使用该程序清单可以生成 XML 文档。下面是使用编译器选项/doc 生成的 XML 文档的内容：

```

< ?xml version= "1.0" ? >
< doc >
  < assembly >
    < name >wr01 < /name >
  < /assembly>
  < members >
    < member name= "M:WR01App.Main" >
      < summary>
        Main () routine that starts the application
      < /summary>
    < /member >
    < member name= "M:WR01App.DisplayMenu" >
      < summary >
        Displays a menu of choices.
      < /summary >

```



```
</member>
<member name="M: WROLApp.GetMenuChoice" >
    < summary >
        Gets a choice from the user and verifies that it is valid.
        Returns a numeric value to indicate which selection was made.
    < /summary>
< /member >
< member name= "M: WROLApp. WorkWithLine" >
    <summary>
        Method to perform code for Working with Line .
    < /summary>
< /member>
<member name= "M: WROLApp. WorkWithCircle" >
    < summary >
        Method to perform code for Working with Circles.
    < /summary>
< /member>
<member name= "M: WROLApp. WorkWithSquare" >
    < summary>
        Method to perform code for Working with Squares.
    < /summary>
< /member >
<member name= "M: WROLApp. WorkWithTriangle" >
    < summary >
        Method to perform code for Working with Triangles.
    < /summary>
< /member >
<member name= "T: point" >
    < summary>
        This is a point class. It is for storing and
        working with an (x, y) value.
    < /summary>
< / member >
<member name= "T: line" >
    < summary>
        This class encapsulates line functionality
        < see >point < /see >
    < / summary >
< /member >
<member name= "T: square" >
    < summary>
        This class encapsulates square functionality
        < see > line< /see >
    < /summary >
< /member >
<member name= "T: circle" >
    < summary >
        This class encapsulates circle functionality
        < see > line < /see >
    < /summary >
```

```
< /member >
    < /members >
< /doc>
```

代码概览

介绍程序清单的输出和 XML 文档后，我们来看一下代码。

该程序清单中有几项内容值得注意。首先，第 9 行将名称空间 `System` 包含到程序清单中。正如第 6 天介绍的，这意味着当您使用名称空间 `System` 内的东西时，不用指定 `System`；这些内容包括诸如 `Console` 方法等。该程序清单中声明了五个类：

- 第 11 ~ 65 行的 `WR01App` 类；
- 第 172 ~ 197 行的 `point` 类；
- 第 204 ~ 262 行的 `line` 类；
- 第 270 ~ 335 行的 `square` 类；
- 第 342 ~ 394 行的 `circle` 类。

注意：在下一周您将知道，对于声明 `Point` 而言，有一种比使用类更好的方法——使用关键字 `struct`。

`line`、`square` 和 `circle` 类相似 `Point` 类用于帮助组织其他类。

Main 方法

仔细查看该程序清单，您将知道该程序将从 `Main` 方法所在的第 16 行开始执行，该方法是在 `WR01App` 类中声明的。该方法使用一个 `do...while` 语句不断地处理菜单，直到用户选择合适的选项为止。菜单是通过调用另一个方法 `GetMenuChoice` 来显示的，然后根据该方法返回的值，执行多个不同例程中的一个。第 24 ~ 37 行的 `switch` 语句用于将程序流程转到正确的语句。

第 39 ~ 43 行使用一个 `if` 语句来检查用户选择的菜单选项。如果 `menuChoice` 为 0，则说明用户要退出。如果为其他的值，则在屏幕上显示信息。为在重新显示菜单之前，暂停程序的运行，加入了第 41 行和第 42 行。第 41 行向用户显示一条消息，指出按 `Enter` 键继续。第 42 行使用 `Console.ReadLine` 方法来等待用户按下 `Enter` 键。对于用户在按 `Enter` 键之前输入的文本，程序将忽略它们。用户按 `Enter` 键继续后，`while` 语句的条件将被检查。如果 `menuChoice` 不为 0，则执行 `do` 语句，从而再次显示菜单，重复前面的处理过程。

来看一下 `switch` 语句，其中前面的四个 `case` 都分别执行一个方法，这些访问 `WR01App` 类的后边。如果 `menuChoice` 不是 1 ~ 4 之间的值，则 `switch` 语句中的 `default`（第 35 行）被执行，即打印一条错误消息。

GetMenuChoice 方法

第 22 行调用了方法 `GetMenuChoice`。该方法位于第 67 ~ 107 行，它显示菜单，并取得用户的选择。第 75 行调用另一个方法来实际显示菜单。显示菜单后，第 77 行使用方法 `Console.ReadLine` 来取得用户的选择，并将其赋给字符串变量 `buf`。

然后使用一个 `switch` 语句将该选项转换为数字值，并将其返回给调用方法。并非一定要进行这种转换，该方法的目的是取得选择的菜单选项，对于每个菜单选项，都有两种方式可以选择它。`Switch` 语句接受其中的每种方式，并将其转换为一个正确的选项。完成这种工作的方式有很多

另外，您也可以返回字符值而不是数字值，这里需要指出的一点是，可以将获取被选择的菜单

选项的功能作为一个独立的方法，这样，您可以以任何方法来获知用户作出的选择，只要返回一组一致的最终选择值即可。您可以使用一个返回 0~4 的值的的方法来替换该方法，而其他代码的运行方式不变。

菜单选项

四个菜单选项中的每一个调用一个方法。第 112~122 行包含方法 `WorkWithLine`，该方法声明一个对象，设置其初始值，最后调用该对象的一个方法来显示关于该对象的信息。方法 `WorkWithSquare` 和 `WorkWithCircle` 的运行方式与此相同。`WorkWithTriangle` 方法没有包含代码，而是留给您来完成。

point 类

`point` 类是在第 172~197 行定义的，它包含两个私有的数据成员：`point_x` 和 `point_y`。由于这些数据成员是私有的，因此 `point` 类外的代码不会访问它们。第 177~190 行使用关键字 `get` 和 `set` 声明了属性，这些属性为用户取得 `point` 的数据提供了入口。

第 192 行为 `point` 类声明了构造函数，该构造函数将数据成员的值设置为 0，这里使用属性来初始化这些值。

如果想改变 `point` 值的存储方式，可以修改属性。

line 类

`line` 类与 `point` 类类似。第 206~207 行声明了数据成员，这里的数据成员为 `point` 对象。第 258~263 行的构造函数为 `line` 类实例化两个 `point` 对象。注意，在该构造函数中，没有设置默认值。不需要这样做，因为当每个 `point` 类被实例化时，其构造函数将设置默认值。

`line` 类还包含其他可被调用的方法，这些方法的代码简单易懂。

其他类

该程序中的其他类与 `line` 类类似，您可以自行查看它们的代码。

注意：您应该理解该程序清单中的所有代码，如果有什么地方不能理解，应回过头去复习相应的课程。以后您将学习如何对该程序清单进行改进。

第二周课程



您已学完了第 1 周的课程，还有两周的课程需要学习。本周将学习余下的大部分 C# 核心概念——不是全部，而是大部分。经过本周的学习后，您将从零开始掌握构建基本的 C# 程序所需的工具。

第 8 天课程“高级数据存储方式：结构、枚举和数组”将介绍其他存储数据的方式，包括结构、枚举和数组，其中结构类似于类。

第 9 天课程“关于方法的高级主题”将扩展第 7 和 8 天介绍的知识，介绍如何重载方法以及如何使用参数数目可变的方法。另外，还将介绍有关作用域方面的知识，让您能够限制对数据和其他类型成员的访问。您将学习关键字 `static` 以及如何创建不能用于创建对象的类。

创建程序时，避免让用户莫名其妙至关重要。第 10 天课程“处理异常”将介绍如何在程序中加入异常处理功能。异常处理技术是一种在问题导致程序崩溃之前捕获并防止它的结构化方式。

面向对象编程的关键之一是继承。第 11 天课程“继承”将介绍如何将继承用于您创建的类（或别人创建的类）。在这里，您将学习几个新的关键字，其中包括 `sealed`、`is` 和 `as`。

第 12 天课程“输入和输出”将复习一些稍微不那么重要的内容，让您稍事休息。这里将重点介绍如何在控制台上显示信息以及从控制台取得信息。您将学习如何格式化数据，使之更为有用。这里包含大量的表格，供您以后参考。

第 13 天课程“接口”介绍了理解 C# 强大功能的另一个核心主题，这里将扩展您已学习到的关于类、结构和继承方面的知识，您将学习如何使用接口将多个特性组合到同一个类中。

最后，第 14 天课程“索引器、代表和事件”将重点介绍三个有趣的主题——索引器、代表和事件。您将学习如何将索引表示法用于类中的数据。另外，还将学习有关代表和事件的知识，让您动态地执行方法以及进行事件编程。事件是编写 Windows 应用程序的关键。

阅读完本周的课程后，您将学到许多 C# 编程的核心概念。您将发现，您能理解大多数 C# 程序中的大部分核心概念——不是全部，而是大部分。

第 8 天课程

高级数据存储方式：结构、枚举和数组

您已经学习过基本数据类型和类。C#还提供了其他一些在程序中存储信息的方式，今天的课程将介绍其中的几种，包括结构、枚举和数组。具体地说，今天将介绍以下内容：

- 如何在结构中存储值；
- 结构与类的异同；
- 枚举是什么？如何使用它来使程序更容易理解？
- 如何声明数组并使用它来存储大量数据类型相同的值？
- 使用关键字 `foreach` 来操纵数组。

8.1 结 构

结构是 C#提供的一种数据类型，它与类类似。和类一样，结构也可以包含数据和方法定义，还可以包含构造函数、常量、字段、方法、属性、索引器（`indexer`）、运算符和嵌套类型。

8.1.1 结构与类

虽然结构和类之间有很多相似性，但它们之间有一主要的区别和一些细微的区别。结构和类的主要区别在于存储和访问方式，结构是一种值数据类型，而类是一种引用数据类型。

虽然值数据类型和引用数据类型之间的差别已经在本书前面介绍过，但有必要重申，以便您能完全理解这种差别。值数据类型存储的是数据类型的名称指定的位置处的值；而引用数据类型存储的是信息的存储位置。图 8.1 再次给出了第 7 天中的一个图，该图说明了值数据类型和引用数据类型之间的差别。

从图 8.1 可以知道，引用变量实际上比存储值变量更复杂，但这种复杂性是由编译器负责处理的。虽然按引用存储信息有其优点，但要占用更多的内存。如果存储的信息量较少，则按引用存储带来的利将小于额外占用内存带来的弊

结构是按值存储的，没有引用带来的内存开销，因此在处理少量数据时优于类。

处理大量数据时，诸如类等引用类型是一种更合适的存储方式，特别是在传递数据给方法时：对于引用变量值，只传递引用，而不是整个数据值，而对于诸如结构等值变量，将被复制并传递给

方法。这种复制将带来额外的开销，如果结构很大，将降低程序的运行速度。

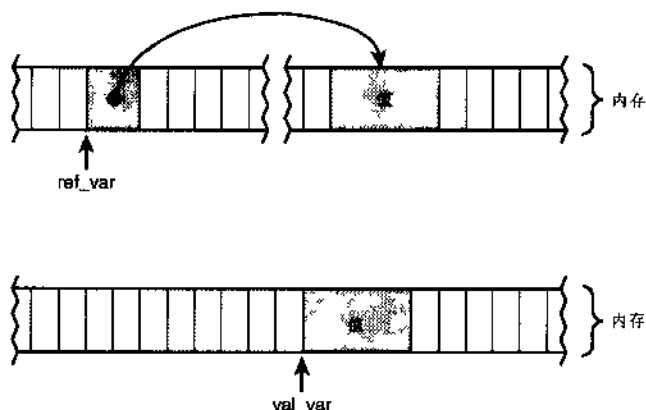


图8.1 按值存储和按引用存储

一个通用的经验规则是，在决定使用类还是结构时，如果数据成员占用的内存不超过 16 个字节，则使用结构；否则，则应考虑您将如何使用这些数据。

8.1.2 结构成员

声明结构成员的方法与类成员相同，下面是一个用于存储点的结构：

```
struct point
{
    public int x;
    public int y;
}
```

这与前几天介绍的类类似，唯一的差别在于使用的是关键字 `struct`，而不是 `class`。您也可以在程序清单中以使用类的方式使用结构，程序清单 8.1 使用了该 `point` 结构。

程序清单 8.1 point.cs: 使用 point 结构

```
1: // point.cs- A structure with two data members
2: //-----
3:
4: struct point
5: {
6:     public int x;
7:     public int y;
8: }
9:
10: class pointApp
11: {
12:     public static void Main()
13:     {
14:         point starting = new point();
15:         point ending = new point();
16:
17:         starting.x = 1;
```

```

18:     starting.y = 4;
19:     ending.x = 10;
20:     ending.y = 11;
21:
22:     System.Console.WriteLine("Point 1: ({0},{1})",
23:                               starting.x, starting.y);
24:     System.Console.WriteLine("Point 2: ({0},{1})",
25:                               ending.x, ending.y);
26: }
27: }

```

该程序清单的输出如下：

```

Point 1: (1,4)
Point 2: (10,11)

```

分析：使用结构和使用类之间的主要差别在于，定义结构时使用的是关键字 `struct`（如第4行所示）。事实上，您可以将关键字 `struct` 替换为 `class`，该程序清单仍将能够运行。正如前面指出的，结构和类的主要差别在于它们在内存中的存储方式。图8.2说明了在内存中，`point` 类对象 `starting` 和 `point` 结构实例是如何存储的。

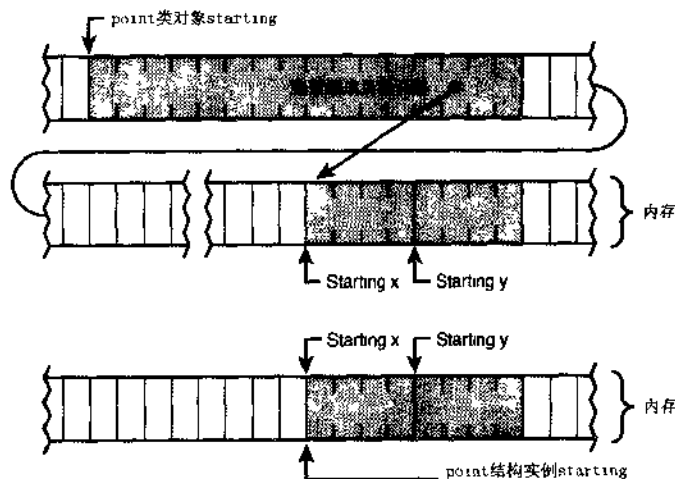


图8.2 将点作为结构和类存储在内存中的情况

另外，访问结构成员的方式与类成员相同，即使用由成员运算符（句点）连接的结构实例名称和数据成员名称。例如，第17行访问 `point` 结构实例 `starting` 的数据成员 `x`。

声明结构实例可以更简单，即可以不使用关键字 `new`，这意味着第14行和15行可以修改成如下所示：

```

14:     point starting;
15:     point ending;

```

做上述修改后，重新编译该程序清单，它将能够通过编译，并能运行。如果就此将关键字 `struct` 改为 `class`，则编译时将出现以下错误：

```

point.cs(17,7): error CS0165: Use of unassigned local variable 'starting'

```

```
point.cs(19,7): error CS0165: Use of unassigned local variable 'ending'
```

8.1.3 结构的嵌套

和类一样，结构中也可以包含其他数据类型，包括其他结构。程序清单 8.2 演示了一个包含两个 point 结构的 line 结构。

程序清单 8.2 line.cs: Line 结构

```
1: // line.cs- A line structure which contains point structures.
2: //-----
3:
4: struct point
5: {
6:     public int x;
7:     public int y;
8: }
9:
10: struct line
11: {
12:     public point starting;
13:     public point ending;
14: }
15:
16: class lineApp
17: {
18:     public static void Main()
19:     {
20:         line myLine;
21:
22:         myLine.starting.x = 1;
23:         myLine.starting.y = 4;
24:         myLine.ending.x = 10;
25:         myLine.ending.y = 11;
26:
27:         System.Console.WriteLine("Point 1: ({0},{1})",
28:                                   myLine.starting.x, myLine.starting.y);
29:         System.Console.WriteLine("Point 2: ({0},{1})",
30:                                   myLine.ending.x, myLine.ending.y);
31:     }
32: }
```

该程序清单的输出如下：

```
Point 1: (1,4)
Point 2: (10,11)
```

分析：line 结构的建立方式与前一天中的 line 类相似，主要差别在于实例化 line 结构时，将给它分配内存并直接存储它。

在该程序清单中，point 结构是在第 4~8 行声明的，第 12 行和 13 行声明了两个公有 point 结构

实例，其中的每个实例都有其坐标值 x 和 y 。

第 22~25 行设置了 `line` 结构的各个值，可以按结构层次来访问嵌套结构的值，使用成员运算符（句点）将结构层次连接起来。第 22 行访问的是 `line` 结构 `myLine` 的 `point` 结构 `starting` 的数据成员 x 。图 8.3 说明了 `line` 的层次。

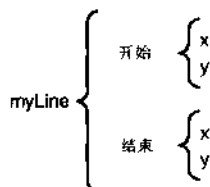


图 8.3 `line` 结构的层次

8.1.4 结构方法

和类一样，结构也可以包含方法和属性，它们的声明方式与类中相同，这包括使用与类中相同的限定符和属性（`attribute`）。您可以重载结构方法、传递值以及返回值。程序清单 8.3 演示了包含方法 `length` 的 `line` 结构。

程序清单 8.3 `Line2.cs`：在结构中添加方法

```
1: // Line2.cs- Adding methods to a structure.
2: //-----
3:
4: struct point
5: {
6:     public int x;
7:     public int y;
8: }
9:
10: struct line
11: {
12:     public point starting;
13:     public point ending;
14:
15:     public double length()
16:     {
17:         double len = 0;
18:         len = System.Math.Sqrt( (ending.x - starting.x)*(ending.x - starting.x) +
19:                                 (ending.y - starting.y)*(ending.y - starting.y));
20:         return len;
21:     }
22: }
23:
24: class lineApp
25: {
26:     public static void Main()
27:     {
28:         line myLine;
29:
```

```

30:     myLine.starting.x = 1;
31:     myLine.starting.y = 4;
32:     myLine.ending.x = 10;
33:     myLine.ending.y = 11;
34:
35:     System.Console.WriteLine("Point 1: ({0},{1})",
36:                               myLine.starting.x, myLine.starting.y);
37:     System.Console.WriteLine("Point 2: ({0},{1})",
38:                               myLine.ending.x, myLine.ending.y);
39:     System.Console.WriteLine("Length of line from Point 1 to Point 2: {0}",
40:                               myLine.length());
41: }
42: )

```

该程序清单的输出如下：

```

Point 1: (1,4)
Point 2: (10,11)
Length of line from Point 1 to Point 2: 11.4017542509914

```

分析：该程序清单中加入了本书前面介绍过的length方法。该方法是在line结构中声明的，如第15~21行所示。与以前相同，该方法使用line结构的数据成员来计算线段的长度。然后将结果赋给变量len，并将其作为double值返回，如第20行所示。

lineApp 类调用该方法，并使用 Console.WriteLine 方法将返回的值显示出来，如第39行和40行所示。

虽然 line 结构只有一个方法，但您可以为它创建多个方法和属性，也可以重载这些方法。

8.1.5 结构的构造函数

除了可以包含常规方法外，结构还可以有构造函数。与类不同的是，为结构声明构造函数时，必须包含参数，不能为结构声明一个不包含任何参数的构造函数。程序清单 8.4 中包含一个带构造函数的 point 结构。

程序清单 8.4 point2.cs: 包含构造函数的 point 结构

```

1: // point2.cs- A structure with two data members
2: //-----
3:
4: struct point
5: {
6:     public int x;
7:     public int y;
8:
9:     public point(int x, int y)
10:    {
11:        this.x = x;
12:        this.y = y;
13:    }
14: //    public point()
15: //    {

```

```

16: //      this.x = 0;
17: //      this.y = 0;
18: //  }
19: }
20:
21: class pointApp
22: {
23:     public static void Main()
24:     {
25:         point point1 = new point();
26:         point point2 = new point(8, 8);
27:
28:         point1.x = 1;
29:         point1.y = 4;
30:
31:         System.Console.WriteLine("Point 1: ({0},{1})",
32:                                   point1.x, point1.y);
33:         System.Console.WriteLine("Point 2: ({0},{1})",
34:                                   point2.x, point2.y);
35:     }
36: }

```

该程序清单 的输出如下：

```

Point 1: (1,4)
Point 2: (10,11)

```

分析：结构和类的差别是不能为结构声明不包含任何参数的构造函数。该程序清单中包含一个不带任何参数的构造函数，但被注释掉了。如果删除这些行中的单行注释标记，并重新编译该程序清单，则将出现如下所示的错误：

```
point2.cs(14,12): error CS0568: Structs cannot explicit parameterless constructors
```

可以声明带参数的构造函数，第 9~13 行声明了一个对点坐标进行初始化的构造函数。point 结构的 x 和 y 的值被设置为传递给构造函数的值。为区分传入的 x、y 和结构实例的变量 x、y，使用了关键字 this，如第 11 行和 12 行所示。

第 25 行使用常规的方法实例化结构 point，也可以使用下面的代码来实例化结构实例 point1：

```
point point1;
```

即不包含 new 关键字和不带参数的构造函数调用。第 26 行使用了带参数的构造函数。

结构的构造函数必须初始化结构的数据成员。调用结构的默认构造函数（不带参数）时，将自动使用默认值初始化每个数据成员。通常，数据成员被初始化为 0，如果调用的是您声明的构造函数，而不是默认的构造函数，则您必须在构造函数中初始化所有的数据成员。

警告：虽然使用默认构造函数时，可以省略关键字 new，但使用参数来实例化结构时，不能省略。将第 26 行修改为如下所示，将出错：

```
point point2(8,8);
```

8.1.6 结构的析构函数

类可以有析构函数。前面介绍过，虽然您可以使用类析构函数，但它们是不可靠的。结构不能有析构函数，如果您试图添加，将出现编译错误。

8.2 枚 举

在 C#中可以使用的另一种数据类型是枚举。枚举让您只能够创建有限的几个值的变量，例如每周只有七天，不将它们称为 1、2、3 等，而将其称为 `Day.Monday`、`Day.Tuesday` 和 `Day.Wednesday` 等，更清楚得多。对于可以是开或关的触发器，可以不使用诸如 0 和 1 等值，而是使用 `Toggle.On` 和 `Toggle.Off`。

枚举让您能够创建这样的值。枚举是使用关键字 `enum` 声明的，声明格式如下：

```
modifiers enum enumName
{
    enumMember1,
    enumMember2,
    ...
    enumMemberN
}
```

其中 `modifiers` 可以是关键字 `new` 或访问限定符（您熟悉的 `public`、`private` 或今天后面将介绍的 `protected` 和 `internal`）；`enumName` 是枚举的名称，可以是任何合法的标识符；`enumMember1`、`enumMember2` 到 `enumMemberN` 是枚举成员，包含描述性值。

下面声明了一个公有的 `toggle` 枚举：

```
public enum toggle
{
    On,
    Off
}
```

其中使用了关键字 `enum`，后面跟枚举的名称 `toggle`。该枚举有两个值：`On` 和 `Off`，它们之间由逗号分隔。要使用 `toggle` 枚举，可以声明一个数据类型为 `toggle` 的变量。例如下面声明了变量 `myToggle`：

```
toggle myToggle;
```

变量 `myToggle` 的合法取值有两个：`On` 和 `Off`。要使用这些值，可以使用枚举名和值名，并用成员运算符（句点）将它们连接起来。例如，`myToggle` 的值可以设置为 `toggle.On` 或 `toggle.Off`。使用 `switch` 语句可以检查这些不同的值。程序清单 8.5 演示了如何创建一个用于存储多个颜色值的枚举。

注意：默认情况下，在枚举变量被声明时，其值被设置为 0。

程序清单 8.5: colors.cs: 使用枚举

```
1: // color.cs- Using an enumeration
2: //           Note: Entering a nonnumeric number when running this
3: //           program will cause an exception to be thrown.
4: //-----
5:
```

```
6: using System;
7:
8: class MyApp
9: {
10:     enum Color
11:     {
12:         red,
13:         white,
14:         blue
15:     }
16:
17:     public static void Main()
18:     {
19:         string buffer;
20:         Color myColor;
21:
22:         Console.Write("Enter a value for a color: 0 = Red, 1 = White, 2 = Blue): ");
23:         buffer = Console.ReadLine();
24:
25:         myColor = (Color) Convert.ToInt32(buffer);
26:
27:         switch( myColor )
28:         {
29:             case Color.red:
30:                 System.Console.WriteLine("\nSwitched to Red...");
31:                 break;
32:             case Color.white:
33:                 System.Console.WriteLine("\nSwitched to White...");
34:                 break;
35:             case Color.blue:
36:                 System.Console.WriteLine("\nSwitched to Blue...");
37:                 break;
38:             default:
39:                 System.Console.WriteLine("\nSwitched to default...");
40:                 break;
41:         }
42:
43:         System.Console.WriteLine("\nColor is {0} ({1})", myColor, (int)myColor);
44:     }
45: }
```

该程序清单的输出如下：

Enter a value for a color: 0 = Red, 1 = White, 2 = Blue): 1

Switched to White...

Color is white (1)

Enter a value for a color: 0 = Red, 1 = White, 2 = Blue): 5

```
Switched to default...
```

```
Color is 5 (5)
```

分析：上述输出是运行两次该程序得到的。第一次运行时，输入的是1，它对应于white；第二次执行时，输入的是5，它不对应于任何颜色。

第10~15行声明了枚举Color，它包含3个成员：red、white和blue。当该枚举被创建时，第一个成员的值被指定为0；第二个成员的值被指定为1；第三个成员的值被指定为2。对于所有的枚举来说，第一个成员的值都是0，其余的成员依次递增。

第20行使用该枚举来创建一个名为myColor的变量，该变量可以用来存储Color枚举的值之一。第25行给该枚举变量赋值。这里有必要对所赋的值做一些说明。第22行在屏幕上提示用户输入一个值，第23行使用Console类的ReadLine方法来读取用户输入的值，由于用户可能输入任何值，因此程序中可能出错。第25行假设用户输入的值可以被转换为一个整数，它使用Convert类的ToInt32方法来转换变量buffer，buffer变量包含用户输入的值。然后将转换结果强行转换为Color类型，并将其赋给变量myColor。如果用于输入的不是数字，将出现运行错误，程序将终止。第10天将介绍一种妥善处理这种错误的方法，以便不显示运行错误，并继续执行程序。

第27行包含一个switch语句，根据myColor的值执行不同的分支。第29~35行的case语句中使用的不是字面值，而是枚举的值。因此myColor的值将与枚举的值进行匹配。该switch语句只是说明了如何根据枚举的不同值执行不同的分支而已。

有必要对第43行进行说明，该行打印两个值。第一个是myColor的值，您可能认为这是赋给变量的数字值，但实际上并非如此，而是枚举成员的名称。当myColor的值为1时，打印的是white，而不是1。如果要的是数字值，则必须显式地强制打印数字，在第43行通过强制转换实现了这一目的。

8.2.1 修改枚举的默认值

给枚举变量设置的默认值是0。虽然给枚举变量指定的默认值为0，但枚举中不一定非得包含一个值为0的成员。前面介绍过，枚举中第一个成员的值是0，然后依次递增。实际上，可以修改这些默认值，例如，您常常希望第一个成员的值是1，而不是0。

要创建第一个成员的值是1的枚举，方法有两种。首先，可以在枚举的开始位置加上一个填充符值。如果希望第一个成员的值是1，则这种方法很简单；但如果希望第一个成员的值是更大的数字，则这种方法便很糟糕。

第二种方法是设置枚举成员的值，可以将它们设置为字面值、其他枚举成员的值或计算得到的值。程序清单8.6并没有执行什么复杂的操作来设置枚举成员的值，而只是将第一个成员的值设置为1（而不是0）。第二个成员（February）的值为第一个成员的值加1，这意味着无论是否像清单中那样显式地将其设置为2，它的值都将是2。

程序清单 8.6 bday.cs：设置枚举成员的数值型值

```
1: // bday.cs- Using an enumeration, setting default values
2: //-----
3:
4: using System;
5:
```

```
6: public class MyApp
7: {
8:     enum Month
9:     {
10:         January = 1,
11:         February = 2,
12:         March = 3,
13:         April = 4,
14:         May = 5,
15:         June = 6,
16:         July = 7,
17:         August = 8,
18:         September = 9,
19:         October = 10,
20:         November = 11,
21:         December = 12
22:     }
23:
24:     struct birthday
25:     {
26:         public Month bmonth;
27:         public int bday;
28:         public int byear;
29:     }
30:
31:     public static void Main()
32:     {
33:         birthday MyBirthday;
34:
35:         MyBirthday.bmonth = Month.August;
36:         MyBirthday.bday = 11;
37:         MyBirthday.byear = 1981; // This is a lie...
38:
39:         System.Console.WriteLine("My birthday is {0}, {1} {2}",
40:             MyBirthday.bmonth, MyBirthday.bday, MyBirthday.byear);
41:     }
42: }
```

该程序清单的输出如下：

My birthday is August, 11 1981

分析：该程序清单创建了一个名为Month的枚举类型，它包含12个月。这里将成员的值设置为1~12，而不是使用默认值0~11。由于成员的值将在前一个值的基础上加1，因此可以不显式地设置第二个成员以及以后的各个成员的值。这里之所以这样做，是为了更加清晰。您可以将这些值设置为其他的值，甚至可以使用公式。例如，可以将June的值设置为下面的公式：

May + 1

由于 May 被看作是 5，因此 June 的值将被设置为 6。

第 26 行使用枚举类型 `Month` 为结构声明了一个名为 `bmonth` 的公有数据成员。第 33 行使用结构 `birthday` 声明了一个名为 `MyBirthday` 的变量。然后在第 35 ~ 37 行给该结构实例的数据成员分别赋值。数据成员 `bmonth` 被赋给值 `Month.August`，也可以使用下面的强制转换方式将 `bmonth` 的值设置为 `Month.August`，但这样做将没有那么清晰：

```
MyBirthday.bmonth = (Month) 8;
```

同样，从第 39 行的打印结果可以知道，存储在 `MyBirthday` 中的值为 `August`，而不是数字 8。

8.2.2 修改枚举的底层类型

到目前为止的所有范例中，枚举的底层数据类型都是 `int`，实际上枚举可以包含数据类型为 `byte`、`sbyte`、`int`、`uint`、`short`、`ushort`、`long` 和 `ulong` 的值。如果未指定数据类型，则默认类型为 `int`。如果要在枚举中存储比 `int` 更大或更小的值，则可以将默认的底层类型修改为其他的类型。

修改默认类型的格式如下：

```
modifiers enum enumName: typeName {member(s)}
```

上述定义与前面介绍的相同，只是加入了冒号和 `typeName`。其中 `typeName` 可以是前面指出的任何一种类型。修改底层类型后，必须确保给成员指定数字型值的类型是有效的。

程序清单 8.7 使用了前面的 `Color` 枚举，但由于这些值较小，因此这里将枚举设置为使用 `byte`，以节省一些内存。

程序清单 8.7 color2.cs: 显示随机的 `byte` 值

```
1: // color2.cs- Using enumerations
2: //-----
3:
4: using System;
5:
6: class MyApp
7: {
8:     enum Color : byte
9:     {
10:         red,
11:         white,
12:         blue
13:     }
14:
15:     public static void Main()
16:     {
17:         Color myColor;
18:         byte roll;
19:
20:         System.Random rnd = new System.Random();
21:
22:         for ( int ctr = 0; ctr < 10; ctr++ )
23:         {
24:             roll = (byte) ((rnd.NextDouble() * 3 )); // random nbr form 0 to 2
25:             myColor = (Color) roll;
26:         }
```



```

27:         System.Console.WriteLine("Color is {0} ({1} of type {2})",
28:             myColor, (byte) myColor, myColor.GetTypeCode());
29:     }
30: }
31: }

```

该程序清单的输出如下：

```

Color is white (1 of type Byte)
Color is white (1 of type Byte)
Color is red (0 of type Byte)
Color is white (1 of type Byte)
Color is blue (2 of type Byte)
Color is red (0 of type Byte)
Color is red (0 of type Byte)
Color is red (0 of type Byte)
Color is blue (2 of type Byte)
Color is red (0 of type Byte)

```

注意：由于使用了随机数生成器，因此您运行该程序得到的输出可能不同。

分析：程序清单不仅仅是使用 byte 来声明枚举，稍后您将明白这一点。首先来看一下第 8 行，这里使用 byte 而不是 int 值来创建枚举 Color，这是由于其中包含冒号和关键字 byte。这意味着 Color.red 的值将是 byte 值 0，Color.white 将是 byte 值 1，而 Color.blue 将是 byte 值 2。

该程序清单的 Main 方法的功能与前一个程序清单不同，它使用了以前介绍过的随机逻辑。第 24 行生成一个 0~2 的随机数，并显式地将其强制转换为 byte 值，然后赋给变量 roll。在 18 行，roll 变量的类型被声明为 byte。第 25 行显式地将 roll 的值转换为 Color 类型，然后赋给变量 myColor。

第 27 行以后与前一个程序清单类似，使用 WriteLine 方法打印变量 myColor 的值（结果为 red、white 或 blue），然后通过显式强制转换为 byte 来打印其数字型值。但打印的第三个值是新的。

和 C# 中的其他任何东西一样，枚举也是对象。因此，存在一些可用于枚举的内置方法。您将发现，其中最有用的一个是 GetTypeCode 方法，它返回变量的存储类型。对于 myColor，返回的类型为 byte，如输出中所示。如果在前两个清单中加入该参数，则结果将为 Int32。由于变量的类型是由运行阶段环境来确定的，因此返回的是 .NET 框架数据类型，而不是 C# 数据类型。

提示：有关用于枚举的其他方法，请参阅 .NET 框架文档中的 Enum 类。

应该	不应该
使用逗号，而不是分号来分隔枚举成员	填充符值不是枚举成员

8.3 使用数组存储数据

除了使用类和结构来将不同类型的相关信息存储在一起外，有时候您还需要存储多个数据类型相同的值。例如，银行可能需要跟踪每个月的结余，教师可能需要记录多次的考试成绩。

如果需要记录多个数据类型相同的条目，则最佳的方案是使用数组。要记录一年中每个月的结余，如果不使用数组，则需要创建 12 个变量来记录这些值：

```

decimal Jan_balance;
decimal Feb_balance;

```

```

decimal Mar_balance;
decimal Apr_balance;
decimal May_balance;
decimal Jun_balance;
decimal Jul_balance;
decimal Aug_balance;
decimal Sep_balance;
decimal Oct_balance;
decimal Nov_balance;
decimal Dec_balance;

```

要使用这些变量，首先要确定是哪个月，然后选择相应的变量。这需要多行代码，其中可能包含一条很长的 switch 语句，如下所示：

```

...
switch (month)
{
    case 1: // do January stuff
        Jan_balance += new_amount;
        break;
    case 2: // do February stuff
        Feb_balance += new_amount;
        break;
    ...

```

上面并没有列出整个 switch 语句，但足以说明确定月份并使用 12 个月结余值中的一个需要编写大量的代码。使用一个 decimal 数组来记录月结余的效率要高得多。

8.3.1 声明数组

数组是可以存储多个值的单个数据变量，其中所有值的数据类型都相同。在计算机内存中，这些数据元素是依次存储的，因此操纵它们以及在它们之间导航更为容易。

注意：在程序清单依次声明多个变量并不意味着它们将在内存中被存储在一起。事实上，即使是依次声明的变量也可能被存储在内存中完全不同的地方。数组是包含多个元素的单个变量，因此在内存中，数组的值被存储在一起。

要声明变量，可在声明该变量时在数据类型后面加上方括号。数组声明的基本格式如下：

```
datatype[] name;
```

其中，datatype 是将在数组中存储的数据类型；方括号指出您要声明一个数组；name 是该组变量的名称。下面的代码声明了一个名为 balances 的数组，用于存储 decimal 值：

```
decimal [] balances;
```

上述声明创建了一个名为 balances 的数组，可用来存储 decimal 值，但并没有为存储变量预留空间。要预留空间，需要像创建其他对象那样，使用 new 关键字来实例化该变量。实例化数组时，必须指出该数组将存储多少个值。为此，可以使用的方法之一是，在实例化数组时在方括号内指出元素数目：

```
balances = new decimal[12]
```

也可以在声明数组的同时进行实例化，对 `balances` 而言，代码如下：

```
decimal[] balances = new decimal[12];
```

从上可以知道，实例化的格式如下：

```
new datatype [nbr_of_elements]
```

其中，`datatype` 是数组的数组类型，`nbr_of_elements` 是一个数值型值，指出该数组中存储的元素数目。从数组 `balances` 的实例化代码可以知道，它可以存储 12 个 `decimal` 值。

新术语：声明并实例化数组后，便可以使用它。数组中的条目被称为元素，其中的每个元素都可以通过索引进行访问。索引是一个标识数组中的偏移量（从而标识元素）的数字。

数组的第一个元素用索引 0 标识，这是因为第一个元素位于数组的开始位置，因此偏移量为 0。第二个元素的索引为 1，因为其偏移量为 1 个元素。最后一个元素的偏移量为数组长度减 1。例如，数组 `balances` 有 12 个元素，因此最后一个元素的索引为 11。

要访问数组中某个特定的元素，可使用后面跟方括号内带相应索引的数组名称。例如，要将数组 `balances` 的第一个元素的值设置为 1297.50，可以使用下面的代码：

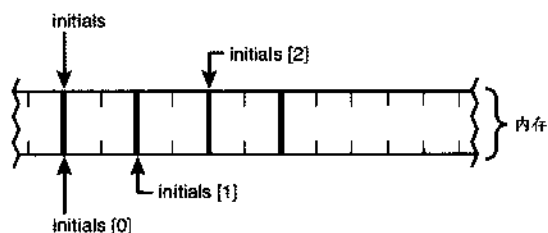
```
balances[0] = 1297.50m;
```

将一个 `decimal` 值赋给数组 `balances` 的第三个元素的代码如下：

```
balances[2] = 1000m;
```

索引 2 用于访问第三个元素。程序清单 8.8 演示了如何使用数组 `balances`，图 8.4 则说明了元素和索引的概念，图中使用了一个更为简单的、包含三个字符的数组，其声明如下：

```
char[] initials = new char[3];
```



```
char[] initials = new char[3];
```

图 8.4 存储在内存中的数组及其索引

警告：一种常见的错误是忘记索引是从 0 而不是 1 开始的。在诸如 Visual Basic 等语言中，索引可以从 1 开始，但在大多数语言（包括 C#）中，索引是从 0 开始的。

程序清单 8.8 balances.cs: 使用数组

```
1: // balances.cs - Using a basic array
2: //-----
3:
4: using System;
5:
6: public class myApp
```

```

7: {
8:     public static void Main()
9:     {
10:         decimal[] balances = new decimal[12];
11:
12:         decimal ttl = 0m;
13:         System.Random rnd = new System.Random();
14:
15:         // Put random values from 0 to 100000 into balances array
16:
17:         for (int indx = 0; indx < 12; indx++)
18:         {
19:             balances[indx] = (decimal) ((rnd.NextDouble() * 10000));
20:         }
21:
22:         //values are initialized in balances
23:
24:         for( int indx = 0; indx < 12; indx++)
25:         {
26:             Console.WriteLine("Balance {0}: {1}", indx, balances[indx]);
27:             ttl += balances[indx];
28:         }
29:
30:         Console.WriteLine("=====");
31:         Console.WriteLine("Total of Balances = {0}", ttl);
32:         Console.WriteLine("Average Balance = {0}", (ttl/12));
33:     }
34: }

```

该程序清单的输出如下:

```

Balance 0: 2276.50146106095
Balance 1: 4055.29556984794
Balance 2: 6192.0053633824
Balance 3: 2651.45477496621
Balance 4: 5885.39904257534
Balance 5: 2200.59107160223
Balance 6: 664.596651058922
Balance 7: 1079.63573237864
Balance 8: 2359.02580076783
Balance 9: 9690.85962031542
Balance 10: 934.673115114995
Balance 11: 7248.27192595614
=====
Total of Balances = 45238.310129027017
Average Balance = 3771.54250645135085

```

分析: 该程序清单演示了如何使用一个名为balances的基本数组。第10行将balances声明为一个decimal数组, 并被实例化为包含12个元素。第13行创建了一个Random对象, 用于创建存储在数组中

的随机数。将随机数赋给数组元素的操作是在第 17~20 行完成的，该 for 循环使用索引计数器 `indx` 从 0 循环到 11，并在第 19 行将该计数器用作数组的索引。

给数组元素赋值后，第 24~28 行遍历该数组。从技术上讲，该循环是多余的，但通常情况下，您会从其他地方获得数组元素的值，而不是将随机数指定给它们。第二个 for 循环将数组 `balances` 的每个元素显示到屏幕上（第 26 行）。第 27 行将数组 `balances` 的每个元素的值加到总数 `tot` 中。第 31 和 32 行则打印了一些关于这些随机结余的总结性信息。第 31 打印了结余总量，而第 32 行打印了平均结余。

与使用 12 个变量相比，使用 `balances` 数组的代码要简单得多。结合使用数组名和索引时，如 `balances[2]`，就像同类型的常规变量一样。

8.3.1.1 使用数组初始化数组元素

在声明和实例化数组的同时，可以初始化各个数组元素的值。为此，可以在变量声明后面指定数组元素的值。这些值包含在大括号内，之间用逗号分隔。要初始化数组 `balance` 的值，可以使用下面的代码：

```
decimal [] balances = new decimal[12] {1000.00m, 2000.00m, 3000.00m, 4000.00m, 5000m, 6000m,
0m, 0m, 9m, 0m, 0m, 12000m};
```

上述声明创建数组 `balances`，并预先给它赋值。第一个值 `1000.00m` 将赋给第一个元素 `balances[0]`；第二个值 `2000.00m` 将赋给第二个元素 `balances[1]`；依此类推。

有趣的是，以这种方式初始化数组的值，不需要在方括号内指定数组的大小。下面的语句与上述语句等效：

```
decimal [] balances = new decimal[] {1000.00m, 2000.00m, 3000.00m, 4000.00m, 5000m, 6000m,
0m, 0m, 9m, 0m, 0m, 12000m};
```

编译器自动将该数组定义为包含 12 个元素，因此初始化的是 12 个值。程序清单 8.9 创建并初始化了一个字符数组。

注意：如果在声明中指出了元素数目，则可以不初始化所有的元素。下面的代码行是合法的，数组将包含 12 个元素，开始两个元素的值被初始化为 111：

```
decimal[] balances = new decimal[12] {111m, 111m};
```

但如果没有指出元素数目，则以后不能增加元素。在下面的声明中，`balances` 数组只有两个元素，因此不能存储两个以上的元素。

```
decimal[] balances = new decimal [] {111m, 111m};
```

程序清单 8.9 fname.cs：使用数组

```
1: // fname.cs - Initializing an array
2: //-----
3:
4: using System;
5:
6: public class MyApp
7: {
8:     public static void Main()
9:     {
10:         char [] name = new char[] { 'B', 'r', 'a', 'd', 'l', 'e', 'y', (char) 0 };
11:
```

```

12:     Console.WriteLine("Display content of name array...");
13:
14:     int ctr = 0;
15:     while (name[ctr] != 0)
16:     {
17:         Console.Write("{0}", name[ctr]);
18:         ctr++;
19:     }
20:     Console.WriteLine("\n...Done.");
21: }
22: }

```

该程序清单的输出如下：

```

Display content of name array...
Bradley
...Done.

```

分析：第10行创建、实例化并初始化了一个名为name的字符数组，该数组被实例化为包含8个元素。您知道它将包含8个元素，因为虽然没有专门指出，但在声明时将8个元素加入了该数组中

该程序清单还包含一些以前没有介绍过的功能，它将一个奇怪的值（ASCII 码为0的字符）赋给数组的最后一个元素，这个值用于指示数组的结尾。第14~19行创建了一个名为ctr的计数器，并将其用作索引。ctr用于遍历数组的元素，直到找到一个ASCII 码为0的字符。至此，while 语句的条件为 false，循环结束。这样可以防止您超越数组的结尾处。

8.3.2 多维数组

多维数组是一个由数组组成的数组，您甚至可以使用三维数组！这将使数组急剧地复杂化，因此建议您不要使用超过三维的数组。

由数组组成的数组通常被称为二维数组，因为它可以以平面的方式进行表示。要声明二维数组，只需对声明常规数组的方法进行扩展即可：

```
byte [,] scores = new byte[15, 30];
```

声明的第一部分增加了一个逗号，而第二部分则使用了由逗号分隔的两个数字。上述声明创建了一个包含15个元素的数组，其中的每个元素都是一个包含30个元素的数组。因此，数组scores总共可以存储450个数据类型为byte的值。

要声明一个存储几个字符的简单多维数组，可以使用下面的代码：

```
char[,] letters = new char[2,3]; //without initializing values
```

该声明创建一个名为letters的二维数组，它包含两个元素，其中每个元素都是一个包含三个字符元素的数组。可以在声明时初始化数组letters的元素：

```
char[,] letters = new char[2,3] {{ 'a', 'b', 'c' }, { 'X', 'Y', 'Z' }};
```

也可以分别初始化各个元素。要访问多维数组的元素，同样也是使用索引。数组letters的第一个元素是letters[0,0]（记住索引从0开始，而不是1开始）；第二个元素是letters[0,1]，其值为字母b。字母X是元素letters[1,0]的值，因为位于第二个数组（偏移量为1）的第一个元素（偏移量为0）中。

要在声明之外初始化数组 `letters`，可以使用下面的代码：

```
letters[0,0] = 'a';
letters[0,1] = 'b';
letters[0,2] = 'c';
letters[1,0] = 'X';
letters[1,1] = 'Y';
letters[1,2] = 'Z';
```

8.3.3 创建包含的数组长度不同的数组

前一节假设二维数组中所有子数组的长度都相同，这样数组将呈矩形。如果要存储长度不同的数组，该如何办呢？请看下面的例子：

```
char [] [] myname = new char[3] [];
myname[0] = new char[] { 'B', 'r', 'a', 'd', 'l', 'e', 'y' };
myname[1] = new char[] { 'L', '.' };
myname[2] = new char[] { 'J', 'o', 'h', 'n', 'e', 's' };
```

数组 `myname` 是一个由数组组成的数组，包含三个长度各不相同的字符数组。由于它们的长度不同，因此使用它们的方式不同于前面的矩形数组。图 8.5 说明了数组 `myname`。

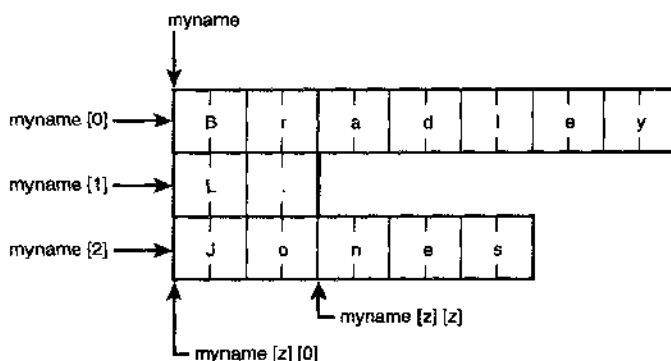


图 8.5 由长度各不相同的数组组成的数组

引用各个元素时，不是将用逗号分隔的索引放在同一个方括号内，而是将索引值放在不同的方括号内。例如，下面的代码行使用 `WriteLine` 打印作者姓和名的头一个字母：

```
System.Console.WriteLine("{0}{1}{2}", myname[0][0], myname[1][0], myname[2][0]);
```

以 `myname[0,0]`、`myname[1,0]` 和 `myname[2,0]` 的方式引用上述元素将出错。

注意：由长度相同的子数组组成的数组被称为矩形数组；由长度不同的子数组组成的数组被称为锯齿状数组（jagged）。从图 8.5 便可以明白为何被称为锯齿状数组。

如果您不想像前面那样在声明数组 `myname` 的同时初始化它，情况将如何呢？您知道该数组有三部分组成，因此第一维的长度为 3；但第二维的长度是多少呢？由于子数组的长度各不相同，因此必须执行多次实例化来创建出整个数组。首先，您声明包含子数组的数组：

```
char[] [] myname = new char[3] [];
```

上述代码声明了一个名为 `myname` 的、包含三个元素的数组，其中的每个元素都是存储字符的数组。完成上述声明后，必须实例化存储在 `myname[]` 中的各个数组（图 8.5 说明了 `myname` 数组）：

```

myname[0] = new char[7]; //first array of seven elements
myname[1] = new char[2]; //second array of two elements
myname[2] = new char[5]; //third array of five elements

```

8.3.4 检查数组长度和边界

在演示锯齿形多维数组 `myname` (如程序清单 8.10 所示)之前,有必要对另一项内容进行介绍。每个数组都知道其长度,数组的长度被存储在一个名为 `Length` 的成员中。和 C# 中的其他所有东西一样,数组也是对象。要知道数组的长度,可以像其他数据成员和对象那样使用数据成员 `Length`。对于一维数组 `balance`,可以通过 `balance.Length` 获知其长度。

在多维数组中,可以使用 `Length` 或数组的方法 `GetLength()` 来获知其子数组的长度。您通过传递了数组的索引号来指出要返回哪个子数组的长度。程序清单 8.10 演示了如何使用锯齿形数组的成员 `Length`。

程序清单 8.10 `names.cs`: 使用锯齿形的二维数组

```

1: // names.cs - Using a two-dimensional array
2: //-----
3:
4: using System;
5:
6: public class MyApp
7: {
8:     public static void Main()
9:     {
10:         char[,] name = new char[3][];
11:
12:         name[0] = new char[7] { 'B', 'r', 'a', 'd', 'l', 'e', 'y' };
13:         name[1] = new char[2] { 'L', '.' };
14:         name[2] = new char[5] { 'J', 'o', 'h', 'n', 's' };
15:
16:         Console.WriteLine("Display the sizes of the arrays...\n");
17:
18:         Console.WriteLine("Length of name array {0}", name.Length);
19:
20:         for( int ctr = 0; ctr < name.Length; ctr++)
21:             Console.WriteLine("Length of name[{0}] is {1}", ctr, name[ctr].Length);
22:
23:         //-----
24:
25:         Console.WriteLine("\n\nDisplaying the content of the name array...");
26:
27:         for( int ctr = 0; ctr < name.Length; ctr++)
28:         {
29:             Console.Write("\n"); // new line
30:             for( int ctr2 = 0; ctr2 < name[ctr].Length; ctr2++ )
31:             {
32:                 Console.Write("{0}", name[ctr][ctr2]);
33:             }

```



```

34:     }
35:     Console.WriteLine("\n...Done displaying");
36: }
37: }

```

该程序清单的输出如下：

```
Display the sizes of the arrays...
```

```

Length of name array 3
Length of name[0] is 7
Length of name[1] is 2
Length of name[2] is 5

```

```
Displaying the content of the name array...
```

```

Bradley
L.
Jones
...Done displaying

```

分析：来看看该程序清单的各个部分。第一部分是第10~14行。第10行声明了一个名为name的二维数组，它包含三个长度可能不同的字符数组。第12~14行分别实例化这三个数组，虽然其中的方括号中包含了数组的长度，但由于数组被初始化，因此可以不包含这些数字。但好的习惯是包含这些数字，并明确您的意图。

第二部分演示了数组成员 Length。第18行打印数组 name 的长度，您可能认为结果为14，但实际上为3。成员 Length 指的是元素数目，name 数组有三个元素，其中的每个元素都是一个数组。

第20行将数组 name 的长度（这里为3）作为上限，来遍历其中的每个数组。使用一个索引计数器打印每个数组的长度。从输出可以知道，这些值与声明的相同。

第三部分是第27~34行，它显示存储在各个 name 数组中的值。通过自动检查每个子数组的 Length 成员，而不是使用硬编码值，使得代码是动态的。如果在第12~14行修改数组的长度，则程序清单的其他部分仍管用。

8.3.5 在类和结构中使用数组

数组只不过是另一种可用于声明变量的类型而已。在可以使用其他数据类型的地方也可以创建和使用数组。这意味着可以在结构、类以及其他数据类型中使用数组。

注意：虽然今天使用的都是基本数据类型，但您几乎可以创建任何数据类型的数组。您可以使用类、结构或其他任何数据类型来创建数组。

8.3.6 使用 foreach 语句

现在是介绍关键字 foreach 的时候了。foreach 关键字使得使用数组更为简单，特别是遍历整个数组时。另外，它不是使用带下标的数组名来引用数组，而是使用一个简单的变量。foreach 语句的缺点在于，使用的简单变量是只读的——不能给它赋值。foreach 命令的格式如下：

```

foreach( datatype varname in arrayName)
{
    statements;
}

```

```

}

```

其中 `datatype` 是数组的数据类型, `varname` 是用于标识数组中各个元素的变量的名称; `arrayname` 是 `foreach` 要遍历的数组的名称。程序清单 8.11 演示了如何使用 `foreach` 来遍历一个名为 `name` 的数组。

程序清单 8.11 foreach1.cs: 将 foreach 用于数组

```

1: // foreach1.cs - Initializing an array
2: //-----
3:
4: using System;
5:
6: public class MyApp
7: {
8:     public static void Main()
9:     {
10:         char[] name = new char[] { 'B', 'r', 'a', 'd', 'l', 'e', 'y' };
11:
12:         Console.WriteLine("Display content of name array...");
13:
14:         foreach( char x in name )
15:         {
16:             Console.Write("{0}", x);
17:         }
18:
19:         Console.WriteLine("\n...Done.");
20:     }
21: }

```

该程序清单的输出如下:

```

Display content of name array...
Bradley
...Done

```

分析: 该程序清单比前一个短, 其重点在第14行, 它使用关键字 `foreach` 遍历数组 `name`。它遍历数组的每一个元素, 然后结束。遍历时, 它依次将各个元素称为 `x`。在 `foreach` 语句的代码中, 不需要使用 `array[index_ctr]`, 而是使用 `x`。

8.4 总 结

今天的课程介绍了三种重要的数据类型: 结构、枚举和数组。结构的工作原理与类类似, 它们之间的重要差别在于, 结构是值类型, 而类是引用类型。枚举 (使用关键字 `enum` 声明) 对提供代码的可读性很有帮助。枚举让您能够创建一种取值范围由您控制的数据类型, 同时可以给这些值指定更适用的名称。

今天讨论的最后一个主题是数组。您知道如何创建数组, 还知道数组可以是多维的。多维数组的子数组的长度可以相同 (矩形数组), 也可以不同 (锯齿形数组)。

最后介绍了 `foreach` 关键字, 该关键字使得使用数组简单得多。

8.5 问与答

问：在结构和类之间，还有今天没有介绍过的其他差别吗？

答：有，还有几个今天没有介绍的其他差别。您知道，结构是按值存储的，而类是按引用存储的。另外，结构的构造函数必须包含参数，而且不能有析构函数。除此之外，结构是隐式密封的（sealed），这一概念将在讨论继承时介绍。

问：据说枚举可以和位字段一起使用，这是如何实现的？

答：这是一个比较高级的主题，本书不打算介绍。可以使用枚举来存储各位的值。可以通过这样来实现，使用 byte 成员，每个成员被设置为字节中某位对应的位置。相应的枚举如下：

```
enum Bits : byte
{
    first = 1,
    second = 2,
    third = 4,
    fourth = 8,
    fifth = 16,
    sixth = 32,
    seventh = 64,
    eighth = 128
}
```

然后使用这些预定义的值，通过位运算符来执行位运算。

问：枚举是值类型还是引用类型？

答：当变量被声明为枚举时，它将是值类型。值实际上被存储在枚举变量中。

问：可以将数组声明为多少维？

答：可以将数组的维数声明为比实际需要的多些。如果声明三维以上的数组，将出现两种问题之一。要么由于使用矩形数组而浪费大量的内存空间，要么代码更为复杂。几乎在所有的情况下，都可以找到更为简单的处理信息的方法，这些方法不需要三维以上的数组。

8.6 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

8.6.1 小测验

1. 值数据类型和引用数据类型的差别何在？结构属于哪一种？
2. 结构和类的区别何在？
3. 结构的构造函数和类的构造函数之间有何差别（或者它们之间有差别吗）？
4. 哪个关键字用于声明枚举？
5. 在枚举中可以存储哪些数据类型？
6. 数组第一个元素的索引值是多少？
7. 如果访问数组元素时使用的索引值大于数组的元素数目，将出现什么情况？

8. 被声明为 `myArray[4,3,2]` 的数组包含多少个元素, 如果这是一个字符数组, 将占用多少内存?
9. 如何知道数组的长度?
10. 判断正误 (如果错误, 请指出错在哪里): `foreach` 的结构与 `for` 语句相同。

8.6.2 练习

1. 修改程序清单 8.3 中的结构 `point` 和 `line`, 给其数据成员加上属性。
2. 选做题: 修改 `line` 结构, 使之包含一个静态数据值, 它保存曾经存储过的最长的线段的长度。

每当 `Length` 方法被调用时, 都需检查并更新该静态变量。

3. 下面的代码片段有问题, 您能够更正吗 (假设 `myArray` 是一个 `decimal` 数组)?

```
foreach( decimal Element in myArray )
{
    System.Console.WriteLine("Element value is: {0}", Element);
    Element *= Element;
    System.Console.WriteLine("Element squared is: {0}", Element);
}
```

4. 为教师编写一个程序, 该程序使用一个数组存储 30 个学生的考试成绩, 并给各个数组元素指定一个 1 ~ 100 的随机值, 然后计算平均成绩。
5. 修改练习 4 中创建的程序, 以记录整个学期中 15 次考试的考试成绩, 当然 30 个学生的考试成绩都要记录。并打印各次考试的平均成绩和各个学生的学期平均成绩。
6. 修改练习 5 创建的程序, 以记录 5 年的全部考试成绩, 使用一个三维数组来实现。



第9天课程

关于方法的高级主题

前8天介绍了大量的内容，今天将以这些知识为基础进一步介绍类方法。第6天和第7天介绍了如何将功能和数据封装到类中，今天将介绍的重要知识之一是如何提高类的灵活性。今天的课程包含以下内容：

- 如何给方法传递不同的参数；
- 如何重载方法；
- 方法的特征标（signature）；
- 再谈作用域；
- 创建自己的名称空间。

9.1 重载方法

多态是面向编程语言的特性之一。正如本书前面介绍的，多态指的是即使提供不同的选项，也能得到相同的结果。重载是多态的形式之一，前面在介绍多态时，使用了一个圆的例子。

在C#中，最常使用重载的是方法。可以创建出对许多不同的值做出反应，并得到相同结果的方法。请看图9.1，图中的黑盒子是计算圆面积的方法。

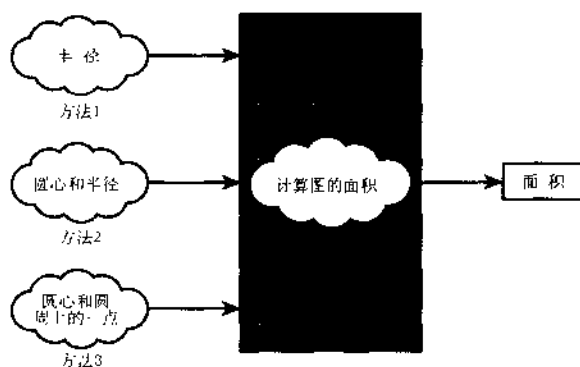


图9.1 计算圆面积的黑盒子

以二种不同的方法来提供圆参数，都将得到正确的结果。第一种方法是只提供半径；第二种方

法是提供半径和圆心；第三种方法是提供圆心和圆周上的一点。三种方法都要求 `area` 方法返回圆的面积。

编写实现上述黑盒子功能的程序时，您可能创建三个不同的方法，并将这些方法分别称为 `CalcArea`、`CalcAreaWithPoints`、`CalcAreaWithRadius` 或无数个其他独一无二的名称之一。如果编程时使用的是诸如 C 语言等非面向对象的语言，则需要创建多个函数；但使用具备多态功能的面向对象语言时，有一种更简单的解决方案——方法重载。

9.1.1 重载函数

新术语：方法重载指的是创建多个名称相同的方法。其中的每个方法都在某些方面具备唯一性，以便编译器能够区别它们。程序清单 9.1 包含一个 `Circle` 类，其 `Area` 方法被重载，以便图 9.1 所示的每种调用方式都可行。

程序清单 9.1 `circle.cs`：通过方法重载来说明多态

```
1: // circle1.cs - polymorphic area method
2: //-----
3:
4: using System;
5:
6: public class Circle
7: {
8:     public int x;
9:     public int y;
10:    public double radius;
11:    private const float PI = 3.14159F;
12:
13:    public double Area() // Uses values from data members
14:    {
15:        return Area(radius);
16:    }
17:
18:    public double Area( double rad )
19:    {
20:        double theArea;
21:        theArea = PI * rad * rad;
22:        Console.WriteLine(" The area for radius {0} is {1}", rad, theArea);
23:        return theArea;
24:    }
25:
26:    public double Area(int x1, int y1, double rad)
27:    {
28:        return Area(rad);
29:    }
30:
31:    public double Area( int x1, int y1, int x2, int y2 )
32:    {
33:        int x_diff;
34:        int y_diff;
35:        double rad;
```

```
36:
37:     x_diff = x2 - x1;
38:     y_diff = y2 - y1;
39:
40:     rad = (double) Math.Sqrt((x_diff * x_diff) + (y_diff * y_diff));
41:
42:     return Area(rad);
43: }
44:
45: public Circle()
46: {
47:     x = 0;
48:     y = 0;
49:     radius = 0.0;
50: }
51: }
52:
53: class CircleApp
54: {
55:     public static void Main()
56:     {
57:         Circle myCircle = new Circle();
58:
59:         Console.WriteLine("Passing nothing...");
60:         myCircle.Area();
61:
62:         Console.WriteLine("\nPassing a radius of 3...");
63:         myCircle.Area( 3 );
64:
65:         Console.WriteLine("\nPassing a center of {2, 4} and a radius of 3...");
66:         myCircle.Area( 2, 4, 3 );
67:
68:         Console.WriteLine("\nPassing center of {2, 3} and a point of {5, 6}...");
69:         myCircle.Area( 2, 3, 4, 5 );
70:     }
71: }
```

该程序清单的输出如下:

Passing nothing...

The area for radius {0} is 0

Passing a radius of 3...

The area for radius {3} is 28.2743110656738

Passing a center of {2, 4} and a radius of 3...

The area for radius {3} is 28.2743110656738

Passing center of {2, 3} and a point of {5, 6}...

The area for radius {2.82842712474619} is 25.1327209472656

分析：首先来看看第60、63、66和69行，它们都调用myCircle对象的Area方法，但提供的参数数目不同，该程序仍然通过编译并正确运行！

这是通过方法重载实现的。Circle 类定义了四个 Area 方法，它们之间的区别在于传递的参数数目不同。第13~16行定义了一个不接受任何参数，但仍然返回一个 double 值的 Area 方法，其方法体调用接受一个参数的 Area 方法，并传递存储在数据成员 radius 中的半径值。

第18~24行定义了第二个 Area 方法，该方法接受一个 double 值，并将其作为圆的半径。第21行使用传入的半径值计算圆的面积。第22行将圆的半径和计算得到的面积打印到屏幕上。该方法的最后部分将面积返回给调用例程。

提示：在该程序清单中，对于PI使用的不是常量值，而是在第11行声明了一个固定变量，这样，当需要修改PI的值时，只要在一个地方进行修改即可，而不用对分布在应用程序各个地方的硬编码进行修改，以后维护起来也将更容易。

第26~29行是第三个 Area 方法的定义。该方法定义有些可笑，因为计算面积只需要半径即可，该方法将半径值 rad 传递给只需要半径的 Area 方法，而不是重复实现相同的功能。面积值将依次从各个方法返回，最后到达最初的调用者。

第31~43行是最复杂的 Area 方法，它接受圆心和圆周上的一点，半径将是这两点之间的直线距离（如图9.2所示）。第40行根据两点的位置计算半径的长度。然后将得到的结果传递给只需要半径的 Area 方法，由它来完成余下的工作。

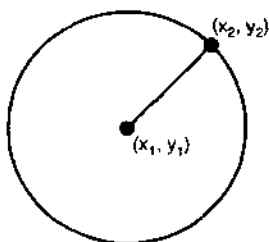


图9.2 圆心和圆周上的一点

虽然这些方法都调用另一个 Area 方法，但并非一定要这样做。这些方法可以包含各自完全独立的编码。如何编写它们取决于您，重要的是，您可以创建多个名称相同的方法，它们根据不同的值执行操作。

警告：虽然可以在名称相同的各个方法中实现完全不同的功能，但这并不意味着应该这样做，因为它们名称相同，因此最后的结果应该是类似的。

9.1.2 构造函数重载

除了可以重载常规方法外，还可以重载构造函数。重载的构造函数让您能够在创建对象的同时给它传递值。程序清单9.2包含一个其构造函数被重载的 Circle 类，该 Circle 类不同于程序清单9.1中的那个。

程序清单9.2 circle1.cs: 重载构造函数

```
1: // circle1.cs - A simple circle class with overloaded constructors
2: //-----
3:
```



```
4: using System;
5:
6: public class Circle
7: {
8:     public int x;
9:     public int y;
10:    public int radius;
11:    private const float PI = 3.14159F;
12:
13:    public double area()
14:    {
15:        double theArea;
16:        theArea = PI * radius * radius;
17:        return theArea;
18:    }
19:
20:    public double circumference()
21:    {
22:        double Circ;
23:        Circ = 2 * PI * radius;
24:        return Circ;
25:    }
26:
27:    public Circle()
28:    {
29:        x = 0;
30:        y = 0;
31:        radius = 0;
32:    }
33:
34:    public Circle( int r )
35:    {
36:        x = 0;
37:        y = 0;
38:        radius = r;
39:    }
40:
41:    public Circle ( int new_x, int new_y )
42:    {
43:        x = new_x;
44:        y = new_y;
45:        radius = 0;
46:    }
47:
48:    public Circle ( int new_x, int new_y, int r )
49:    {
50:        x = new_x;
51:        y = new_y;
52:        radius = r;
53:    }
```

```
54:
55:     public void print_circle_info()
56:     {
57:         Console.WriteLine("Circle: Center = ({0},{1})", x, y);
58:         Console.WriteLine("        Radius = {0}", radius);
59:         Console.WriteLine("        Area   = {0}", area());
60:         Console.WriteLine("        Circum = {0}", circumference());
61:     }
62: }
63:
64: class CircleApp
65: {
66:     public static void Main()
67:     {
68:         Circle first = new Circle();
69:         Circle second = new Circle(4);
70:         Circle third = new Circle(3,4);
71:         Circle fourth = new Circle(1, 2, 5);
72:
73:         Console.WriteLine("\nFirst Circle:");
74:         first.print_circle_info();
75:
76:         Console.WriteLine("\nSecond Circle:");
77:         second.print_circle_info();
78:
79:         Console.WriteLine("\nThird Circle:");
80:         third.print_circle_info();
81:
82:         Console.WriteLine("\nFourth Circle:");
83:         fourth.print_circle_info();
84:     }
85: }
```

该程序清单的输出如下:

First Circle:

Circle: Center = (0,0)

Radius = 0

Area = 0

Circum = 0

Second Circle:

Circle: Center = (0,0)

Radius = 4

Area = 50.26544

Circum = 25.13272

Third Circle:

Circle: Center = (3,4)

Radius = 0

```

        Area    = 0
        Circum  = 0

Fourth Circle:
Circle: Center = (1,2)
      Radius  = 5
      Area    = 78.53975
      Circum  = 31.41590001106262

```

分析：该程序清单的重点是Circle类的构造函数。有多个构造函数，每一个接受的参数个数都不同，第27~32行定义了第一个构造函数，您在本书的前面见到过这样的构造函数。该构造函数声明为公有的，名称为类名，这完全符合第7天介绍的格式。

第34~39行是另外三个Circle构造函数的第一个，它接受一个表示半径的整数，这个值被作为类的半径字段。

第41~46行是第三个构造函数，它与其他构造函数的区别在于接受两个整数值——圆心的x坐标和y坐标。第43~44行将对象的值设置为这些值。

第四个也是最后一个Circle构造函数位于第48~53行，它接受三个值——半径以及圆心的x坐标和y坐标。

这些方法的声明方式和名称都相同，唯一的差别是介绍的参数数目不同。

CircleApp类中的第68~71行使用了这些构造函数。第68行创建一个名叫first的新Circle对象，声明和创建的方式与以前介绍的创建对象方式相同。

第69行创建第二个对象的方式则不同，它不是使用new Circle()，而是包含添加了一个参数——4。创建该对象需要使用接受一个数字值(4)的构造函数，这正是从第34行开始的构造函数，该构造函数的格式与第69行的调用相同。

基于上述关于创建第二个对象的描述，可以很容易知道，创建第三个和第四个对象时，将调用相应的构造函数。相应的构造函数指的是其参数与调用的参数匹配的构造函数。

如果以下面的方式创建一个Circle对象，将发生什么情况呢：

```
Circle myCircle = new Circle(1, 2, 3, 4);
```

这将出错，因为它将传递四个值，而Circle类没有接受四个值的构造函数，因此上述声明无效。

9.1.3 理解方法的特征标

方法之所以可以重载是由于各个方法都有其独特性。每个方法都有其特征标(signature)。正如前一节介绍的，方法的参数个数能够决定那个方法是合适的。对于被重载方法，还有其他方式可以区分各个方法。这些差异性最终构成了方法的特征标。

方法的特征标由其参数个数和类型构成。Circle的构造函数有四个特征标：

- Circle();
- Circle(int);
- Circle(int, int);
- Circle(int, int, int);

程序清单9.1中的Area方法也有4个特征标：

- double area();

- `double area( double );`
- `double area( int, int, double );`
- `double area( int, int, int, int );`

下述重载方式是合法的:

```
MyFun( int )
MyFun( float )
MyFun( ref int )
MyFun( val int )
```

有些因素不能用作特征标的组成部分。例如返回类型, 因为调用方法时, 并不使用返回类型。

另外, 不能通过一个方法使用某种数据类型, 而另一种方法使用一个数组来得到不同的特征标。

例如, 如果您使用下面两个特征标来重载, 将出错:

```
int myMethod( int )
int myMethod( int[] )
```

也不能通过关键字 `params` 来区分特征标。本章后面将介绍如何使用 `params`。在一个地方声明下面两个方法将出错:

```
void myMethod( string, float)
void myMethod( string, params float[])
```

您可以根据需要重载方法任意多次, 只要每一个方法有独特的特征标。

9.2 使用不同数目的参数

前面介绍了如何创建方法、如何给方法传递信息以及传递信息的多种方式——包括按值传递、按引用传递以及传递可用于返回输出的变量。另外, 还介绍了使用关键字 `return` 从方法中返回一个值。所有这一切都要求以有组织的方式使用方法。

当需要给方法传递不同数目的参数时, 该如何办呢? 例如, 假设您需要将一组值累加起来, 但并不知道其中到底有多少个值。在这种情况下, 可以调用同一个例程多次, 或创建一个能接受不同数目参数的例程。请看方法 `Console.WriteLine` 和 `Console.Write`, 它们都可以接受一个字符串以及不同数目的其他参数, 这些参数的数据类型和值可以各不相同。

要接受未知数目的参数, 可以使用关键字 `params`, 该关键字用于参数列表中, 声明参数列表最后面的值。 `params` 关键字与数组一起使用。

程序清单 9.3 在一个方法中使用了关键字 `params`, 该方法可以接受不同数目的参数, 将其累计起来, 并将结果作为一个 `long` 值返回。

程序清单 9.3 使用关键字 `params`

```
1: // addem.cs - Using a variable number of arguments
2: //-----
3:
4: using System;
5:
6: public class AddEm
7: {
```

```
8:     public static long Add( params int[] args )
9:     {
10:         int ctr = 0;
11:         long Total = 0;
12:
13:         for( ctr = 0; ctr < args.Length; ctr++)
14:         {
15:             Total += args[ctr];
16:         }
17:         return Total;
18:     }
19: }
20:
21: class MyApp
22: {
23:     public static void Main()
24:     {
25:         long Total = 0;
26:
27:         Total = AddEm.Add( 1 );
28:         Console.WriteLine("Total of {1} = {0}", Total);
29:
30:         Total = AddEm.Add( 1, 2 );
31:         Console.WriteLine("Total of {1, 2} = {0}", Total);
32:
33:         Total = AddEm.Add( 1, 2, 3 );
34:         Console.WriteLine("Total of {1, 2, 3} = {0}", Total);
35:
36:         Total = AddEm.Add( 1, 2, 3, 4 );
37:         Console.WriteLine("Total of {1, 2, 3, 4} = {0}", Total);
38:     }
39: }
```

该程序清单的输出如下：

```
Total of {1} = 1
Total of {1, 2} = 3
Total of {1, 2, 3} = 6
Total of {1, 2, 3, 4} = 10
```

分析：看到该程序清单时，您会说“这可以使用一个简单的整数数组来实现”。如果您这样认为，那么您是完全正确的。对于这种简单的例子，不使用关键字`params`也能实现，但是，如果不使用关键字`params`，将无法仅使用第30、33和36行的代码就解决问题。您不能够将值传递给方法，而需要将每个值放在同一个`int`数组中，然后传递该数组。关键字`params`让编译器来为您完成这些工作。

仔细查看该程序清单，您将发现它并没有执行什么复杂的工作。第6~19行创建了一个名为`AddEm`的类，这个类只有一个名为`Add`静态函数，它接受存储在数组`args`中的不同数目的整数。由于使用了关键字`params`，因此可以分别传递整数，而不必将其填充到同一个数组中，然后传递该数组。

AddEm 方法非常简单。第 13~16 行的 for 循环遍历数组 args。记住，该数据是使用传递给方法 AddEm 的整数值创建的。和其他数组一样，您也可以检查该数组的标准属性和方法，这包括使用 args.Length 方法来确定数组的长度。for 循环从数组的开始循环到数组的结尾，并将每个数字加入到累积变量 Total 中。然后在第 17 行将累计值返回给调用方法。

第 21~39 行的 MyApp 类使用 AddEm 方法将所有的数字相加。从中可以看到，调用该方法时，传递的整数个数各不相同。您还可以传递更多的整数来调用该方法，而它仍将正确运行。

注意：这里没有创建 AddEm 对象，因为 Add 方法是静态的，因此可以使用类名 AddEm 来调用它。这意味着即使不创建对象，也可以调用该方法。

9.2.1 使用 params 来指定多种数据类型

前一个例子中所有的可变参数都是 int 型的。由于所有的数据类型都是基于同一个类——object，因此可以使用数据类型 object 传递各种不同数据类型的参数。程序清单 9.4 虽然不实用，但却说明了可以传递任何数据类型的值。

程序清单 9.4 grabgae.cs: 传递不同数据类型的值

```
1: // grabgae.cs - Using a variable number of arguments
2: //           of different types
3: //-----
4:
5: using System;
6:
7: public class Garbage
8: {
9:     public static void Print( params object[] args )
10:    {
11:        int ctr = 0;
12:
13:        for( ctr = 0; ctr < args.Length; ctr++)
14:        {
15:            Console.WriteLine("Argument {0} is: {1}", ctr, args[ctr]);
16:        }
17:    }
18: }
19:
20: class MyApp
21: {
22:     public static void Main()
23:     {
24:         long ALong = 1234567890987654321L;
25:         decimal ADec = 1234.56M;
26:         byte AByte = 42;
27:         string AString = "Cole McCrary";
28:
29:         Console.WriteLine("First call...");
30:         Garbage.Print( 1 ); // pass a simple integer
31:
32:         Console.WriteLine("\nSecond call...");
```

```
33:     Garbage.Print( );    // pass nothing
34:
35:     Console.WriteLine("\nThird call...");
36:     Garbage.Print( ALong, ADec, Abyte, AString ); // Pass lots
37:
38:     Console.WriteLine("\nFourth call...");
39:     Garbage.Print( AString, "is cool", '!' );    // more stuff
40: }
41: }
```

该程序清单的输出如下：

```
First call...
Argument 0 is: 1

Second call...

Third call...
Argument 0 is: 1234567890987654321
Argument 1 is: 1234.56
Argument 2 is: 42
Argument 3 is: Cole McCrary

Fourth call...
Argument 0 is: Cole McCrary
Argument 1 is: is cool
Argument 2 is: !
```

分析：第9~17行是Garbage类的Print方法，它被声明为可接受不同数目的对象。任何数据类型都是对象，因此该方法可以接受任何数据类型的值。方法内的代码很容易。从输出中可以知道，第30行首次调用Garbage.Print方法打印的是一个值——1。

在第32行，第二次调用该方法时，没有传递任何参数。方法 Garbage.Print 仍将被调用，但该方法将不打印任何东西。在第一次检查 args.Length 的值后，for 语句便结束了。

第三次和第四次对 Garbage.Print 的调用打印其他一些不同的值。通过使用 object 类型，任何数据类型的变量或字面值都可以传递给该方法。

注意：第一周的课程中介绍过，以L结尾的字面值将被视为long值，以M结尾的字面值将被视为decimal（见该程序清单的第24和25行）。

9.2.2 详谈 params

有必要对关键字 params 导致的结果做更详细的解释。当值被传递给方法时，编译器首先查看是否有匹配的方法。如果有，则调用该方法；如果没有，编译器将查看是否有包含参数 params 的方法。如果找到这样的方法，则使用它。编译器将这些值放到一个数组中，并将该数组传递给方法。以程序清单 9.3 中最后一次对 AddEm 类的 Add 方法的调用为例：

```
AddEm.Add(1, 2, 3, 4);
```

编译器将在幕后完成以下工作：

```
int[] x = new int[4];
```

```
int[0] = 1;
int[1] = 2;
int[2] = 3;
int[3] = 4;
AddEm.Add(x);
```

对于程序清单 9.4，编译器将创建并使用一个 `object` 类型的数组，而不是 `int` 类型的数组。

警告：别忘了，数组元素的编号是从0而不是1开始的。

9.2.3 Main 方法和命令参数

您知道，`Main` 是一个特殊的方法，因为它总是首先被调用。`Main` 方法也可以接受不同数目的参数，但不需要在 `Main` 方法中使用 `params` 关键字。

之所以不需要使用关键字 `params`，是由于命令行参数将自动加入到一个字符串数组中。这基本上与 `params` 的功能相同。由于命令行参数已被加入到一个数组中，因此不需要使用关键字 `params`。

声明 `Main` 方法时，如果期望传递参数，则标准的做法是使用下面的格式：

```
public static [int | void] Main( string[] args)
```

其中 `int` 和 `void` 是可选的。返回整数时使用 `int`，返回其他数据类型则使用 `void`。这里的重点是参数列表：一个名为 `args` 的字符串数组。可将 `args` 改为任何名称，但几乎所有的 C# 程序员都将使用 `args`。程序清单 9.5 演示了如何使用命令行参数。

程序清单 9.5 command.cs：使用命令行参数

```
1: // command.cs - Checking for command-line arguments
2: //-----
3: .
4: using System;
5:
6: class CommandLine
7: {
8:     public static void Main(string[] args)
9:     {
10:         int ctr=0;
11:         if (args.Length <= 0 )
12:         {
13:             Console.WriteLine("No Command Line arguments were provided.");
14:             return;
15:         }
16:         else
17:         {
18:             for( ctr = 0; ctr < args.Length; ctr++)
19:             {
20:                 Console.WriteLine("Argument {0} is {1}", ctr+1, args[ctr]);
21:             }
22:         }
23:     }
24: }
```


下面是不提供任何参数时，该程序清单的输出：

```
C:\code\Day09>command
No Command Line argument were provided.
```

下面是提供命令行参数时，该程序清单的输出：

```
C:\code\Day09>command xxx 123 456 789.012
Argument 1 is xxx
Argument 2 is 123
Argument 3 is 456
Argument 4 is 789.012
```

分析：该程序清单简单扼要，从第8行开始的Main函数接受命令行参数，其声明中包含参数 `string[] args`，因此将捕获任何命令行参数。第11行检查数组 `args` 的数据成员 `Length`。如果为0，则打印一条消息，指出没有提供任何命令行参数；否则执行第18~21行，使用一个for循环打印每一个值。第20行不是从0（而是将计数器加1，从而从1）开始标识各个参数，这样可以避免最终用户有这样的疑问，即为什么第一个参数被称为0。毕竟最终用户不一定是一名C#程序员。

应该	不应该
务必理解方法重载	应尽可能将各种东西设置为私有的，通过属性来提供对私有数据成员的公有访问权限
务必以最常用的方式重载方法	编程时别忽略了命令行参数，可以让程序接受命令行参数，并给操作系统返回一个值

9.3 作用域

新术语：理解被运行阶段环境释放之前，变量的存活时间至关重要。变量的寿命及其可访问性被称为**作用域**。作用域有许多级别，最常用的两个是局部和全局。

全局作用域指的是在整个程序清单中都可见，因此可用。在小范围内可用的变量被称为局部的。

9.3.1 局部作用域

最小的作用域是代码块。代码块可以包含一个简单的循环语句，请看程序清单 9.6 中第 15 行的 `x`，它的值是多少呢？

程序清单 9.6 `scope.cs`：在局部变量的作用域之外引用它

```
1: // scope.cs - Local scope with an error
2: // *** You will get a compile error ***
3: //-----
4:
5: using System;
6:
7: class Scope
8: {
9:     public static void Main()
10:    {
11:        for( int x; x < 10; x++ )
12:        {
```

```

13:         Console.WriteLine("x is {0}", x);
14:     }
15:     Console.WriteLine("Out of For Loop. x is {0}", x);
16: }
17: }

```

编译该程序清单时，将出现下面的错误：

```
scope.cs(15,55): error CS0103: The name 'x' does not exist in the class or namespace 'Scope'
```

您可能认为 x 的值应为 10，但实际上在这里引用 x 是错误的。该变量在第 11 行被声明为 `for` 语句的一部分，`for` 语句结束后，便在其作用域之外了。离开其作用域，便不能再使用它，因此会出错。

现在来看一下程序清单 9.7。它在 `for` 语句中包含一个与程序清单 9.6 类似的声明，但在第二个 `for` 语句中，再次声明了 x 变量，这会导致错误吗？

程序清单 9.7 scope2.cs: 声明多个局部变量 x

```

1: // scope2.cs - Local scope.
2: //-----
3:
4: using System;
5:
6: class Scope
7: {
8:     public static void Main()
9:     {
10:         for( int x = 1; x < 5; x++ )
11:         {
12:             Console.WriteLine("x is {0}", x);
13:         }
14:
15:         // Second for statement trying to redeclare x...
16:         for( int x = 1; x < 5; x++ )
17:         {
18:             Console.WriteLine("x is {0}", x);
19:         }
20:     }
21: }

```

该程序清单的输出如下：

```

x is 1
x is 2
x is 3
x is 4
x is 1
x is 2
x is 3
x is 4

```

分析：该程序清单能够运行！其中每个x变量都是其所在代码块（for语句）内的局部变量，因此，所有的x变量都是彼此独立的。

现在来看程序清单 9.8 及其使用的多个 x 变量

程序清单 9.8 scope3.cs: 多个 x 变量

```
1: // scope3.cs - Local scope.
2: // *** Error if lines are uncommented ***
3: //-----
4:
5: using System;
6:
7: class Scope
8: {
9:     static int x = 987;
10:
11:     public static void Main()
12:     {
13:         Console.WriteLine("x is {0}", x);
14:
15:         //      for( int x = 1; x < 5; x++ )
16:         //      {
17:         //          Console.WriteLine("x is {0}", x);
18:         //      }
19:         Console.WriteLine("x is {0}", x);
20:     }
21: }
```

该程序清单的输出如下：

```
x is 987
x is 987
```

分析：该程序清单的第15~18行被注释掉了。输入该程序清单时，应原原本本地输入这些注释行，然后编译并执行该程序清单。得到的输出如上所示。正如您预想的，第13行和19行打印的是类中的静态变量x。

第15~18行声明了一个局部变量x，它只能在for循环内部使用。根据前一节的介绍，您可能认为，除掉这些代码前面的注释标记后，程序将正常运行——for循环使用其局部变量x，而其他方法使用类中的变量x。但是您错了！编译器如何知道第17行使用的不是类中的变量x呢？无法知道。除掉第15~18行的注释标记，并重新编译该程序清单时，将得到下面的结果：

```
scope3.cs(15,17): error CS0136: A local variable named 'x' cannot be declared in this scope
because it would give a different meaning to 'x', which is already used in a 'parent or current'
scope to denote something else
```

编译器不知道使用哪个x变量，局部变量x和类的变量x之间出现了冲突。有一种方法可以避免这个问题。

9.3.2 区分类变量和局部变量

区分类变量和局部变量的方法之一是总是引用类。本书前面介绍了如何引用，但有必要在这里重温一下。对于程序清单 9.8 中出现的错误，解决方案有两种：一是修改局部变量的名称，一是在第 14 和 19 行明确地指出是类变量。

根据声明变量的方式，明确地引用类变量的方式有两种。如果类变量是标准的、非静态变量，可以使用关键字 `this`。例如，要访问类的数据成员 `x`，使用 `this.x`。

如果数据成员像程序清单 9.8 中那样，是静态的，则使用类名而不是关键字 `this` 来引用它。有关关键字 `this` 和如何访问静态数据成员，请参阅第 7 天的课程。

9.3.3 类作用域限定符

对于方法和数据成员，可使用的限定符有两种：`private` 和 `public`。前三天的课程中介绍了这两个限定符，本书的后面将介绍其他的限定符。

使用 `public` 限定符时，数据成员和成员函数在类的外面可以被访问的，前面介绍过很多这样的例子；使用限定符 `private` 时，数据成员和方法只能在其所在的默认类中被访问。默认情况下，数据成员和方法是私有的。

注意：如果在类中声明变量时，没有指定是私有的还是公有的，则它将是私有的。

另外，一些语言可以在函数或方法的外面声明变量。这种变量将是全局的，在程序的任何地方都可以使用它。C#不允许在类的外面声明变量。

9.4 不能用于创建对象的类

可能创建一个类，并禁止使用它来创建对象。您可能会问，为何要这样做呢？如果不能创建对象来访问类，那么如何使用它呢？实际上，您已经使用过大量不能用于创建对象的类。例如 `Console` 类，您在未声明 `Console` 对象的情况下，便使用了它的 `WriteLine` 和其他方法。另外，诸如 `Math` 等类允许您不声明对象便使用它们。

没有对象，如何使用类呢？静态方法和数据成员属于类，而不是各个对象。如果您声明了一个其数据和方法都为静态的类，则使用该类声明的对象将不包含任何值。程序清单 9.9 声明了一个 `MyMath` 类，其中包含确良多个执行数学运算的方法。

程序清单 9.9 MyMath.cs: 执行数学运算的方法

```
1: // MyMath.cs - Static members.
2: //-----
3:
4: using System;
5:
6: public class MyMath
7: {
8:     public static long Add( params int[] args )
9:     {
10:         int ctr = 0;
11:         long Answer = 0;
12:
13:         for( ctr = 0; ctr < args.Length; ctr++)
```

```

14:     {
15:         Answer += args[ctr];
16:     }
17:     return Answer;
18: }
19:
20: public static long Subtract( int arg1, int arg2 )
21: {
22:     long Answer = 0;
23:     Answer = arg1 - arg2;
24:     return Answer;
25: }
26: }
27:
28: class MyApp
29: {
30:     public static void Main()
31:     {
32:         long Result = 0;
33:
34:         Result = MyMath.Add( 1, 2, 3 );
35:         Console.WriteLine("Add result is {0}", Result);
36:
37:         Result = MyMath.Subtract( 5, 2 );
38:         Console.WriteLine("Subtract result is {0}", Result);
39:     }
40: }

```

该程序清单的输出如下:

```

Add result is 6
Subtract result is 3

```

分析: 第6~26行的MyMath类声明了两个方法: Subtract和Add, 它们对整数执行加法和减法运算, 并将结果返回。这些方法可以更复杂些, 这些工作留给读者去完成。

虽然没有理由去创建一个 MyMath 对象, 但您可以创建这样的对象, 不过确实可以禁止创建某种对象。

9.4.1 私有构造函数

要禁止使用某种类来创建对象, 可以创建一个私有的构造函数。正如前面介绍的, 带关键字 `private` 的方法只能在其所属的类中调用。这意味着不能在类的外面调用私有构造函数。由于构造函数是在创建对象时被调用的, 因此给构造函数加上限定符 `private` 实际上可以禁止创建对象。程序清单 9.10 包含一个构造函数为私有的 MyMath 类。

程序清单 9.10 MyMath2.cs: 构造函数为私有的 MyMath 类

```

1: // MyMath2.cs - Private constructor
2: //-----
3:

```

```
4: using System;
5:
6: public class MyMath
7: {
8:     public static long Add( params int[] args )
9:     {
10:         int ctr = 0;
11:         long Answer = 0;
12:
13:         for( ctr = 0; ctr < args.Length; ctr++)
14:         {
15:             Answer += args[ctr];
16:         }
17:         return Answer;
18:     }
19:
20:     public static long Subtract( int arg1, int arg2 )
21:     {
22:         long Answer = 0;
23:         Answer = arg1 - arg2;
24:         return Answer;
25:     }
26:
27:     private MyMath()
28:     {
29:         // nothing to do here since this will never get called!
30:     }
31: }
32:
33: class MyApp
34: {
35:     public static void Main()
36:     {
37:         long Result = 0;
38:
39:         // MyMath var = new MyMath();
40:
41:         Result = MyMath.Add( 1, 2, 3 );
42:         Console.WriteLine("Add result is {0}", Result);
43:
44:         Result = MyMath.Subtract( 5, 2 );
45:         Console.WriteLine("Subtract result is {0}", Result);
46:     }
47: }
```

该程序清单的输出如下:

Add result is 6

Subtract result is 3

分析：第27~30行是类的构造函数。如果删除第39行的注释标记，并重新编译该程序清单，将发生下面的错误：

```
MyMath3.cs(39,20): error CS0122: 'MyMath.MyMath()' is inaccessible due to its protection level
```

无法创建对象。限定符 `private` 禁止您创建对象。但这并不是什么问题，因为您可以访问公有的静态类成员。

9.5 再谈名称空间

名称空间可用于帮助组织类和其他类型。您已经使用过多个由框架提供的名称空间，其中包括 `System` 名称空间，该名称空间包含大量系统方法和类，其中的 `Console` 类包含读写信息的例程。

名称空间可以包含其他名称空间、类、结构、枚举、接口和代表。其中名称空间、类、结构和枚举，您已经熟悉了，本书后面将介绍接口和代表。

9.5.1 给名称空间命名

名称空间的名称可以是任何合法的标识符，这意味着其名称必须由标准字符和下划线组成，另外还可以包含句点。和其他标识符一样，应该给名称空间取一个描述性的名称。

9.5.2 声明名称空间

要创建名称空间，可以使用关键字 `namespace` 后面跟标识该名称空间的名称。然后使用花括号使该名称空间包含的类型括起。

正如前面指出的，名称空间可以包含其他名称空间。名称空间的格式化方式与前面介绍的相同。程序清单 9.11 声明了一个名称空间。

程序清单 9.11 `namesp.cs`：声明名称空间

```
1: // namesp.cs - Declaring namespaces
2: //-----
3:
4: using System;
5:
6: namespace Consts
7: {
8:     public class PI
9:     {
10:         public static double value = 3.14159;
11:         private PI() {} // private constructor
12:     }
13:     public class three
14:     {
15:         public static int value = 3;
16:         private three() {} // private constructor
17:     }
18: }
19:
20: namespace MyMath
```

```
21: {
22:     public class Routine
23:     {
24:         public static long Add( params int[] args )
25:         {
26:             int ctr = 0;
27:             long Answer = 0;
28:
29:             for( ctr = 0; ctr < args.Length; ctr++)
30:             {
31:                 Answer += args[ctr];
32:             }
33:             return Answer;
34:         }
35:
36:         public static long Subtract( int arg1, int arg2 )
37:         {
38:             long Answer = 0;
39:             Answer = arg1 - arg2;
40:             return Answer;
41:         }
42:     }
43: }
44:
45: class MyApp
46: {
47:     public static void Main()
48:     {
49:         long Result = 0;
50:
51:         Result = MyMath.Routine.Add( 1, 2, 3 );
52:         Console.WriteLine("Add result is {0}", Result);
53:
54:         Result = MyMath.Routine.Subtract( 5, 2 );
55:         Console.WriteLine("Subtract result is {0}", Result);
56:
57:         Console.WriteLine("\nThe value of PI is {0}", Consts.PI.value );
58:         Console.WriteLine("The value of three is {0}",Consts.three.value);
59:     }
60: }
```

该程序清单的输出如下:

Add result is 6

Subtract result is 3

The value of PI is 4.14159

The value of three is 3

分析: 该程序清单在前一个程序清单的基础上做了修改, 另外, 还声明了其他几个不实用的类, 但它确实有助于说明名称空间的概念。

该程序清单声明了两个名称空间, 第6行是第一个。名称空间 `Consts` 包含两个类: `PI` 和 `three`, 第57和58行分别使用了这两个类。在这些代码行中, 必须指定名称空间、类和数据成员。从其他名称空间中 (如第57和58行) 访问这些类时, 如果省略 `Consts`, 将出错:

```
namesbad.cs(20,1): error CS1529: A using clause must precede all other namespace elements
```

但可以使用关键字 `using`, 来避免这种错误。在程序清单的开始位置添加下面的语句, 将可以消除这种错误:

```
using Consts;
```

注意: 每个文件都包含了一个全局名称空间, 即使不明确声明。该全局名称空间的所有东西在文件中指定的每个名称空间中都是可用的。

9.5.3 using 和名称空间

C#提供了关键字 `using`, 它使得使用名称空间更为容易, 并提供了两项功能。首先, 可以使用 `using` 给名称空间取一个别名; 其次, 使用 `using` 后, 不必使用全限定名称, 因此访问名称空间中的类型将更为容易。

9.5.3.1 避免使用全限定名称

您已经知道如何使用 `using` 来避免使用全限定名称。通过包含下面的代码行:

```
using System;
```

在使用名称空间 `System` 中的类和类型时, 不用再包含名称空间名称 `System`。这样, 您可以使用 `Console.WriteLine`, 而不用包含名称空间名称 `System`。对于程序清单 9.11, 可以在第5行加入如下所示的代码:

```
using Consts;
```

这样, 您便可以使用 `PI.value` 和 `three.value`, 而不必使用名称空间 `Consts` 进行限定。

警告: 必须将 `using` 语句放在其他代码元素之前, 这意味着最好将它放在程序清单的开始位置。如果试图将其放在其他位置, 将出错。

9.5.3.2 使用 using 提供别名

也可以使用关键字 `using` 给名称空间提供别名, 这让您能够给名称空间 (甚至名称空间中的类) 取一个不同的名称。别名可以是任何合法的标识符, 取别名的格式如下:

```
using aliasname = namespaceOrClassName;
```

其中 `aliasname` 是要使用的别名, `namespaceOrClassName` 是名称空间或类的名称。例如下面的代码行:

```
using doit = System.Console;
```

如果在程序清单中包含上述代码行, 则可以在需要使用 `System.Console` 的地方使用 `doit`。这样, 要将一行文本写到屏幕上, 可以使用下面的代码:

```
doit.WriteLine( "blah blah blah");
```

程序清单 9.12 使用 `System.Console` 的别名演示了“Hello World”程序

程序清单 9.12 `useit.cs`: 使用 `using` 提供别名

```
1: // useit.cs
2: //-----
3:
4: using doit = System.Console;
5:
6: class MyApp
7: {
8:     public static void Main()
9:     {
10:         doit.WriteLine("Hello World!");
11:     }
12: }
```

该程序清单的输出如下:

Hello World!

分析: 该程序清单简单易懂。第4行使用 `using` 给 `System.Console` 创建了一个别名 `doit`。然后在第10行使用该别名来打印一条消息。

应该	不应该
务必理解作用域	应尽可能将数据成员设置为私有的
应使用关键字 <code>using</code> 来简化对名称空间成员的访问	别忘了, 默认情况下, 数据成员是私有的
应使用名称空间来组织您的类	

9.6 总 结

今天的课程对前几天所学的知识进行了扩展, 介绍了如何重载方法, 以便能够接受不同数据和类型的参数。这可以通过创建具有独特特征标重载方法来实现。除了重载常规方法外, 还介绍了如何重载类的构造函数。

另外, 今天还更详细地介绍了类成员的作用域。关键字 `private` 使得成员只在其所属的类中可用; 而关键字 `public` 则使得可以在类所属的成员外访问成员。另外, 还可以创建只存活于代码块内部的变量。 `this` 关键字用于标识类的特定实例的数据成员。

今天还介绍了名称空间, 其中包含如何创建自己的名称空间。在讨论名称空间时, 还介绍了关键字 `using`, 该关键字让您引用名称空间的成员时, 不用包含全限定名称; 另外它还可用于给名称空间或类取别名。

9.7 问与答

问: 可以将 `Main` 方法声明为私有的吗?

答：可以，但这样将无法访问该方法。要运行程序，Main 方法必须是公有的。如果不能在 Main 方法所在的类的外面访问它，将无法运行程序。

问：今天的课程简要地介绍了作用域。如果没有明确地给变量指定值，则其默认值是什么？

答：很多类型的变量一开始并没有被赋值，这包括没有被初始化的结构实例变量、输出参数和局部变量；而很多类型的变量开始被赋了值，这包括静态变量、对象实例变量、被初始化的结构实例变量、数组元素、用作方法参数的值变量和引用。虽然这些变量一开始被赋了值，但对于所有的变量，都应该设置其初始值。

问：为什么不将所有的东西都声明为公有的以简化它们？

答：面向对象语言的优点之一是能够将数据和函数封装到可以被看作是一个黑盒子的类中。将成员设置为私有的，使得内部结构的修改不影响使用类的程序成为可能。

9.8 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

9.8.1 小测验

1. 重载函数是封装、继承、多态还是重用的方式之一？
2. 成员函数可以被重载多少次？
3. 下面哪些成员可被重载：
 - a. 数据成员；
 - b. 成员方法；
 - c. 构造函数；
 - d. 析构函数。
4. 哪个关键字用于使方法能够接受不同数目的参数？
5. 如何使方法可以接受不同数目、不同（甚至未知）数据类型的参数？
6. 要从命令行接受不同数目的参数，需要包含哪个关键字？
7. 类成员的默认作用域是什么？
8. 限定符 public 和 private 之间的差别何在？
9. 如何禁止使用类实例化对象？
10. using 关键字有哪两种用途？

9.8.2 练习

1. 为名为 abc 的公有函数编写方法头，该函数接受两个 short 参数，返回值类型为 byte。
2. 编写一行接受命令行参数的代码。
3. 如果有一个名为 aClass 的类，可以加入什么代码来禁止该类被用于实例化对象。
4. 下面的程序有问题吗？在编辑器中输入该程序，并编译它。如果有问题，是什么问题？

```
1: using doit = System.Console.WriteLine;
2:
3: class MyApp
4: {
5:     public static void Main()
```

```
6:  {  
7:      doit("Hello World!");  
8:  }  
9: }
```

5. 创建一个名称空间，它包含一个类和另一个名称空间，被包含的名称空间也包含一个类。
然后创建一个使用这两个类的应用程序类。

6. 创建一个包含被重载方法的程序，该重载方法有以下特征标，这些特征标都合法吗？

```
public myFunc( )  
public myFunc( int )  
public myFunc( float )  
public myFunc( ref int )  
public myFunc( ref flaot )
```

第 10 天课程

处理异常

如果每个人编写的代码都是完美无缺的，每位用户都输入正确的信息，而所有的计算机都不出现错误，则当今的大量程序将无用武之地。实际情况是，计算机、用户和程序员都不可避免地会犯错误。因此，编写程序时，必须预防意外情况。另外，编写程序时，还需要处理另一种不同的问题——异常。今天的课程包含以下内容：

- 异常处理的概念；
- 关键字 `try` 和 `catch`；
- 通过关键字 `finally`；
- 一些常见的异常及其引发原因；
- 将异常传递给其他例程；
- 定义自己的异常；
- 引发和再引发异常。

10.1 异常处理的概念

创建程序时，需要考虑各种可能出现的问题。创建那些使用来自文件、用户、服务或程序的另一部分信息的程序时，应该对信息进行检查，以确保您收到的或使用的信息是您预料之中的。来看一下第 2 天的练习 2 中创建的程序 `ListIt`，它要求您在运行程序时指定一个文件名。如果没有指定文件名将出现什么情况？如果指定的文件不存在将出现什么情况？如果指定的是一个机器码文件将出现什么情况呢？如果指定了多个文件名又将出现什么情况呢？

您当然可以不管这些问题，但用户将放弃使用您的程序。这种错误的后果各不相同。编写程序时，您应该做出这样的决定，即哪些问题足够严重，需要考虑，哪些问题不需要。优秀的程序员总会对那些可能发生的异常和意外情况进行预防。

10.1.1 通过逻辑代码预防错误

通过在代码中加入简单的编程逻辑，便可以处理大量的问题。如果一种简单的编程逻辑便可以防止一种错误，则应该加入这种逻辑。例如，您可以检查值的长度、是否有代码行参数或核实某个数字是否在合法的范围之内。使用第一周学习的编程结构，可以很容易地实现这种检查。

请考虑下列情况，如何在代码中处理这些情况呢？

- 用户试图打开一个不存在的文件；
- 给数组指定过多的元素；
- 程序中的代码将一个字符串赋给整型变量；
- 例程试图使用包含空值的引用变量。

可以编写代码来避免这些问题，但万一遗漏了一个，将出现什么情况呢？

10.1.2 导致异常的原因

如果不有计划地捕获问题，将发生异常。异常是未被捕获的编程错误，这包括逻辑错误、逻辑错误指的是结果不正确，而不是由于编码方面的问题引起的。当发生意外的错误时，运行阶段环境将终止，并引发异常。程序清单 10.1 包含一个将引发异常的错误。运行该程序清单，看看当您访问包含五个元素的数组的第六个元素（不存在）时，将出现什么情况。

程序清单 10.1 Error.cs: 导致异常

```

1: // error.cs
2: // A program that throws an exception
3: //=====
4: using System;
5:
6: class MyError
7: {
8:     public static void Main()
9:     {
10:         int [] myArray = new int[5];
11:
12:         for ( int ctr = 0; ctr < 10; ctr++ )
13:         {
14:             myArray[ctr] = ctr;
15:         }
16:     }
17: }
```

分析：编译该程序清单时不会出现错误。该程序清单并不实用，因为它没有提供实际的输出或执行任何有价值的操作。第10行创建了一个名为myArray的、包含五个元素的Int数组。第12~15行的for循环将计数器的值赋给该数组的每一个元素。0被赋给了myArray[0]，1被赋给了myArray[1]，等等。

当ctr的值等于5时，将出现什么情况呢？第12行的条件语句使得这要ctr的值小于10，就不断得将其递增。但当ctr的值等于5后，出现了问题。在第14行myArray[5]是非法的——该数组的最后一个元素是myArray[4]。运行阶段环境知道该数组的索引值不能为5，因此它引发错误，指出出现了异常情况。当您运行该程序时，可能出现一个窗口，指出异常错误，如图10.1所示。另外，运行阶段环境还会显示如下的晦涩文本：

```
Exception occurred: System.IndexOutOfRangeException: A exception of type
System.IndexOutOfRangeException was thrown at MyError.Main()
```

上述文本实际上是由您的程序正在使用的一个底层对象——数组生成的。

程序清单 10.1 的结果不合适，您不希望用户看到这样的信息。为使程序对用户是友好的，您需要以一种更为友好的方式来处理这种异常错误。另外，该程序在异常发生时突然终止，您希望在

异常发生时，仍能够控制程序。

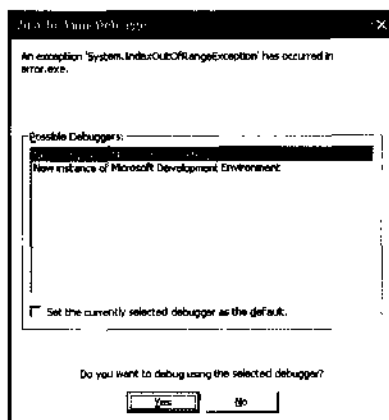


图 10.1 运行阶段环境显示的异常错误

提示：对程序清单 10.1 进行修改，在给每个元素赋值后，打印它的值。方法是在第 14 行的后面加入下面的代码：

```
Console.WriteLine("myArray[{0}] equals {1}", ctr, myArray[ctr]);
```

（运行该程序清单，您将发现程序将在打印 myArray[5] 之前终止。）

10.2 异常处理

异常处理指的是处理诸如程序清单 10.1 中出现的运行阶段错误。可以在程序中加入代码，提供更为明确的结果，而不是弹出窗口、终止程序并显示晦涩的消息。为此，需要使用关键字 `try` 和 `catch`。

10.2.1 使用 `try` 和 `catch`

关键字 `try` 和 `catch` 对于异常处理至关重要。`try` 命令让您能够给代码块加上包装，帮助发送（route）任何可能发生的异常。

关键字 `catch` 让您能够捕获 `try` 命令发送的异常。通过使用 `catch`，您能够执行代码并控制出现的问题，而不是让程序终止。程序清单 10.2 演示了命令 `try` 和 `catch` 的一种基本用途。

程序清单 10.2 tryit.cs: 使用 `try-catch`

```
1: // tryit.cs
2: // A program that throws an exception
3: //=====
4: using System;
5:
6: class MyAppClass
7: {
8:     public static void Main()
9:     {
10:         int [] myArray = new int[5];
```

```
11:
12:     try
13:     {
14:         for ( int ctr = 0; ctr < 10; ctr++ ) // Array only has 5 elements!
15:         {
16:             myArray[ctr] = ctr;
17:         }
18:     }
19:
20:     catch
21:     {
22:         Console.WriteLine("The exception was caught!");
23:     }
24:
25:     Console.WriteLine("At end of class");
26: }
27: }
```

该程序清单的输出如下：

```
The exception was caught!
At end of class
```

分析：该程序清单与10.1类似，只是使用try和catch添加了基本的异常处理而已。该程序清单的主要代码被封装到一个try语句中。该try语句是从第12行开始的，它使用花括号（第13行和18行）将要监视其异常的代码块——即操纵数组的代码——括起。

try 语句后面是从第 20 行开始的 catch 语句，该语句由第 21 行和 23 行的花括号之间的语句组成。如果在执行 try 语句中的代码时发生异常，程序流程将立刻跳到 catch 语句，执行 catch 语句中的代码（而不是像程序清单 10.1 那样显示一条晦涩的消息，并终止程序），并继续执行程序。在程序清单 10.2 中，catch 语句打印一条消息并继续执行程序，因此第 25 行对 WriteLine 方法的调用仍将被执行。

10.2.2 捕获异常信息

在程序清单 10.2 中，catch 语句将捕获 try 语句的代码执行时可能发生的任何异常。除了捕获引发的异常外，还可以在 catch 中包含参数来确定引发的是哪种异常。catch 语句的格式如下：

```
catch( System.Exception e ) { }
```

catch 语句能够将异常作为一个参数接受。在这个例子中，异常是一个名为 e 的变量。可以使用更具描述性的名称，但就这个例子而言，名称 e 是管用的。

从上面可以知道，变量 e 的类型为 System.Exception，这是一个全限定名称，表明 Exception 类型是在名称空间 System 中定义的。如果使用 using 语句将 System 名称空间包含进来，则可以将上述 catch 语句简化为：

```
catch( Exception e ) { }
```

Exception 变量 e 中包含关于发生的特定异常的描述性信息。程序清单 10.3 在程序清单 10.2 的基础上进行了修改，包含一条将任何异常都作为参数进行接受的 catch 语句。其中修改过的代码行

以粗体显示。

程序清单 10.3 tryit2.cs: 捕获异常信息

```
1: // tryit2.cs
2: // A program that throws an exception
3: //=====
4: using System;
5:
6: class MyAppClass
7: {
8:     public static void Main()
9:     {
10:         int[] myArray = new int[5];
11:
12:         try
13:         {
14:             for ( int ctr = 0; ctr < 10; ctr++ ) // Array only has 5 elements!
15:             {
16:                 myArray[ctr] = ctr;
17:             }
18:         }
19:
20:         catch( Exception e)
21:         {
22:             Console.WriteLine("The following exception was caught:\n{0}", e);
23:         }
24:
25:         Console.WriteLine("At end of class");
26:     }
27: }
```

该程序清单的输出如下:

```
The following exception was caught:
System.IndexOutOfRangeException: An exception of type System.IndexOutOfRangeException was
thrown at MyAppClass.Main()
At end of class
```

分析: 该程序清单并没有对异常做太多的处理, 但确实提供了不少的信息。第22行使用方法 `WriteLine` 打印变量 `e` 的值, 即显示关于异常的信息。从输出中可以知道, `e` 的值表明引发的异常是 `IndexOutOfRangeException`, 发生在 `MyAppClass` (应用程序类) 的 `Main()` 方法中。

该程序清单将捕获 `try` 语句中发生的所有异常, 打印的错误取决于异常的类型。实际上, 可以在程序中加入代码来处理特定的异常。

注意: 同样, 异常也被捕获, 而程序继续执行。使用 `catch` 语句, 可以防止自动显示怪异的错误消息。另外, 程序也不会异常发生时终止。

10.2.3 给 try 语句提供多个 catch 语句

程序清单 10.2 中的 catch 语句过于全面，它将捕获 try 语句中的代码发生的所有错误。可以使用更为具体的 catch 语句，事实上可以编写专门捕获特定异常的 catch 语句。程序清单 10.4 包含一条专门捕获您熟悉的异常 `IndexOutOfRangeException` 的 catch 语句。

程序清单 10.4 catchIndex.cs: 捕获特定的异常

```

1: // catchIndex.cs
2: // A program that throws an exception
3: //=====
4: using System;
5:
6: class MyAppClass
7: {
8:     public static void Main()
9:     {
10:         int [] myArray = new int[5];
11:
12:         try
13:         {
14:             for ( int ctr = 0; ctr < 10; ctr++ ) // Array only has 5 elements!
15:             {
16:                 myArray[ctr] = ctr;
17:             }
18:         }
19:
20:         catch (IndexOutOfRangeException e)
21:         {
22:             Console.WriteLine("You were very goofy trying to use a bad array index!!", e);
23:         }
24:
25:         catch (Exception e)
26:         {
27:             Console.WriteLine("Exception caught: {0}", e);
28:         }
29:
30:         Console.WriteLine("\nDone with the catch statements. Done with program.");
31:     }
32: }

```

该程序清单的输出如下:

```
You were very goofy trying to use a bad array index!!
```

```
Done with the catch statements. Done with program.
```

分析: 该程序清单使用的数组和 try 命令与前几个程序清单相同, 但第 20~23 行包含一些新内容。第 20 行的 catch 语句包含一个 `IndexOutOfRangeException` 类型的参数, 而不是一个通用异常参数 和通

用的 `Exception` 类型一样, `IndexOutOfRangeException` 类型也位于名称空间 `System` 中。顾名思义, 该异常类型是专门用于索引超出范围的情况的, 因此 `catch` 语句将只捕获这种异常。

为应付可能发生的其他异常, 第 25~28 行包含了另一个 `catch` 语句。该 `catch` 语句使用的是通用的 `Exception` 类型的参数, 因此将捕获可能发生的其他任何异常。将该程序清单的第 16 行改为如下所示:

```
16:         myArray[ctr] = 100/ctr; // division by zero....
```

重新编译并运行该程序清单, 将出现如下所示的输出:

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
    at MyAppClass.Main( )
```

```
Done with the catch statements. Done with program.
```

修改后的第 16 行将导致另一种错误。第一次执行第 14 行的 `for` 循环时, `ctr` 等于 0, 因此第 16 行执行的运算为 100 除以 0 (`ctr`)。被 0 除是非法的, 因此这将导致无穷大, 所有引发一个异常。这种异常不是索引超出范围, 因此第 20 行的 `catch` 语句将被忽略, 因为该异常与 `IndexOutOfRangeException` 不匹配。第 25 行的 `catch` 语句将捕获任何异常, 因此被执行。第 27 行打印文本 “Exception Caught:” 和变量 `e` 中的异常描述。从输出可以知道, 引发的异常是 `DivideByZeroException`。

10.2.4 理解异常的处理顺序

在程序清单 10.4 中, 两个 `catch` 语句的顺序至关重要。总是在前面捕获较具体的异常, 在后面捕获较通用的异常。对于程序清单 10.4, 如果改变两个 `catch` 语句的顺序:

```
catch (Exception e)
{
    Console.WriteLine("Exception caught: {0}", e);
}

catch (IndexOutOfRangeException e)
{
    Console.WriteLine("You were very goofy trying to use a bad array index!!", e);
}
```

然后重新编译该程序清单, 将出错。这是因为通用的 `catch(Exception e)` 将捕获所有的异常, 因此其他的 `catch` 语句将不会被执行。

10.3 使用 `finally` 添加最后执行的操作

有时候, 不管 `try` 语句中的代码是否成功执行, 都需要执行一个代码块。为此, C# 提供了关键字 `finally`。 `finally` 语句块中的代码总是会执行。程序清单 10.5 演示了如何使用该关键字。

程序清单 10.5 final.cs: 使用关键字 `finally`

```
1: // final.cs
2: // A program that throws an exception
```

```

3: //=====
4: using System;
5:
6: class MyAppClass
7: {
8:     public static void Main()
9:     {
10:         int [] myArray = new int[5];
11:
12:         try
13:         {
14:             for ( int ctr = 0; ctr < 10; ctr++ ) // Array only has 5 elements!
15:             {
16:                 myArray[ctr] = ctr;
17:             }
18:         }
19:
20:         // catch
21:         {
22:             Console.WriteLine("Exception caught");
23:         }
24:
25:         finally
26:         {
27:             Console.WriteLine("Done with exception handling");
28:         }
29:
30:         Console.WriteLine("End of Program");
31:     }
32: }

```

该程序清单的输出如下:

```

Exception occurred: System.IndexOutOfRangeException: An exception of type
System.IndexOutOfRangeException was thrown.
    at MyAppClass.Main()
Done with exception handling

```

分析: 该程序清单与前面的几个相同, 主要区别在于第25~28行, 这里加入了一个finally子句, 它打印一条消息。注意, 虽然catch子句(第20~23行被注释掉了)没有捕获任何异常, 但在程序结束前, finally仍将被执行。但第30行的WriteLine方法没有被执行。

删除第20~23行的注释标记, 重新编译并执行该程序, 将得到如下所示的结果:

```

Exception caught.
Done with exception handling
End of Program

```

使用 catch 语句后, finally 中的代码仍将执行。如果将第14行修改为如下所示:

```

14:         for ( int ctr = 0; ctr < 5; ctr++ )

```

然后重新编译并执行该程序，则输出如下：

```
Done with exception handling
End of Program
```

对第 14 行修改后，消除了导致异常的问题，这意味着该程序清单运行时不会出现问题。从输出可以知道，`finally` 代码块仍将被执行。由此可见，不管情况如何，`finally` 语句块都将被执行。

下面介绍一个更强大的使用异常处理的例子。程序清单 10.6 是一个更为实用的程序。

程序清单 10.6 ListFile.cs: 使用异常处理

```
1: // ListFile.cs - program to print a listing to the console
2: //-----
3:
4: using System;
5: using System.IO;
6:
7: class ListFile
8: {
9:     public static void Main(string[] args)
10:    {
11:        try
12:        {
13:
14:            int ctr=0;
15:            if (args.Length <= 0 )
16:            {
17:                Console.WriteLine("Format: ListFile filename");
18:                return;
19:            }
20:            else
21:            {
22:                FileStream fstr = new FileStream(args[0], FileMode.Open);
23:                try
24:                {
25:                    StreamReader t = new StreamReader(fstr);
26:                    string line;
27:                    while ((line = t.ReadLine()) != null)
28:                    {
29:                        ctr++;
30:                        Console.WriteLine("{0}: {1}", ctr, line);
31:                    }
32:                }
33:                catch( Exception e )
34:                {
35:                    Console.WriteLine("Exception during read/write: {0}\n", e);
36:                }
37:                finally
38:                {
```

```
39:         fstr.Close();
40:     }
41: }
42: }
43:
44: catch (System.IO.FileNotFoundException)
45: {
46:     Console.WriteLine ("ListFile could not find the file {0}", args[0]);
47: }
48: catch (Exception e)
49: {
50:     Console.WriteLine("Exception: {0}\n\n", e);
51: }
52: }
53: }
```

如果运行该程序清单时，不提供任何参数，则输出如下：

Format: ListFile filename

如果运行该程序时，提供参数 ListFile.cs，则输出的将是该程序清单，且包含行号：

```
1: // ListFile.cs - program to print a listing to the console
2: //-----
3:
4: using System;
5: using System.IO;
6:
7: class ListFile
8: {
9:     public static void Main(string[] args)
10:    {
11:        try
12:        {
13:
14:            int ctr=0;
15:            if (args.Length <= 0 )
16:            {
17:                Console.WriteLine("Format: ListFile filename");
18:                return;
19:            }
20:            else
21:            {
22:                FileStream fstr = new FileStream(args[0], FileMode.Open);
23:                try
24:                {
25:                    StreamReader t = new StreamReader(fstr);
26:                    string line;
27:                    while ((line = t.ReadLine()) != null)
28:                    {
29:                        ctr++;
```

```
30:         Console.WriteLine("{0}: {1}", ctr, line);
31:     }
32: }
33: catch( Exception e )
34: {
35:     Console.WriteLine("Exception during read/write: {0}\n", e);
36: }
37: finally
38: {
39:     fstr.Close();
40: }
41: }
42: }
43:
44: catch (System.IO.FileNotFoundException)
45: {
46:     Console.WriteLine ("ListFile could not find the file {0}", args[0]);
47: }
48: catch (Exception e)
49: {
50:     Console.WriteLine("Exception: {0}\n\n", e);
51: }
52: }
53: }
```

也可以以其他文件名为参数, 如果该文件存在, 则输出将与此类似。如果参数指示的文件不存在, 则屏幕上将显示如下所示的消息 (其中 xxx 为指定的文件名):

```
List could not find the file xxx
```

注意, 该程序运行时, 运行阶段环境不会给用户显示晦涩的异常消息, 而是提供有用的信息, 指出发生的情况。这是通过结合使用编程逻辑和异常处理实现的。

该程序清单将前面介绍的知识组合在一起。第 4 和 5 行不但包含了名称空间 `System`, 还包含了 `System` 中的名称空间 `IO`。`IO` 名称空间包含了用于发送和接收 (输入和输出) 信息的例程。

从第 7 行开始是应用程序类 `ListFile`, 该类包含一个 `Main()` 方法, 程序从这里开始执行。第 9 行的 `Main()` 方法接收一个名为 `args` 的字符串数组作为参数, `args` 中的值来自您运行程序时包含的命令行参数。

从第 11 行开始, 是与今天重点介绍的内容相关的代码。这一行声明了一个 `try` 代码块, 该代码块包含第 11 ~ 42 行的代码, 它包含大量的代码, 其中包括还有另一个 `try` 命令。如果其中的任何代码导致异常发生, 而这种异常又没有被处理, 则 `try` 语句将失败, 程序流程将进入相应的 `catch` 代码块。注意, 仅当 `try` 代码块中发生未被处理的异常时, 程序流程才会进入其 `catch` 语句。

与该最优先的 `try` 语句对应的 `catch` 代码块有两个。第一个位于第 44 ~ 47 行, 它捕获特定异常 `FileNotFoundException`。出于清晰方面的考虑, 使用的是全限定异常名, 但可以省略其中的名称空间, 因为第 5 行已经将名称空间 `System.IO` 包含进来了。`FileNotFoundException` 异常在您试图打开一个不存在的文件时发生。在这个例子中, 如果文件不存在, 第 46 行将打印一条简单的消息, 指出文件找不到。

虽然该程序处理了 `FileNotFoundException` 异常，但还是加入了第 48~51 行，以防出现其他意外的异常。这样程序可以得体地结束，而不是依赖于运行阶段环境。

下面详细地探讨 `try` 语句中的代码，以便您能更深入地理解该程序的功能。第 14 行创建了一个简单的计数器 `ctr`，用于给程序清单加上行号。

第 15 行包含了确保用户运行程序时提供一个文件名的编程逻辑。如果没有提供文件名，则结束程序。它使用 `if` 语句来检查字符串数组 `args` 的 `Length` 属性，如果数组长度不大于 0，则说明用户没有提供命令行参数。用户必须提供至少一个命令行参数，如果没有提供任何参数，则显示一条描述性消息，并使用 `return` 语句结束程序。

如果用户输入了命令行参数——`args.Length` 大于 0，则执行第 20~41 行的 `else` 语句。第 22 行创建了一个名为 `fstr` 的新对象，其类型为 `FileStream`，它有一个接受两个参数的构造函数。第一个是文件名。您传递的是用户输入的用户名，它位于 `args[0]` 中；第二个是一个关于执行何种操作的指示器，您传递的是 `FileMode.Open`，它指示 `FileStream` 对象打开一个文件，以便能够读取其内容。您使用创建的 `FileStream` 对象 `fstr` 来引用该文件。

如果第 22 行失败，并引发一个异常，将跳到第 44 行的 `catch` 语句。该行的 `catch` 语句与 `try` 语句最近。

从第 23 行开始是一个新的 `try` 代码块，其对应的 `catch` 语句位于第 33 行。第 25 行创建一个名为 `t` 的 `StreamReader` 变量，该变量与第 22 行打开的文件 `fstr` 相关。文件被看作是一个流向程序的字符流，变量 `t` 用于读取该字符流。

第 26 行声明了一个名为 `line` 的字符串变量，用于存储流向程序的一系列字符。第 27 行演示了如何使用变量 `line`。

第 27 行完成的工作很多，有必要对其进行详细地剖析。首先使用 `t` 读取一行字符。`StreamReader` 类有一个名为 `ReadLine` 的方法，它读取一行字符。一行字符指的是换行符之前的所有字符。由于 `t` 与 `fstr` 相关，而后者与用户输入的文件相关，因此 `ReadLine` 方法将返回用户指定的文件中的下一行字符。然后，该行字符被赋给字符串变量 `fstr`。读取一行字符，并将其赋给变量 `line` 后，将 `line` 与 `null` 进行比较，如果返回的字符串为空，则说明到了文件的结尾处或发生了读取错误。无论是哪种情况，当遇到 `null` 后，都没有必要继续处理该文件。

如果读取并放入到 `line` 的字符串不为空，则执行 `while` 语句中的代码块。这里是将行计数器 `ctr` 加 1，并打印一行文本，其中包含行号和 `line` 中的内容。这一过程将一直重复下去，直到 `line` 的值为 `null` 位置。

如果从文件中读取一行时遇到问题，很可能会引发异常。第 33~36 行将不获任何可能发生的异常，并显示描述性文本和异常消息。该 `catch` 语句防止运行阶段环境接管控制权，并通过提供额外的关于什么地方出了问题的信息，以帮助您和用户。

第 37~40 行包含一个 `finally` 语句，它对应于第 23 行的 `try` 语句。该 `finally` 语句不做任何工作，而只是关闭第 22 行打开的文件。由于第 22 行已成功执行——如果失败，则它将引发异常，从而跳到第 44 行的 `catch` 语句——因此，在程序结束前，需要关闭该文件。无论第 24~42 行是否出现异常，在程序结束前，都应该将文件关闭，而 `finally` 语句确保 `Close` 方法被调用。

正如您在该程序清单中看到的，`try-catch-finally` 语句可以嵌套。另外，还可用于使程序对用户更友好。

注意：本书中程序清单的行号就是使用一个与 `ListFile` 极其类似的程序加上的。

10.4 常见的异常

大量的异常被定义为.NET 框架类，前面已经介绍过其中的几个。表 10.1 列出了 System 名称空间中许多常见的异常类。

表 10.1 System 名称空间中常用的异常类

异常类名称	描 述
MemberAccessException	访问错误：类型成员（如方法）不能被访问
ArgumentException	参数错误：方法的参数无效
ArgumentNullException	参数为空：给方法传递一个不可接受的空参数
ArithmeticException	数学计算错误：由于数学运算导致的异常，其覆盖面比 DivideByZeroException 和 OverflowException 更大
ArrayTypeMismatchException	数组类型不匹配：试图将不兼容类型的数据存储到数组中时引发
DivideByZeroException	被零除：试图除以零时引发
FormatException	格式不正确：参数的格式不正确
IndexOutOfRangeException	索引超出范围：使用的索引小于 0 或比最后一个数组元素的索引还大
InvalidCastException	非法强制转换：显式转换失败时引发
MulticastNotSupportedException	不支持的组播：组合两个非空代表（将在第 14 天的课程中介绍）失败时引发
NotFiniteNumberException	无限大的值：数字不合法
NotSupportedException	方法不被执行：调用的方法在类中没有实现
NullReferenceException	空引用：引用空引用对象时引发
OutOfMemoryException	内存不足：无法为新语句分配内存时引发
OverflowException	溢出：使用 checked 关键字后，数学运算导致所赋的值过大或过小时引发
StackOverflowException	栈溢出：栈中的命令过多时引发
TypeInitializationException	错误的初始化类型：静态构造函数有问题时引发

注意：表10.1中的名称是假设您使用using语句包含了System名称空间；否则需要使用全限定名称System.ExceptionName，其中ExceptionName是表中使用的名称。

10.5 定义自己的异常类

除了框架中定义好的异常外，您还可以创建自己的异常。在 C#中，应该引发异常，而不是传回大量不同的错误码。因此总是应该在代码中处理异常，以防出现异常，这很重要。虽然这会增加程序中的代码，但可以使程序对用户更友好。

创建自己的异常类后，您可能想让它发生。为此，需要引发（throw）异常。要引发自己的异常，需要使用关键字 `throw`。

您可以引发预定义的异常，也可以引发自己的异常。预定义的异常指的是已经在名称空间中定义好的异常。例如，您可以引发表 10.1 中的所有异常。为此，需要使用关键字 `throw`，其格式如下：

```
throw (exception);
```

如果该异常不存在，还需要使用关键字 `new` 来创建它。例如，程序清单 10.7 引发一个新的 `DivideByZeroException` 异常。虽然该程序清单毫无作用，但它确实以最基本的方式说明了关键字 `throw`。

注意：使用关键字 `throw` 时，括号是可有可无的，下面的两行代码等效：

```
throw (exception);  
throw exception;
```

程序清单 10.7 zero.cs：引发异常

```
1: // zero.cs  
2: // Throwing a predefined exception.  
3: // This listing gives a runtime exception error!  
4: // =====  
5: using System;  
6:  
7: class SimpleApp  
8: {  
9:     public static void Main()  
10:    {  
11:        Console.WriteLine("Before Exception...");  
12:        throw( new DivideByZeroException() );  
13:        Console.WriteLine("After Exception...");  
14:    }  
15: }
```

该程序清单的输出如下：

```
Before Exception..
```

```
Exception occurred: System.DivideByZeroException: Attempted to divide by zero.  
at SimpleApp.Main( )
```

分析：该程序清单只是打印一条消息，并在第 12 行引发 `DivideByZeroException` 异常而已。程序执行时，出现一个运行错误，指出引发了异常。它简单，但也不实用。

编译并运行该程序时，将出现运行错误；第 13 行永远不会被执行，因为 `throw` 命令将终止程序。记住，`throw` 命令将立刻离开当前的例程。可以删除第 13 行（因为它不会被执行），以免编译器提出警告。这里之所以加入这行代码，只是为了强调异常对程序流程的影响。

注意：可以将该程序清单中的 `DivideByZeroException` 换成表 10.1 列出的任何异常，而输出中将出现相应的信息。

10.6 引发自己的异常

也可以创建并引发自己的异常，这种异常也更有价值。要创建自己的异常，必须首先声明它。声明的格式如下：

```
class ExceptionName: Exception { }
```

其中 `ExceptionName` 是要声明的异常的名称。这行代码指出您的异常是一个类，并与一个已有的类 `Exception` 相关。在关于继承的明天的课程中，将更详细地介绍这种关系。

提示：异常的名称以 `Exception` 结尾，从表 10.1 可以知道，这与预定义异常一致。

创建自己的异常只需一行代码，创建后便可以捕获它。程序清单 10.8 演示了如何创建并引发自己的异常。

程序清单 10.8 throwit.cs: 创建并引发自己的异常

```
1: // throwit.cs
2: // Throwing your own error.
3: //=====
4: using System;
5:
6: class MyThreeException : Exception { }
7:
8: class MyAppClass
9: {
10:     public static void Main()
11:     {
12:         int result;
13:
14:         try
15:         {
16:             result = MyMath.AddEm( 1, 2 );
17:             Console.WriteLine( "Result of AddEm(1, 2) is {0}", result);
18:
19:             result = MyMath.AddEm( 3, 4 );
20:             Console.WriteLine( "Result of AddEm(3, 4) is {0}", result);
21:         }
22:
23:         catch (MyThreeException)
24:         {
25:             Console.WriteLine("Ack! We don't like adding threes.");
26:         }
27:
28:         catch (Exception e)
29:         {
30:             Console.WriteLine("Exception caught: {0}", e);
31:         }
32:
33:         Console.WriteLine("\nAt end of program");
34:     }
```

```

35: }
36:
37: class MyMath
38: {
39:     static public int AddEm(int x, int y)
40:     {
41:         if(x == 3 || y == 3)
42:             throw( new MyThreeException() );
43:
44:         return( x + y );
45:     }
46: }

```

该程序清单的输出如下:

```

Result of AddEm(1, 2) is 3
Ack! We don't like adding threes.

```

At end of program

分析: 该程序清单演示了如何创建自己的异常 `MyThreeException`。该异常是在第 6 行定义的, 其格式与前面介绍的相同。这让您能够引发一个基本的异常。

介绍 `MyAppClass` 之前, 我们先来看看位于第 37 ~ 46 行的第二个类。这个类名为 `MyMath`, 只包含一个简单的静态方法 `AddEm`。`AddEm` 方法将两个数相加, 并将结果返回。第 41 行使用 `if` 语句检查传递给 `AddEm` 的值是否至少有一个为 3, 如果是, 则引发第 6 行声明的 `MyThreeException` 异常。

第 8 ~ 34 行是 `MyAppClass` 类中的 `Main` 方法, 它调用 `AddEm` 方法。这种调用是在 `try` 语句中进行的, 这样如果引发了异常, 便可以做出反应。第 16 行是对 `AddEm` 的第一次调用, 传递的值是 1 和 2。这些值不会引发异常, 因此程序将继续执行。第 19 行再次调用方法 `AddEm`, 这次传递的第一个值为 3, 它将导致 `AddEm` 方法引发异常 `MyThreeException`。第 23 行包含一个捕获 `MyThreeException` 的 `catch` 语句, 因此该语句将捕获并处理这种异常。

如果不捕获异常, 运行阶段环境将显示一条异常消息。如果将程序清单 10.8 的第 23 ~ 26 行注释掉, 然后重新编译并运行该程序, 将得到如下所示的输出:

```

Result of AddEm(1, 2) is 3
Exception caught: MyThreeException: An exception of type MyThreeException was thrown.
at MyAppClass.Main( )

```

At end of program

该消息与其他异常完全相同。您也可以给处理异常的 `catch` 语句传递一个参数, 其中包含用于通用系统消息的信息。例如, 如果将第 23 ~ 26 行修改为如下所示:

```

23:     catch (MyThreeException e)
24:     {
25:         Console.WriteLine("Ack! We don't like adding threes. \n {0}" e);
26:     }

```

则得到的结果如下所示:

```
Result of AddEm(1, 2) is 3
Ack! We don't like adding threes.
MyThreeException: An exception of type MyThreeException was thrown at MyAppClass.Main()

At end of program
```

新创建的异常的运行方式与已有的异常完全相同。

提示: 程序清单10.8创建了一个基本异常。为使之更为完整,应该给该异常加上三个构造函数。在明天学习继承后,您对这些重载方法的细节将更为清楚。就现在而言,您只需知道加入下述包含三个构造函数的代码后,异常将更为完整:

```
class MyThreeException : Exception
{
    public MyThreeException()
    {
    }

    public MyThreeException( string e ) : base (e)
    {
    }

    public MyThreeException( string e, Exception inner ) :
        base ( e, inner )
    {
    }
}
```

您可以将异常名MyThreeException替换为自己的异常名。

10.6.1 重新引发异常

既然可以引发自己的异常和系统异常,便可以重新引发已有的异常。为何要这样做呢?什么时候这样做呢?

您知道,可以捕获异常并通过执行自己的代码来做出反应。如果上述操作是在一个类中进行的,而这个类又被另一个类所调用,则您可能希望将问题告诉调用类,但之前可能自己做一些处理。

来看一个基于前面显示文件内容的程序的例子,您可以创建打开文件,计算其包含的字符数,并将结果返回给调用程序的类。如果在打开文件并计算其中的字符时发生错误,将会引发异常。您捕获该异常,将字符数设置为 0,并将其返回给调用程序。这样,调用程序将只知道返回的字节数为 0。

一种更佳的反应方式是,将字符数设置为 0,并再次引发异常,让调用程序来处理。这样调用程序将知道发生的情况。

要重新引发异常,需要在 catch 语句中包含一个异常类型的参数。下面的代码是能够重新引发异常,让调用方法对其进行处理的 catch 语句:

```
catch ( Exception e)
{
    //My personal exception logic here
    throw( e ); //e is the argument received by this catch
}
```

```
}
```

注意：当您开始编写更复杂的应用程序时，可能想进一步了解异常处理技术。今天，您学习了最重要的异常处理特性，此外还有很多其他的特性，但这种主题已超出了本书的范围。

10.7 checked 语句和 unchecked 语句

有两个 C# 关键字决定了将过大或过小的值赋给变量是否会引发异常，它们是 `checked` 和 `unchecked`。如果代码是 `checked`，则将过大或过小的值赋给变量以引发异常。如果代码 `unchecked`，则将所赋的值进行裁剪，以便将其存储到变量中。程序清单 10.9 演示了这两个关键字的用法。

程序清单 10.9 `checkit.cs`：使用关键字 `checked`

```
1: // checkit.cs
2: //=====
3:
4: using System;
5:
6: class MyAppClass
7: {
8:     public static void Main()
9:     {
10:         int result;
11:         const int topval = 2147483647;
12:
13:         for( long ctr = topval - 5L; ctr < (topval+10L); ctr++ )
14:         {
15:             checked
16:             {
17:                 result = (int) ctr;
18:                 Console.WriteLine("{0} assigned from {1}", result, ctr);
19:             }
20:         }
21:     }
22: }
23:
```

该程序清单的输出如下：

```
2147483642 assigned from 2147483642
2147483643 assigned from 2147483643
2147483644 assigned from 2147483644
2147483645 assigned from 2147483645
2147483646 assigned from 2147483646
2147483647 assigned from 2147483647
```

```
Exception occurred: System.OverflowException: An exception of type System.OverflowException
was thrown.
```

```
at MyAppClass.Main()
```

分析：第11行声明了一个名为topval的固定变量，其值为int变量能够存储的最大值2147483647。第13行的for循环的计数器上限为最大值加10。该上限被放在一个long变量中，因此万事大吉。但17行显式地将ctr的值赋给int变量result。执行该程序清单时，将出错，因为第16~19行的代码将被检查，而这些代码试图将一个比result变量能够存储的最大值还大的值赋给它。

注意：如果将第13行的+10删除，并重新编译该程序清单，则它将正常运行。这是因为没有错误。仅当所赋的值比topval大时，溢出错误才会发生。

现在修改该程序清单，以使用关键字 unchecked。将第 15 行修改为如下所示：

```
15:      unchecked
```

重新编译并执行该程序清单。该程序清单将通过编译，但输出结果出乎意料，如下所示：

```
2147483642 assigned from 2147483642
2147483643 assigned from 2147483643
2147483644 assigned from 2147483644
2147483645 assigned from 2147483645
2147483646 assigned from 2147483646
2147483647 assigned from 2147483647
-2147483648 assigned from 2147483648
-2147483647 assigned from 2147483649
-2147483646 assigned from 2147483650
-2147483645 assigned from 2147483651
-2147483644 assigned from 2147483652
-2147483643 assigned from 2147483653
-2147483642 assigned from 2147483654
-2147483641 assigned from 2147483655
-2147483640 assigned from 2147483656
```

这次没有出现异常，这是因为代码将不被检查（unchecked）。但结果却不是您想要的。

10.7.1 checked 和 unchecked 的格式

在程序清单 10.9，checked 和 unchecked 被用作语句。它们的格式如下：

```
[un]checked { //statements }
```

也可以将它们用作运算符，其格式如下：

```
[un]checked (expression)
```

其中要检查（或不检查）的表达式被放在括号中。

警告：决不要认为默认为checked或unchecked。通常，默认为checked，但有些因素将改变这种默认情况。对于命令行编译器，可以通过加入/checked选项，强行检查；如果您使用的是集成开发工具，则编译选项中应该包含检查项。可以在命令行中加入/checked-选项，以不进行检查。

10.8 总 结

今天介绍了异常处理。try 语句用于检查发生的异常，在发生异常时，可以通过 catch 语句以控

制权更大的方式来处理错误。可以为 try 语句提供多个 catch 语句，这样便可以定制不同异常的处理方式，并使用 Exception 来捕获所有的基本异常。

可以创建在异常处理代码（try 和 catch）语句执行后被执行的代码块，无论是否引发了异常，这种代码块都将被执行。这种代码块使用关键字 finally 进行标记。

最后讨论了两个与处理溢出错误相关的关键字：checked 和 unchecked。将过大值赋给变量时将发生溢出错误。可以使用关键字 checked 检查代码或表达式的是否溢出，也可以使用关键字 unchecked 忽略溢出错误。

10.9 问与答

问：看起来单独使用 catch 的功能最为强大，为何不在使用 catch 时，不带任何参数，并在其中自己完成所有的逻辑呢？

答：虽然单独使用 catch 的功能最为强大，但这样将无法知道关于引发异常的信息。因此，使用 catch(Exception e) 更佳。这让您能够获得关于引发异常的信息。如果不使用这些信息，则可以将其传递给其他类，这样这些类就可以选择是否使用这些信息。

问：所有的异常将被平等对待吗？

答：不会。实际上，异常分为两类：系统异常和应用程序异常。前者将终止应用程序，而后者不会。今天介绍的大部分是更为常见的异常——应用程序异常。更详细的关于异常以及这两类异常之间的差别的信息，请查看 .NET 框架文档或 C# 文档。

问：前面指出过，关于异常处理还有大量的内容，我需要学习这些知识吗？

答：今天介绍的关于异常处理的知识对于编写程序而言足够了。通过进一步学习异常处理方面的知识，将能够更好地操纵错误和消息。另外，您还知道如何将一个异常嵌入到另一个异常中，等等。并不一定要知道这些高级概念，但知道这些概念将使您成为一名更优秀、更高级的 C# 程序员。

10.10 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

10.10.1 小测验

1. 哪些关键字用于处理异常？
2. 下面的情况哪些应通过异常进行处理，哪些应通过常规代码进行处理？
 - a. 用户输入的值不在指定的范围之内。
 - b. 无法正确地读取文件。
 - c. 传递给方法的参数包含一个无效的值。
 - d. 传递给方法的参数的类型非法。
3. 什么会导致异常？
4. 异常是在什么时候发生的？
 - a. 编写程序时。
 - b. 编译时。

- c. 运行时。
- d. 最终用户发出请求时。
- 5. 哪个关键字用于对异常做出反应?
- 6. finally 代码块何时执行?
- 7. 单个 try 语句可以有多少个 catch 语句与之对应?
- 8. catch 语句的次序有关系吗? 为什么?
- 9. 大多数预定义的常用异常都是在哪个名称空间中定义的?
- 10. throw 命令的功能是什么?

10.10.2 练习

1. 可以使用什么代码来检查下述代码行是否引发了异常?

```
GradePercentage = MyValue / Total;
```

2. 下面的程序有问题, 错误的原因是什么?

```
int zero = 0;
try
{
    int result = 1000 / zero;
}

catch (Exception e)
{
    Console.WriteLine("Exception caught: {0}", e);
}
catch (DivideByZeroException e)
{
    Console.WriteLine("This is my error message. ", e);
}
finally
{
    Console.WriteLine("Can't get here");
}
```

3. 编写创建自己的异常类的代码, 该异常类包含今天的补充材料中建议的三个被重载的构造函数。将该异常命名为 `NegativeValueException`。

4. 在一个完整的程序清单中使用练习 3 中创建的 `NegativeValueException` 类。请在程序清单 10.8 的基础上创建该程序清单, 并创建一个名为 `SubtractEm` 类, 它在减法运算的结果为负值时, 引发异常 `NegativeValueException`。

第 11 天课程

继 承

面向对象语言最重要的功能之一是扩展已有的类,这可以通过继承来实现。今天介绍以下内容:

- 基类;
- 通过继承来扩展基类;
- 通过继承来扩充类的功能;
- 保护数据,但允许派生类共享;
- 对类进行密封;
- 使用关键字 `is` 和 `as` 将对象作为不同的类型进行操纵。

11.1 继承的基本知识

我们都从父母那里继承了一些特征,如眼睛的颜色、头发的颜色和质地(texture)等。除了这些特征外,我们还有扩展的特征。就想我们从父母那里继承并扩展特征一样,类也可以从其他类派生而来。父母的一些特征将被后代的特征覆盖,而一些特征不会。

继承让我们能够在已有类的基础上创建新类。新类可以使用原有类的所有特性,可以覆盖已有的特性、扩展已有的特性或添加自己的特性。

同一个类可以派生多个类,但每个类都只能从一个类派生而来。例如,您有一个名为 `control` 的基类,它可以派生出许多不同的控件类型。图 11.1 帮助说明了这种一对多的关系。从图的左边往右看,便可以看到这种一对多的关系;但从右边往左看,可以发现每个类都是从一个类派生而来的。

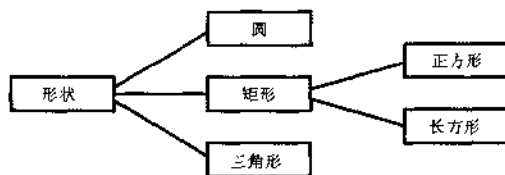


图 11.1 继承关系

讨论继承时,常常使用一些基本术语:

- 基类: 原来的类;
- 父类: 同基类的含义相同;

- 派生类：通过继承基类创建的新类；
- 子类：含义与派生类相同。
- 单继承：派生类是使用一个（且只能是一个）基类创建的。C#只支持单继承。图 11.1 说明的便是单继承。

- 多重继承：派生类是使用两个或更多的基类创建的。C#不支持多重继承。

上面列出的并不是与继承相关的所有重要术语，今天的课程还将介绍许多其他术语。

单继承和多重继承

与 C++不同，C#不支持多重继承。多重继承指的是新的派生类是使用多个基类创建的。例如，可以使用姓名类和地址类派生出一个业务联系类。后者将包含前两个类的特征。多重继承将导致许多问题，并增加复杂程度。通过使用接口（将在第 13 天的课程中介绍），可以实现多重继承的大部分功能。

11.1.1 简单继承

理解继承的最佳方式是使用它。要继承，首先需要基类。程序清单 11.1 包含一个今天将用来说明继承的类，同时将使用该程序清单的一部分。

程序清单 11.1 基类及其用法

```

1: // inherit01.cs
2: // A relatively simple class to use as a starting point
3: //-----
4: using System;
5: using System.Text;
6:
7: class Person
8: {
9:     private string firstName;
10:    private string middleName;
11:    private string lastName ;
12:    private int    age;
13:
14:    // ToDo: Add properties to access the data members
15:
16:    public Person()
17:    {
18:    }
19:
20:    public Person(string fn, string ln)
21:    {
22:        firstName = fn;
23:        lastName = ln;
24:    }
25:
26:    public Person(string fn, string mn, string ln)
27:    {
28:        firstName = fn;

```

```
29:     middleName = mn;
30:     lastName = ln;
31: }
32:
33: public Person(string fn, string mn, string ln, int a)
34: {
35:     firstName = fn;
36:     middleName = mn;
37:     lastName = ln;
38:     age = a;
39: }
40:
41: public void displayAge()
42: {
43:     Console.WriteLine("Age {0}", age);
44: }
45:
46: public void displayFullName()
47: {
48:     StringBuilder FullName = new StringBuilder();
49:
50:     FullName.Append(firstName);
51:     FullName.Append(" ");
52:     if( middleName[0] != "" )
53:     {
54:         FullName.Append(middleName[0]);
55:         FullName.Append(". ");
56:     }
57:     FullName.Append(lastName);
58:
59:     Console.WriteLine(FullName);
60: }
61: }
62:
63: // NameApp class. Illustrates the use of the Person class
64: class NameApp
65: {
66:     public static void Main()
67:     {
68:         Person me = new Person("Bradley", "Lee", "Jones");
69:         Person myWife = new Person("Melissa", "Anne", "Jones", 21);
70:
71:         me.displayFullName();
72:         me.displayAge();
73:
74:         myWife.displayFullName();
75:         myWife.displayAge();
76:     }
77: }
```

该程序清单的输出如下：

```
Bradley L. Jones
Age 0
Melissa A. Jones
Age 21
```

分析：用于继承的类 `Person` 是在第 7~61 行定义的。虽然仅仅为了说明继承，这个类远不用这么复杂，但这也使得它更为实用。第 9~12 行包含类的四个数据成员，它们存储关于人的信息。每个数据成员的存取限定符都是 `private`，`private` 使得对数据成员的访问仅限于类的内部。要在这个类的外部修改这些值，需要加入属性。

注意：为缩短该程序清单的长度，省略了属性。您应该加入属性，也可以将存取限定符改为 `public`，以便能够在外访问这些数据成员，但建议不要这样做。最好是封装数据成员，并使用属性。

第 16~39 行包含四个构造函数，这让用户可以以多种方式创建 `Person` 对象。另外，这个类还包含两个成员方法，第 41~44 行是一个简单的 `displayAge` 方法，它显示人的年龄。第 46~60 行是一个较为复杂的方法 `displayFullName`。

`DisplayFullName` 方法使用了一个以前没有介绍过的新类 `StringBuilder`，它位于名称空间 `System` 中。具体地说，是位于名称空间 `System.Text` 中。第 5 行将名称空间 `System.Text` 包含进来，以便能够方便地使用 `StringBuilder` 类。`StringBuilder` 类创建一个字符串，这种字符串可以以普通字符串不能的方式进行操纵。例如，字符串的长度不能修改。与普通字符串相同，修改 `StringBuilder` 对象或追加字符更容易。

第 48 行创建了一个 `StringBuilder` 对象 `FullName`，用来保存格式化后的人的全名。第 50 行将姓 `firstName` 追加到新创建的字符串 `FullName` 中。由于之前 `FullName` 为空，因此第 50 行基本上是将姓复制到 `FullName` 中。第 51 行在姓的后面加上一个空格。第 52~56 行将中名的大写首字母（而不是整个中名）加入到 `FullName` 中。第 52 行使用 `if` 语句确保中名不为 ""（空字符串）。如果有中名，则第 54 行将其首写字母追加到 `FullName` 中，然后第 55 行追加一个句点和空格。最后第 57 行追加名。

最后一行将全名显示到屏幕上。您可能需要将该方法的名称改为 `getFullName`，并返回格式化后的姓名，这样调用程序可以以其他方式使用全名。

程序清单的余下部分是 `NameApp` 类，之所以提供这个类，是为了让您看到使用 `Person` 类的情况。第 68 和 69 行声明了两个名为 `my` 和 `myWife` 的 `Person` 对象。第 71~75 行调用了这些对象的方法。

11.1.2 使用继承

虽然程序清单 11.1 中的代码很多，但您应该完全理解了它。除了 `StringBuilder` 类之外，这里并没有什么新东西。但程序清单 11.2 中包含了几个新特性。例如，使用下面的格式从基类继承：

```
class derived_class : base_class
```

冒号将新的 `derived_class` 类与原有的 `base_class` 类分开，表示继承。

程序清单 11.2 inherit02.cs：基本继承

```
1: // inherit02.cs
2: // Basic inheritance.
```

```
3: //=====
4: using System;
5: using System.Text;
6:
7: class Person
8: {
9:     protected string firstName;
10:    protected string middleName;
11:    protected string lastName;
12:    private int age;
13:
14:    //ToDo: Add properties to access data members
15:
16:    public Person()
17:    {
18:    }
19:
20:    public Person(string fn, string ln)
21:    {
22:        firstName = fn;
23:        lastName = ln;
24:    }
25:
26:    public Person(string fn, string mn, string ln)
27:    {
28:        firstName = fn;
29:        middleName = mn;
30:        lastName = ln;
31:    }
32:
33:    public Person(string fn, string mn, string ln, int a)
34:    {
35:        firstName = fn;
36:        middleName = mn;
37:        lastName = ln;
38:        age = a;
39:    }
40:
41:    public void displayAge()
42:    {
43:        Console.WriteLine("Age {0}", age);
44:    }
45:
46:    public void displayFullName()
47:    {
48:        StringBuilder FullName = new StringBuilder();
49:
50:        FullName.Append(firstName);
51:        FullName.Append(" ");
52:        if( middleName != "" )
```

```
53:     {
54:         FullName.Append(middleName[0]);
55:         FullName.Append(" ");
56:     }
57:     FullName.Append(lastName);
58:
59:     Console.WriteLine(FullName);
60: }
61: }
62:
63: class Employee : Person
64: {
65:     private ushort hYear;
66:
67:     public ushort hireYear
68:     {
69:         get { return(hYear); }
70:         set { hYear = value; }
71:     }
72:
73:     public Employee() : base()
74:     {
75:     }
76:
77:     public Employee( string fn, string ln ) : base( fn, ln)
78:     {
79:     }
80:
81:     public Employee(string fn, string mn, string ln, int a) :
82:         base(fn, mn, ln, a)
83:     {
84:     }
85:
86:     public Employee(string fn, string ln, ushort hy) : base(fn, ln)
87:     {
88:         hireYear = hy;
89:     }
90:
91:     public new void displayFullName()
92:     {
93:         Console.WriteLine("Employee: {0} {1} {2}",
94:             firstName, middleName, lastName);
95:     }
96: }
97:
98: class NameApp
99: {
100:     public static void Main()
101:     {
102:         Person myWife = new Person("Melissa", "Anne", "Jones", 21);
```

```

103:     Employee me = new Employee("Bradley", "L.", "Jones", 23);
104:     Employee you = new Employee("Kyle", "Rinni", 2000);
105:
106:     myWife.displayFullName();
107:     myWife.displayAge();
108:
109:     me.displayFullName();
110:     Console.WriteLine("Year hired: {0}", me.hireYear);
111:     me.displayAge();
112:
113:     you.displayFullName();
114:     Console.WriteLine("Year hired of him: {0}", you.hireYear);
115:     you.displayAge();
116: }
117: }

```

提示：程序清单变得很长。如果不想输入所有这些代码，可以从Sam出版社的网站 (www.sampublishing.com) 或作者的网站 (www.teachyourselfcsharp.com) 下载。

该程序清单的输出如下：

```

Melissa A. Jones
Age 21
Employee: Bradley L. Jones
Year hired: 0
Age 23
Employee: Kyle Rinni
Year hired of him: 2000
Age 0

```

分析：该程序清单演示了最简单的继承方式。在今天课程的后面您将知道，当您开始编写更为复杂的程序时，一些问题便出现了。现在，您应重点理解该程序清单完成的工作。

第4~61行包含的代码几乎与程序清单11.1相同。但对类做了修改，您注意到了吗？第9~11行的存取限定符不同了，不是 `private`，而是 `protected`。由于派生类是一个位于基类外面的新类，因此不能访问基类的私有变量，基类的私有变量只能在基类的内面访问。限定符 `protected` 也有一定的限定性，但同时让派生类能够访问数据成员。Person 类中修改的地方就是这些。今天课程的后面将介绍将类作为基类时，需要修改的一些地方。

第63~96行是一个新的派生类 `Employee`。第63行使用了前面提到过的冒号。从基类 `Person` 派生出了一个新的 `Employee` 类，后者包含前者的所有功能。

第65行给 `Employee` 添加了一个名为 `hYear` 的数据成员，用于保存雇员被雇佣时的年份。这是一个私有变量，通过使用第67~71行声明的属性来访问。

第73~89行是 `Employee` 类的构造函数，其中也使用了冒号，指示方式不同而已。请看第77行，这一行声明了 `Employee` 的一个构造函数，它接受两个字符串参数：`fn` 和 `ln`。

冒号后面是关键字 `base`，可以以这种方式使用关键字 `base` 来调用基类的构造函数。第77行调用基类的构造函数 `Person`，并将传递给构造函数 `Employee` 的两个参数 (`fn` 和 `ln`) 传递给它。当基类的构造函数执行完毕后，将执行构造函数 `Employee` 中的代码（第78和79行），这里没有额外的代

码。在第 86 ~ 95 行的构造函数中，则使用属性 `hireYear` 来设置 `hYear` 的值。

第 91 ~ 95 行是 `Employee` 类的 `displayFullName` 方法，声明中使用了关键字 `new`。由于这个方法名称与基类的一个成员方法相同，同时使用了关键字 `new`，因此该方法将覆盖基类中同名的方法。通过 `Employee` 对象调用 `displayFullName` 方法时，都将执行第 91 ~ 95 行的代码，而不是第 46 ~ 60 行的代码。

程序清单的最后声明了一个 `NameApp` 类，以演示如何使用派生类。第 102 ~ 104 行声明了三个对象，其他的代码则调用这些对象的方法。第 106 行调用 `myWife` 对象的 `displayFullName` 方法，由于 `myWife` 被声明为一个 `Person` 对象，因此显示的是 `Person` 类中该方法的输出，其中名使用的是缩写。第 107 行调用了 `myWife` 对象的 `displayAge` 方法。

第 109 行调用了 `me` 对象的 `displayFullName` 方法。由于在第 103 行，`me` 被声明为一个 `Employee` 对象，因此调用的将是 `Employee` 类中的该方法。第 111 行调用了 `me` 对象的 `displayAge` 方法。由于 `Employee` 类没有 `displayAge` 方法，因此程序将自动检查基类是否有这样的方法，然后使用基类的 `displayAge` 方法。

第 110 行使用 `Employee` 类的属性显示 `hireYear`。如果试图使用 `myWife` 对象的 `hireYear`，将出现什么情况呢？如果在第 107 行后面加上下面的代码行，将出现什么情况呢？

```
Console.WriteLine("Year hired: {0}", myWife.hireYear);
```

这将出错。不能通过基类访问派生类中的方法，而 `Person` 类又没有 `hireYear` 数据成员，因此这行代码非法。

11.1.3 在派生类的方法中使用基类的方法

关键字 `base` 也可用于直接调用基类的方法。例如，将第 91 ~ 95 行改为如下所示：

```
public new void displayFullName()
{
    Console.Write("Employee: ");
    Base.displayFullName();
}
```

这修改了派生类中新的 `displayFullName` 方法。现在，它不是自己来完成所有的工作，而是先显示一部分文本，然后调用基类的 `displayFullName` 方法。这让您能够扩展基类的功能，而不用重新编写所有的代码。

11.2 多 态

至此您学会了继承的简单应用。您知道，可以通过添加数据成员和方法来扩展基类，还可以使用关键字 `base` 来调用基类的方法。您还知道使用 `new` 关键字定义同名的方法，来覆盖基类的方法。

虽然这些都管用，但由于某些原因，还可能需要完成其他的工作。面向对象语言的重要概念之一是多态。如果继承得正确，则类将具有多态方面的优点。具体地说，您将能够创建一系列以层次方式排列的类，并以几乎相同的方式处理它们。

来看 `Employee` 和 `Person` 类。`Employee` 是 `Person`，但又不仅仅是 `Person`，它拥有 `Person` 的一切，还有其他东西。程序清单 11.1 最大限度地压缩了 `Person` 和 `Employee` 的例子，请注意该程序清单的 `NameApp` 中的声明。

程序清单 11.3 inherit03.cs: 给 Employee 和 Person 变量赋值

```
1: // inherit03.cs
2: //=====
3: using System;
4:
5: class Person
6: {
7:     protected string firstName;
8:     protected string lastName;
9:
10:    public Person()
11:    {
12:    }
13:
14:    public Person(string fn, string ln)
15:    {
16:        firstName = fn;
17:        lastName = ln;
18:    }
19:
20:    public void displayFullName()
21:    {
22:        Console.WriteLine("{0} {1}", firstName, lastName);
23:    }
24: }
25:
26: class Employee : Person
27: {
28:     public ushort hireYear;
29:
30:     public Employee() : base()
31:     {
32:     }
33:
34:     public Employee( string fn, string ln ) : base( fn, ln)
35:     {
36:     }
37:
38:     public Employee(string fn, string ln, ushort hy) : base(fn, ln)
39:     {
40:         hireYear = hy;
41:     }
42:
43:     public new void displayFullName()
44:     {
45:         Console.WriteLine("Employee: {0} {1}", firstName, lastName);
46:     }
47: }
48:
```

```

49: class NameApp
50: {
51:     public static void Main()
52:     {
53:         Employee me = new Employee("Bradley", "Jones", 1983);
54:
55:         Person Brad = me;
56:
57:         me.displayFullName();
58:         Console.WriteLine("Year hired: {0}", me.hireYear);
59:
60:         Brad.displayFullName();
61:     }
62: }

```

该程序清单的输出如下：

```

Employee: Bradley Jones
Year hired: 1983
Bradley Jones

```

分析：该程序清单的重点是第53和55行。第53行创建了一个名为me的Employee对象，并给它赋值。第55行创建了一个名为Brad的Person对象，并将其设置为等于Employee对象me，即将Employee对象赋给了Person对象。这是如何完成的呢？前面讲过，Employee是Person，而又不仅仅是Person。Employee包含Person的所有功能。从更广泛的角度说，基类的所有东西都是派生类的一部分。

可以像使用其他 Person 对象一样使用 Brad 对象。第 60 行调用了 displayFullName 方法，而全名确实是使用 Person 类的 displayFullName 方法来显示的。

由于已经将 一个 Employee 对象赋给了 Person 对象 Brad，那么可以调用 Employee 类的方法或使用其数据成员吗？例如，可以使用 Brad.hireYear 吗？这很容易进行测试，方法是在第 60 行后面添加下述代码：

```
Console.WriteLine("Year hired: ", Brad.hireYear);
```

您可能认为这将显示 1983，但是您错了！Brad 是一个 Person 对象，而 Person 类没有数据成员 hireYear，因此上述代码将导致编译错误：

```
Inherit03.cs(61,45): error CS0117: 'person' does not contain a definition for 'hireYear'
```

虽然 Employee 包含 Person 的一切，但 Person 并不包含 Employee 的一切，即使将一个 Employee 赋给它。从更广泛的角度说，派生类包含基类的一切，但基类却没有包含派生类的一切。

这种多态指的是什么呢？简单地说，您可以对多种对象类型调用同一个方法，而这种调用有效。在这个例子中，您对 Person 和 Employee 调用 displayFullName 方法，虽然这两个对象被赋给了相同的值，但调用的却是各自所属类的 displayFullName 方法。您不用指定要调用的是哪个类的方法。

11.3 虚拟方法

在面向对象编程中，使用到派生对象的基类引用的情况很常见。在前一个例子中，Brad 被声明

为一个 Person 变量，但却被赋给了一个 Employee 对象。在这种情况下，调用 displayFullName 方法时，实际调用的将是哪个类中相应的方法呢？基类的，该变量被赋给了一个 Employee 对象。在大多数情况下，您希望调用被赋给的对象所属类中的方法。

新术语：在 C# 中，这是通过虚拟方法实现的。虚拟方法让您能够调用实际被赋给的对象所属类中的方法，而不是基类的方法。

要在基类中将方法声明为虚拟的，可以在该方法的定义中使用关键字 virtual。如果这种方法在派生类中被覆盖（overridden），则在运行阶段，调用的将是变量实际所属类的方法，而不是其被声明的类的方法。这意味着可以使用基类来指向多个派生类，而调用相应的方法。

要覆盖虚拟方法，必须在派生类中指出，这是通过在声明新方法时使用关键字 override 实现的。程序清单 11.4 对 11.3 做了修改，请注意输出中不同的地方。

注意：为减少代码，该程序清单删除了一些构造函数。对于这个例子而言，这没有影响。

程序清单 11.4 inherit04.cs：使用虚拟方法

```
1: // inherit04.cs - Virtual Methods
2: //=====
3: using System;
4:
5: class Person
6: {
7:     protected string firstName;
8:     protected string lastName;
9:
10:    public Person()
11:    {
12:    }
13:
14:    public Person(string fn, string ln)
15:    {
16:        firstName = fn;
17:        lastName = ln;
18:    }
19:
20:    public virtual void displayFullName()
21:    {
22:        Console.WriteLine("{0} {1}", firstName, lastName);
23:    }
24: }
25:
26: class Employee : Person
27: {
28:     public ushort hireYear;
29:
30:     public Employee() : base()
31:     {
32:     }
33:
34:     public Employee(string fn, string ln, ushort hy) : base(fn, ln)
```

```
35:     {
36:         hireYear = hy;
37:     }
38:
39:     public override void displayFullName()
40:     {
41:         Console.WriteLine("Employee: {0} {1}", firstName, lastName);
42:     }
43: }
44:
45: // A new class derived from Person...
46: class Contractor : Person
47: {
48:     public string company;
49:
50:     public Contractor() : base()
51:     {
52:     }
53:
54:     public Contractor(string fn, string ln, string c) : base(fn, ln)
55:     {
56:         company = c;
57:     }
58:
59:     public override void displayFullName()
60:     {
61:         Console.WriteLine("Contractor: {0} {1}", firstName, lastName);
62:     }
63: }
64:
65: class NameApp
66: {
67:     public static void Main()
68:     {
69:
70:         Person Brad = new Person("Bradley", "Jones");
71:         Person me = new Employee("Bradley", "Jones", 1983);
72:         Person Greg = new Contractor("Hill", "Batfield", "Data Diggers");
73:
74:         Brad.displayFullName();
75:         me.displayFullName();
76:         Greg.displayFullName();
77:     }
78: }
```

该程序清单的输出如下:

```
Bradley Jones
Employee: Bradley Jones
Contractor: Hill Batfield
```

分析：首先来看一下该程序清单所做的修改。为减少代码，删除了一些构造函数。更重要的是，第20行是做了重要修改的第一个地方，Person类的displayFullName方法被声明为虚拟的，这指出：如果赋给Person变量的是派生类对象，则调用该方法时，实际将调用派生类中相应的方法。

第39行的Employee类是从Person类派生而来的，这里包含第二个重要的修改——使用的是关键字override，而不是new。这表明：使用值为Employee对象的Person变量调用displayFullName方法时，调用的将是Employee类中的该方法。

为使该程序清单更有趣一些，并帮助说明，第46~63行加入了另一个从Person类派生而来的类Contractors，它有自己的数据成员，用于存储雇佣的咨询人员所属的公司。这个类也包含一个displayFullName方法的覆盖版本。被调用时，该方法指出这个人是承包者（contractor）。

NameApp类中的Main方法被修改得更简单明了。第70~72行声明了三个Person变量，但每一个都被赋给不同的对象。第70行赋给的是一个Person对象；第71行是Employee对象；而第72行是Contractor对象。

警告：虽然第70~72行的每个变量（Brad、me和Greg）被赋给了不同类型的对象，但只能使用它们被声明的类型Person内的数据成员和方法。

第74~76行使用虚拟方法和覆盖方法。虽然用于调用displayFullName方法的三个变量都是Person类型的，但调用的是它们被实际赋给的值所属类中相应的方法，而并不都调用Person类的displayFullName方法。您几乎总是希望如此。

11.4 抽象类

在程序清单11.4中，并不一定非得在声明Employee和Contractor类的displayFullName方法时，使用关键字override。如果将第39和59行修改为使用new关键字，而不是override：

```
public new void displayFullName()
```

则程序清单的运行结果将不同，如下所示：

```
Bradley Jones
Bradley Jones
Hill Batfield
```

为何会出现这种情况呢？虽然基类的displayFullName被声明为虚拟的（以便实现多态），但要根据赋给变量的数据类型来调用相应的方法，还必须在派生类的方法中使用关键字override。

可以强制派生类覆盖基类的方法：将基类的方法声明为抽象的。为此，需要使用关键字abstract。抽象方法没有方法体，由派生类来提供。

当方法被声明为抽象的时，其所属的类也必须被声明为抽象的。程序清单11.5使用了抽象类，其中也包含Person、Employee和Contractor类。

程序清单 11.5 inherit05.cs：使用抽象类

```
1: // inherit05.cs - Abstract Methods
2: //=====
3: using System;
4:
```

```
5: abstract class Person
6: {
7:     protected string firstName;
8:     protected string lastName;
9:
10:    public Person()
11:    {
12:    }
13:
14:    public Person(string fn, string ln)
15:    {
16:        firstName = fn;
17:        lastName = ln;
18:    }
19:
20:    public abstract void displayFullName();
21: }
22:
23: class Employee : Person
24: {
25:     public ushort hireYear;
26:
27:     public Employee() : base()
28:     {
29:     }
30:
31:     public Employee(string fn, string ln, ushort hy) : base(fn, ln)
32:     {
33:         hireYear = hy;
34:     }
35:
36:     public override void displayFullName()
37:     {
38:         Console.WriteLine("Employee: {0} {1}", firstName, lastName);
39:     }
40: }
41:
42: // A new class derived from Person...
43: class Contractor : Person
44: {
45:     public string company;
46:
47:     public Contractor() : base()
48:     {
49:     }
50:
51:     public Contractor(string fn, string ln, string c) : base(fn, ln)
52:     {
53:         company = c;
54:     }
```

```

55:
56:     public override void displayFullName()
57:     {
58:         Console.WriteLine("Contractor: {0} {1}", firstName, lastName);
59:     }
60: }
61:
62: class NameApp
63: {
64:     public static void Main()
65:     {
66:
67:         //      Person Brad = new Person("Bradley", "Jones");
68:         Person me = new Employee("Bradley", "Jones", 1983);
69:         Person Greg = new Contractor("Hill", "Batfield", "Data Diggers");
70:
71:         //      Brad.displayFullName();
72:         me.displayFullName();
73:         Greg.displayFullName();
74:     }
75: }

```

该程序清单的输出如下:

```

Employee: Bradley Jones
Contractor: Hill Batfield

```

分析: 特别要注意的是该程序清单的第20行, `displayFullName`方法被声明为抽象的。这表明该方法将在派生类中实现, 因此这里没有该方法的方法体。

注意: 第20行是以分号结束的。

由于 `Person` 类有一个抽象方法, 因此类本身也必须被声明为抽象的。所以, 第5行加入了关键字 `abstract`。

在第36和56行的 `Employee` 和 `Contractor` 类都实现了覆盖方法 `displayFullName`。最后, 应用程序类中的第68和69行将 `Employee` 和 `Contractor` 对象赋给了两个 `Person` 变量。当第72和73行调用 `displayFullName` 方法时, 实际执行的是变量包含的数据类型所属类的方法, 而不是其被声明的数据类型所属的方法。

第67和71行被注释掉, 以防止它们执行。如果将第67行的注释标记删除, 试图创建一个 `Person` 对象, 将发生如下错误:

```

Inherit05.cs(67,22): error CS0144: Cannot create an instance of the abstract class or
interface 'person'

```

也可以尝试将第36和56行的关键字 `override` 改为 `new`, 如果基类相应的方法是抽象的, 则这将导致如下错误:

```

Inherit05.cs(23,7): error CS0534: 'Employee' does not implement inherited abstract member
'Person.displayFullName()'
Inherit05.cs(20,25): (Location of symbol related to previous error)
Inherit05.cs(43,7): error CS0534: 'Contractor' does not implement inherited abstract member

```



```
'Person.displayName()'
Inherit05.cs(20,25): (Location of symbol related to previous error)
```

编译器将确保基类的抽象方法被正确地覆盖。

11.5 密封类

创建抽象类时，期望从它派生出其他类。如果要禁止类被继承，该如何办呢？如果要封锁类，又该如何办呢？

C#提供了关键字 `sealed` 来防止类被继承。在定义类时包含限定符 `sealed`，可以禁止它被继承。程序清单 11.6 演示了一个非常简单的被密封（`sealed`）的类。

程序清单 11.6 inherit06.cs: 密封类

```
1: // inherit06.cs - Sealed Classes
2: //=====
3: using System;
4:
5: sealed public class number
6: {
7:     private float pi;
8:
9:     public number()
10:    {
11:        pi = 3.14159F;
12:    }
13:
14:     public float PI
15:    {
16:        get {
17:            return pi;
18:        }
19:    }
20: }
21:
22: //public class numbers : number
23: //{
24: //    public float myVal = 123.456F;
25: //}
26:
27: class myApp
28: {
29:     public static void Main()
30:     {
31:         number myNumbers = new number();
32:         Console.WriteLine("PI = {0}", myNumbers.PI);
33:
34:         //    numbers moreNumbers = new numbers();
35:         //    Console.WriteLine("PI = {0}", moreNumbers.PI);
```

```

36: //      Console.WriteLine("myVal = {0}", moreNumbers.myVal);
37:     }
38: }keyword to

```

该程序清单的输出如下:

```
PI = 3.14159
```

该程序清单的大部分代码都简单易懂。第 5 行声明 `number` 类时使用了限定符 `sealed`。如果删除第 22 ~ 25 行的注释标记,并重新编译该程序清单,将出现以下错误:

```

inherit06b.cs(22,14): error CS0509: 'numbers': cannot inherit from sealed class 'number'
inherit06b.cs(5,14): (Location of symbol related to previous error)

```

这是由于密封类不能被继承,第 22 行试图继承 `number` 类,但没有成功。

注意: 如果将密封类的数据成员声明为 `protected`,编译器将发出警告。应该将数据成员声明为私有的,因为其所属的类不会被继承。

11.6 终极基类: Object

C#中的一切都是类,C#中的终极基类是 `Object`,它是.NET 框架类层次结构中的根类。这意味着它是最初的基类。

基于今天介绍的知识,可以说 C#中的任何东西都是一个 `Object`,也就是说所有的数据类型和类都是从 `Object` 类派生而来的,同时所有的.NET 类中都包含 `Object` 类中的所有方法。

11.6.1 Object 类中的方法

`Object` 实例有两个有趣的方法,因此所有的类都有这两个方法。它们是 `GetType` 和 `ToString`。前者返回对象的数据类型;后者返回一个表示当前对象的字符串。程序清单 11.7 通过前面创建的 `PI` 类调用了这些方法。

程序清单 11.7 obj.cs: 任何东西都是 Object

```

1: // obj.cs - Object Properties
2: //=====
3: using System;
4:
5: sealed class PI
6: {
7:     public static float nbr;
8:
9:     static PI()
10:    {
11:        nbr = 3.14159F;
12:    }
13:
14:     static public float val()
15:    {
16:        return(nbr);
17:    }

```

```
18: }
19:
20: class MyApp
21: {
22:     public static void Main()
23:     {
24:         Console.WriteLine("PI = {0}", PI.val());
25:
26:         Object x = new PI();
27:         Console.WriteLine("ToString: {0}", x.ToString());
28:         Console.WriteLine("Type: {0}", x.GetType());
29:
30:         Console.WriteLine("ToString: {0}", 123.ToString());
31:         Console.WriteLine("Type: {0}", 123.GetType());
32:     }
33: }
```

该程序清单的输出如下：

```
PI = 3.14159
ToString: PI
Type: PI
ToString: 123
Type: System.Int32
```

分析：出于简化的目的，该程序清单的第一部分使用了程序清单11.6中的密封类PI，不同的是myApp类。第24行使用PI类的val方法来显示PI类中静态数据成员的值。从输出可以知道，这仍然是3.14159。但第26~31行的内容是新的。

第26行声明了一个名为x的变量，其数据类型为Object，但指向的却是一个PI对象。由于Object类是所有类（包含该程序清单中创建的PI类）的基类，因此可以使用Object变量指向一个新的PI对象。第27和28行调用了Object类的两个方法：GetType和ToString。这些方法指出，x的类型是PI，它存储的是一个PI对象。

第30和31行包含一些奇怪的东西。别忘了，在C#中，所有的东西（包括字面值）都是基于类的，而所有的类都是从Object派生而来的。这意味着诸如数字123等字面值实际上也是对象。使用从Object类派生而来的所有对象都可用的方法，将数字123转换为字符串的结果是123(字符串)。另外，数字123的数据类型是System.Int32（这是与C#中的int对应的.NET框架数据类型）。

11.6.2 装箱和拆箱

更深入地了解派生对象之间的关系后，我们来讨论另一个主题——装箱（boxing）和拆箱（unboxing）。

前面指出过，C#中的一切都是对象。这不太准确，而是所有的东西都可以被看作对象。前几天介绍过，值类型和引用类型的存储方式是不同的，而对象是引用类型。然而，程序清单11.7却像处理对象一样来处理字面值，这是如何实现的？

在C#中，可以将值类型转换为对象，这能自动进行。在程序清单11.7中，值123被隐式地转换为一个对象。别忘了，所有的对象都是从终极基类Object派生而来的。对于Object对象以及从Object派生而来的对象，可以执行的操作很多，其中包含确定其类型。

新术语：装箱(boxing)指的是将值类型转换为引用类型(对象)；拆箱指的是显式地将引用类型转换为值类型。被拆箱的值必须存储到相应的数据类型变量中。

拆箱需要显式地将对象转换为值，这可以通过强制转换来实现。程序清单 11.8 演示了简单的装箱和拆箱操作。图 11.2 和图 11.3 帮助说明该程序清单完成的工作。

程序清单 11.8 boxit.cs: 装箱和拆箱

```
1: // boxit.cs - boxing and unboxing
2: //=====
3: using System;
4:
5: class MyApp
6: {
7:     public static void Main()
8:     {
9:
10:         float val = 3.14F;    // Assign a value type a value
11:         object boxed = val;   // boxing val into boxed
12:
13:         float unboxed = (float) boxed; // unboxing boxed into unboxed
14:
15:         Console.WriteLine("val: {0}", val);
16:         Console.WriteLine("boxed: {0}", boxed);
17:         Console.WriteLine("unboxed: {0}", unboxed);
18:
19:         Console.WriteLine("\nTypes...");
20:         Console.WriteLine("val: {0}", val.GetType());
21:         Console.WriteLine("boxed: {0}", boxed.GetType());
22:         Console.WriteLine("unboxed: {0}", unboxed.GetType());
23:     }
24: }
```

该程序清单的输出如下:

```
val: 3.14
boxed: 3.14
unboxed: 3.14

Types...
val: System.Single
boxed: System.Single
unboxed: System.Single
```

分析：该程序清单重点介绍装箱和拆箱。第10行声明了一个值数据类型变量，并将值3.14赋给它。第11行执行装箱，将值类型val装箱到变量boxed中，后者是一个object变量。图11.2示出了val和boxed的存储方式，以说明它们之间的差别。

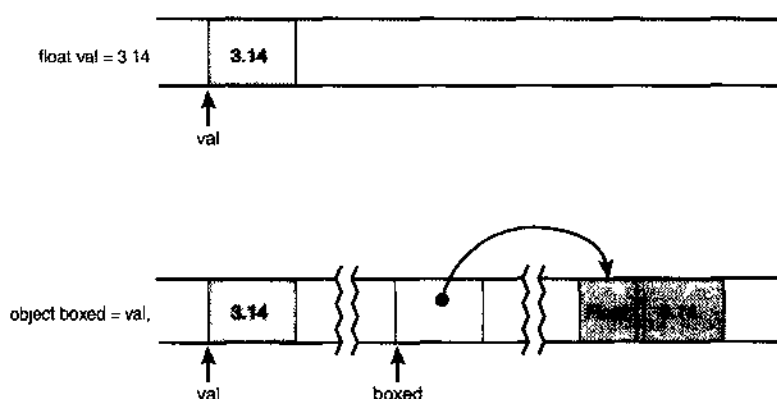


图 11.2 将值装箱

第 13 行将 boxed 中的值拆箱，并将其存储到变量 unboxed 中，后者是一个值类型变量，存储的是 boxed 中的值的拷贝——3.14。图 11.3 说明了这两个变量在内存中是如何存储的。

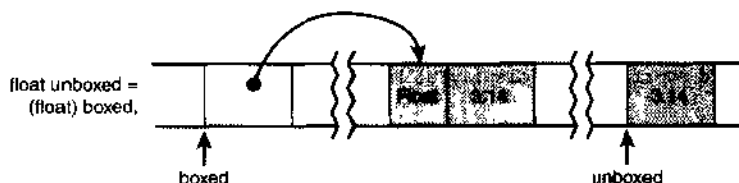


图 11.3 将值拆箱

第 20 ~ 22 行分别通过这三个变量调用 Object 的方法。您知道，val 和 unboxed 都是值类型，因此您可能会问，由于值类型并不真正存储其类型，而只存储其值，GetType 方法如何起作用呢？第 20 和 22 行，值类型自动装箱，因而转换为对象。这样，便可以调用诸如 GetType 等方法了。

11.7 将关键字 is 和 as 用于类——类转换

有两个关键字可用于类：is 和 as。

11.7.1 关键字 is

关键字 is 用于确定变量是否为指定的类型。is 语句的格式如下：

```
(expression is type)
```

其中 expression 的结果为引用类型，type 是一种有效的类型，通常是类。

如果 expression 与 type 兼容，则返回 true；否则返回 false。程序清单 11.9 虽然不实用，但说明了 is 的价值。

程序清单 11.9 islist.cs: 使用关键字 is

```
1: // islist.cs - Using the is keyword
2: //=====
3: using System;
4:
5: class Person
```

```
6: {
7:     protected string Name;
8:
9:     public Person() { }
10:
11:     public Person(string n) { Name = n; }
12:
13:     public virtual void displayFullName()
14:     {
15:         Console.WriteLine("Name: {0}", Name);
16:     }
17: }
18:
19: class Employee : Person
20: {
21:     public Employee() : base() { }
22:
23:     public Employee(string n) : base(n) { }
24:
25:     public override void displayFullName()
26:     {
27:         Console.WriteLine("Employee: {0}", Name);
28:     }
29: }
30:
31: class IsApp
32: {
33:     public static void Main()
34:     {
35:         Person pers = new Person();
36:         Object emp = new Employee();
37:         string str = "String";
38:
39:         if( pers is Person )
40:             Console.WriteLine("pers is a Person");
41:         else
42:             Console.WriteLine("pers is NOT a Person");
43:
44:         if( pers is Object )
45:             Console.WriteLine("pers is an Object");
46:         else
47:             Console.WriteLine("pers is NOT an Object");
48:
49:         if( pers is Employee )
50:             Console.WriteLine("pers is an Employee");
51:         else
52:             Console.WriteLine("pers is NOT an Employee");
53:
54:         if( emp is Person )
55:             Console.WriteLine("emp is a Person");
```

```

56:     else
57:         Console.WriteLine("emp is NOT a Person");
58:
59:     if( str is Person )
60:         Console.WriteLine("str is a Person");
61:     else
62:         Console.WriteLine("str is NOT a Person");
63: }
64: }

```

该程序清单的输出如下：

```

pers is a Person
pers is an Object
pers is NOT an Employee
emp is a Person
str is NOT a Person

```

分析：编译该程序清单时，编译器可能提出警告。这是因为对编译器而言，其中的一些 `is` 比较是显而易见的，因此它告诉您它们总是正确的。对于在程序运行时检查引用变量的类型而言，关键字 `is` 是一个非常棒的工具。

该程序清单首先声明了两个类，然后是 `Main()` 方法。这些类的功能很少。第一个类是 `Person`；第二个类是 `Employee`，它是从 `Person` 派生而来的。然后，`Main()` 方法使用了这些类。

第 35 ~ 37 行声明了三个变量。第一个是 `Person` 变量 `pers`，指向一个 `Person` 对象；第二个是 `Object` 变量 `emp`，指向一个 `Employee` 对象。您知道，可以将任何类型的对象赋给其基类变量，这意味着可以将 `Employee` 对象赋给 `Object` 变量、`Person` 变量或 `Employee` 变量。第 37 行声明了一个字符串变量。

其余的代码执行简单的检查，看这三个变量的类型是否不同。您可以再看一下输出，确定哪些比较返回 `true`，哪些返回 `false`。

注意：变量 `pers` 是 `Person` 和 `Object`；变量 `emp` 是 `Employee`、`Person` 和 `Object`。如果您不这么认为，则应该复习今天的课程。

11.7.2 关键字 `as`

`as` 运算符的功能与强制转换类似，将对象强制转换为另一种类型。目标类型必须与源类型兼容。`as` 的使用格式如下：

```
expression as DataType
```

其中 `expression` 的结果为一个引用类型，`DataType` 也是一种引用类型。相应的强制转换格式如下：

```
(DataType) expression
```

虽然 `as` 关键字的功能与强制转换类似，但它们并不是同一回事。使用强制转换时，如果出现问题——比如试图强制将字符串转换为数字，将引发异常。

但使用 `as` 关键字时，如果源类型无法转换为目标类型，则首先将其值设置为 `null`，然后再转换为目标类型，因此不会引发异常。

11.8 由不同类型的对象组成的数组

在今天课程的最后，我们来讨论另一个主题。通过使用关键字 `as` 和 `is`，您将有更大的能力。通过使用基类类型来定义数组变量，可以创建包含不同类型对象的数组。程序清单 11.10 演示了如何使用关键字 `as` 以及在一个数组中存储不同类型的对象。这些不同的数据类型必须位于同一个继承层次结构中。

程序清单 11.10 `objs.cs`: 对象数组

```
1: // objs.cs - Using an array containing different types
2: //=====
3: using System;
4:
5: public class Person
6: {
7:     public string Name;
8:
9:     public Person()
10:    {
11:    }
12:
13:    public Person(string nm)
14:    {
15:        Name = nm;
16:    }
17:
18:    public virtual void displayFullName()
19:    {
20:        Console.WriteLine("Person: {0}", Name);
21:    }
22: }
23:
24: class Employee : Person
25: {
26:     // public ushort hireYear;
27:
28:     public Employee() : base()
29:     {
30:     }
31:
32:     public Employee(string nm) : base(nm)
33:     {
34:     }
35:
36:     public override void displayFullName()
37:     {
38:         Console.WriteLine("Employee: {0}", Name);
39:     }
40: }
```



```

41:
42: // A new class derived from Person...
43: class Contractor : Person
44: {
45:     // public string company;
46:
47:     public Contractor() : base()
48:     {
49:     }
50:
51:     public Contractor(string nm) : base(nm)
52:     {
53:     }
54:
55:     public override void displayFullName()
56:     {
57:         Console.WriteLine("Contractor: {0}", Name);
58:     }
59: }
60:
61: class NameApp
62: {
63:     public static void Main()
64:     {
65:         Person [] myCompany = new Person[5];
66:         int ctr = 0;
67:         string buffer;
68:
69:         do
70:         {
71:             do
72:             {
73:                 Console.Write("\nEnter \'c\' for Contractor, \'e\' for Employee then press
ENTER: ");
74:                 buffer = Console.ReadLine();
75:             } while (buffer == "");
76:
77:             if ( buffer[0] == 'c' || buffer[0] == 'C' )
78:             {
79:                 Console.Write("\nEnter the contractor\'s name: ");
80:                 buffer = Console.ReadLine();
81:                 // do other Contractor stuff...
82:                 Contractor contr = new Contractor(buffer);
83:                 myCompany[ctr] = contr as Person;
84:             }
85:             else
86:             if ( buffer[0] == 'e' || buffer[0] == 'E' )
87:             {
88:                 Console.Write("\nEnter the employee\'s name: ");
89:                 buffer = Console.ReadLine();

```

```

90:         // Do other employee stuff...
91:         Employee emp = new Employee(buffer);
92:         myCompany[ctr] = emp as Person;
93:     }
94:     else
95:     {
96:         Person pers = new Person("Not an Employee or Contractor");
97:         myCompany[ctr] = pers;
98:     }
99:
100:     ctr++;
101:
102: } while ( ctr < 5 );
103:
104: // Display the results of what was entered....
105:
106: Console.WriteLine( "\n\n\n=====");
107:
108: for( ctr = 0; ctr < 5; ctr++ )
109: {
110:     if( myCompany[ctr] is Employee )
111:     {
112:         Console.WriteLine("Employee: {0}", myCompany[ctr].Name);
113:     }
114:     else
115:     if( myCompany[ctr] is Contractor )
116:     {
117:         Console.WriteLine("Contractor: {0}", myCompany[ctr].Name);
118:     }
119:     else
120:     {
121:         Console.WriteLine("Person: {0}", myCompany[ctr].Name);
122:     }
123: }
124: Console.WriteLine( "=====");
125: }
126: }

```

该程序清单的输入如下:

Enter 'c' for Contractor, 'e' for Employee then press ENTER: **C**

Enter the contractor's name: **Conner Bradshaw**

Enter 'c' for Contractor, 'e' for Employee then press ENTER: **E**

Enter the employee's name: **Tyler Truman**

Enter 'c' for Contractor, 'e' for Employee then press ENTER: **e**

```

Enter the employee's name: Cody McCracker

Enter 'c' for Contractor, 'e' for Employee then press ENTER: c

Enter the contractor's name: Johnny Appleseed

Enter 'c' for Contractor, 'e' for Employee then press ENTER: z

=====
Contractor: Conner Bradshaw
Employee: Tyler Truman
Employee: Cody McCracker
Contractor: Johnny Appleseed
Person: Not an Employee or Contractor
=====

```

分析：与以前的程序清单相比，该程序清单很长，但却只实现一部分功能。今天的一个练习将要求您对其进行扩展。

该程序清单旨在让您能够将输入人员名单。这里只能输入 5 个，但您可以让用户不断输入，直到输入某个值为止。程序提示用户输入 e 或 c，来表示这个人是雇员还是承包者。然后，根据用户的输入再用相应的提示，让用户输入姓名。您可以要求用户提供其他的信息，但这里没有这样做。

如果用户输入的不是 c 或 e，则程序将姓名设置为一条错误信息。您很可能使用不同的逻辑。您应该注意到了，虽然程序提示用户输入 c 或 e，但如果输入的是 C 或 E，则仍然管用。

您应该熟悉该程序清单的大部分代码。其中定义的类已被最大限度的压缩。第 26 和 45 行被注释掉了，今天的一个练习将要求您使用这些数据成员。

从第 63 行开始是该应用程序的 Main 方法。第 69 ~ 102 行包含一个 do...while 语句，它让用户输入人名。第 71 ~ 75 行包含一个嵌套 do...while 语句，它提示用户输入 e 或 c，以指示人的类型。第 74 行使用 ReadLine 来获取用户的回答。如果用户按 Enter 键，将再次进行提示。

用户输入值后，if...else 语句被用来决定执行什么操作。第 77 行只检查用户输入的文本中的第一个字符，该字符被存储在字符串数组的第一个元素（buffer[0]）中。

如果用户输入的第一个字符为 c，则执行第 79 ~ 84 行。第 79 行要求用户输入承包者的姓名，这里使用的是 Write 方法，而不是 WriteLine，这样用户便可以在提示所在的行输入姓名。如果使用 WriteLine，则将进行换行，用户将在接下来的一行中输入姓名。

第 80 行使用 ReadLine 方法读取用户输入的姓名。第 82 行创建一个名为 contr 的 Contractor 对象，并使用 ReadLine 读取的姓名初始化它。第 83 行将该新对象赋给数组 myCompany。由于 myCompany 是一个 Person 数组，因此 contr 变量将作为 Person 类型被赋给该数组。由于 Person 是 Contractor 的基类，因此可以这样做——今天课程的前面介绍过了。

第 108 ~ 123 行遍历 myCompany 数组，将其每个元素中的姓名打印到屏幕上。第 110 行检查数组元素是否为 Employee，如果是，则显示对应于雇员的输出。如果不是，则第 115 行的 if 语句检查它是不是 Constrator，如果是，则打印一条消息。如果不是，则打印一条消息，指出它只是一个 Person。

第 124 行打印虚线，以格式化输出。然后程序结束。

使用 is 和 as 让您能够在同一个数组中存储数据类型不同的对象，只要这些对象的基类相同即

可。由于所有的对象都是从 `Object` 派生而来的，因此总能找到相同的基类，从而实现上述目的。该程序清单说明了面向对象编程的一些重要特性。

11.9 总结

这是到目前为止内容最多的课程，也是最重要的。今天介绍了继承。您学习了如何创建基类和继承它们。另外，您还学习了将影响您如何使用基类和派生类的不同关键字（如 `abstract`、`virtual` 和 `protected`）以及如何密封类以禁止它被继承。

然后，您学习了如何使用这样的对象，即其包含的值与被声明的不同。可以将对象赋给其基类变量，并访问该变量。另外，可以使用 `as` 关键字将对象强制转换为其他类型，`as` 关键字的功能与强制转换运算符类似，只是如果转换时出现问题，将把被转换的变量设置为 `null`，而不是引发异常。另外，还可以使用 `is` 关键字来确定对象的类型。

11.10 问与答

问：可以继承使用其他语言编写的类吗？

答：可以。`.NET` 的特性之一便是，类可以从其他语言编写的类继承而来。这意味着 C# 类可以从使用其他语言编写的类派生而来。另外，其他语言的程序员也可以将 C# 类用作基类。

问：程序清单 11.3 包含一个将派生类对象赋给基类变量的例子，那么可以将基类对象赋给派生类变量吗？

答：可以。只要仔细，便可以。在程序清单 11.3 中，将派生类对象赋给了一个基类变量。虽然只能使用那些通过基类能够建立的东西，但派生类的其他部分并没有丢失。如果基类变量的值为派生类对象，则可以将该基类变量赋给派生类变量。这种赋值是通过强制转换实现的。对于程序清单 11.3，如果在第 55 行后面加上下面的代码，将不会出错：

```
Employee you = (Employee) Brad;
```

这是合法的，因为 `Brad` 的值是一个 `Employee` 对象。如果 `Brad` 的值不是 `Employee` 对象，则上述代码将引发非法强制转换异常（`System.InvalidCastException`）。

问：何为数据或方法覆盖？

答：数据或方法覆盖指的是在派生类中创建的方法或数据元素将替换基类的方法或数据元素，此时需要使用关键字 `new`。

问：何为强制向上转换（`upcasting`）和强制向下转换（`downcasting`）？

答：强制向下转换指的是将对象强制转换为其派生类对象；强制向上转换指的是强制将对象转换为其基类对象。在 C# 中，强制向上转换是安全的，是一种隐式操作；而强制向下转换是不安全的，要强制向下转换必须显式进行转换。

11.11 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

11.11.1 小测验

1. 在 C# 中，可以使用多少个类来派生出一个新类？
2. 下述哪个术语的含义与基类相同？
 - a. 父类。
 - b. 派生类。
 - c. 子类。
3. 哪种存取限定符用于禁止在类的外面访问数据？哪种存取限定符只允许数据在其所属的类及其派生类中使用？
4. 如何覆盖基类的方法？
5. 可以在基类中使用哪个关键字来确保派生类创建自己的方法版本？
6. 哪个关键字用于禁止类被继承？
7. 指出两个所有的类都有的方法。
8. 哪个类是终极基类，所有的类都是从它派生而来的？
9. 装箱有何功能？
10. 关键字 `as` 有何用途？

11.11.2 练习

1. 编写为 ABC 类声明构造函数的方法头，它接受两个 `int` 参数 ARG1 和 ARG2。该构造函数调用基类的构造函数，并将 ARG2 传递给它。调用是在方法头中完成的：

```
public ABC (int ARG1, int ARG2): base (ARG2)
{
}
```

2. 修改下面的类，以禁止它被用作基类：

```
1: class aLetter
2: {
3:     private static char A_ch;
4:
5:     public char ch
6:     {
7:         get { return A_ch; }
8:         set { A_ch = value; }
9:     }
10:
11:     static aLetter()
12:     {
13:         A_ch = 'X';
14:     }
15: }
```

3. 下面的代码有问题，错误在哪一行？

```
// Bug Buster

// Class definition for Person would need to be included here...

class NameApp
```

```
{  
    public static void Main()  
    {  
        Person me = new Person();  
        Object you = new Object();  
  
        me = you;  
  
        System.Console.WriteLine("Type: {0}", me.GetType());  
    }  
}
```

4. 选做题：对程序清单 11.9 进行修改，设置 `hireyear` 或 `company` 的值。并在适当的位置打印这些值。

第 12 天课程

输入和输出

前几天介绍了很多 C# 开发的核心主题，这些主题对于开发专业级的应用程序至关重要。介绍其他核心主题之前，今天先换换口味，介绍以下内容：

- 输出和输出之间的差别；
- 更多用于在控制台中显示信息的格式化选项；
- 从控制台读取信息；
- 格式化和使用字符串；
- 流的概念；
- 操纵基本的文件信息。

注意：与本书中其他的课程相比，今天的课程包含多得多的参考信息。

12.1 理解控制台输入和输出

前面介绍过术语输入和输出。今天回过头来重点介绍以更好的格式提供输出。另外，还将更详细地介绍如何通过控制台输入从用户那里获取的信息。

另外，今天还将更详细地介绍字符串。Write 和 WriteLine 方法实际上在幕后对字符串进行格式化。今天您还将学习使用其他方法格式化字符串。

12.2 格式化信息

显示信息时，如果首先将其转换为字符串，则显示起来最为简单。您知道，Console 类的 Write 和 WriteLine 方法使用字符串来显示输出。另外，.NET 框架提供了大量可用于处理字符串的方法和说明符（specifier）。说明符指示要格式的信息。

格式说明符可用于任何字符串。接下来的几节将介绍大量的格式说明符，这包括用于以下方面的说明符：

- 标准数字格式；
- 格式化金额；
- 格式化指数数值；
- 格式化指数；

- 创建自定义数字格式;
- 格式化日期和时间;
- 格式化作举。

使用这些格式说明符的方式很多。最显而易见的方式是在 `Write` 和 `WriteLine` 方法中使用说明符来提供额外的格式编排。

有些情况下,可以在调用 `ToString` 方法时使用格式说明符。前一天介绍过, `Object` 类包含一个 `ToString` 方法。由于所有的类都是从 `Object` 派生而来的,因此所有的对象都可以使用 `ToString` 方法。对于诸如 `Int32` 等类,可以给该方法传递格式化说明符,以格式化数据。例如,如果 `var` 是一个 `int` 变量,包含的值为 123,则通过下面的代码行使用金额格式标识符 (“C”) 返回值 \$123.00:

```
var.ToString("C");
```

第三种使用说明符的方式是通过 `string` 数据类型。`string` 类包含一个名为 `Format` 的静态方法。由于该方法是静态的,因此可以以 `string.Format` 的方式直接使用它。该方法的使用格式与 `Console` 中显示文本的方法相同:

```
string newString = string.Format("format_string", value(2));
```

其中 `newString` 是格式化后得到的字符串; `format_string` 是一个包含格式化说明符的字符串,可用的说明符与 `Write` 和 `WriteLine` 方法相同; `value(s)` 包含要被格式化到字符串中的值。

注意: 在介绍说明符时,将介绍更多的关于格式化值的知识。

字符 `C` 是用于金额的格式说明符。前面说过,可以将其放在引号内,并作为参数传递给 `ToString` 方法,来指定这种格式。当格式化字符串中的信息时,如使用 `WriteLine`,您将该格式说明符与变量占位符放在一起。其基本格式如下:

```
{hldr:X#}
```

其中 `hldr` 是变量的占位符编号; `X` 是用于格式化数字的说明符,对于金额格式为 `C`; `#` 是一个可选值,是需要的位数。根据说明符,该数字可执行不同类型的填充。对于金额说明符,该数字指定小数位数。程序清单 12.1 演示了前面介绍的三种使用说明符的方式,它包含一个使用金额说明符和其他说明符的小例子。今天课程的后面将更详细地介绍这些说明符以及其他说明符。

程序清单 12.1 formatit.cs: 基本的格式化方式

```
1: // formatit.cs - Different places for specifiers
2: //           to be used.
3: //-----
4:
5: using System;
6:
7: class MyApp
8: {
9:     public static void Main()
10:    {
11:        int var = 12345;
12:
13:        // Format using WriteLine
14:
```



```

15: Console.Write("You can format text using Write");
16: Console.WriteLine(" and WriteLine. You can insert");
17: Console.Write("variables (such as {0}) into a string", var );
18: Console.WriteLine(" as well as do other formatting!");
19: Console.WriteLine("\n{0:C}\n{0:C4}", var);
20: Console.WriteLine("\n{0:f}\n{0:f3}", var);
21:
22:
23: // Format using ToString
24:
25: string str1 = var.ToString("C");
26: string str2 = var.ToString("C3");
27: string str3 = var.ToString("E8");
28:
29: Console.WriteLine("\nYou can also format using ToString");
30: Console.WriteLine(str1);
31: Console.WriteLine(str2);
32: Console.WriteLine(str3);
33:
34: // Formatting with string.Format
35:
36: string str4 = string.Format("\nOr, you can use string.Format: ");
37: string str5 = string.Format("Nbr {0:F3} \n{0:C} \n{0:C0}", var);
38:
39: Console.WriteLine(str4);
40: Console.WriteLine(str5);
41: }
42: }

```

该程序清单的输出如下:

You can format text using Write and WriteLine. You can insert
variables (such as 12345) into a string as well as do other formatting!

\$12,345.00
\$12,345.0000

12345.00
12345.000

You can also format using ToString

\$12,345.00
\$12,345.000
1.23450000E+004

Or, you can use string.Format:

Nbr 12345.000
\$12,345.00
\$12,345

分析：这里不打算对该程序清单做完整的分析。这里提供该程序清单，只是为了简略地介绍一下可进行的格式化。该程序清单完成了一些格式化操作。var 中的数字被格式化为金额、小数和指数表示法。这里要提到的更重要的一点是，位于说明符后面的数字决定包含的小数位数。

12.2.1 格式化数字

可用于格式化数值的格式说明符很多，表 12.1 列出了这些说明符。

表 12.1 用于格式化数值的字符

说 明 符	描 述	默认格式	输出范例
C 或 c	金额	\$xx,xxx (\$xx,xxx)	\$12,345.67 (\$12,345.67)
D 或 d	十进制数	xxxxxx -xxxxxx	1234567 -1234567
E 或 e	指数表示法	xxxxxxE+xxx xxxxxxE+xxx -xxxxxxE+xxx -xxxxxxE+xxx xxxxxxE-xxx xxxxxxE-xxx -xxxxxxE-xxx -xxxxxxE-xxx	1.234567E+123 1.234567e+123 -1.234567E+123 -1.234567e+123 1.234567E-123 1.234567e-123 -1.234567E-123 -1.234567e-123
F 或 f	定点数	xxxxxx.xx -xxxxxx.xx	1234567.89 -1234567.89
N 或 n	数值	xx,xxx -xx,xxx	12,345.67 -12,345.67
X 或 x	十六进制数		12K87 12D687
G 或 g	通用	随具体情况而异（将使用最紧凑的格式）	
R 或 r	来回转换（Roundtrip）	将数字转换为字符串后，再转换为数字时，精度不变	

可以按前面介绍的方式使用上述格式说明符以及今天介绍的其他格式说明符。在 ToString 方法中使用说明符时，将相应的字符放在引号内，并将其作为参数传递。前面将 123 格式化为金额的范例中：

```
string newString = var.ToString("C");
```

newString 的值将为\$123.00。接下来的各节将简要地讨论上述各种格式。

警告：格式说明符可能随系统的区域设置而异。

12.2.1.1 标准格式（定点、十进制和数值）

标准格式说明符用于相应的数值类型。F 用于浮点数，D 用于标准的整数（如整型和长整型）。如果试图将 D 用于浮点数，将出现异常。

说明符 N 以更好的格式表示数字，它在数字格式中加入两位小数和逗号。

12.2.1.2 格式化金额

您知道，金额说明符是 C。可以单独使用 C，使金额带两位小数。如果希望不带小数，可以使用 C0。0 指示不要包含小数部分。

12.2.1.3 格式化指数数字

由于指数数字非常大或非常小，因此使用科学计数法来表示。说明符 E 和 e 用于格式化这种数字。输出中 E 的大小写与格式化说明符相同。

12.2.1.4 数字的通用格式

通用格式说明符（G 或 g）用于将数字格式化为最紧凑的字符串格式。这种格式符根据具体的数字决定指数表示法更紧凑还是标准格式更紧凑，并返回更紧凑的一种格式。程序清单 12.2 说明了这种说明符可能返回的不同格式。

程序清单 12.2 general.cs: 使用通用说明符

```
1: // general.cs - Using the General format specifier
2: //-----
3: using System;
4:
5: class MyApp
6: {
7:     public static void Main()
8:     {
9:         float f1 = .000000789F;
10:        float f2 = 1.2F;
11:
12:        Console.WriteLine("f1  ({0:f}). Format (G): {0:G}", f1);
13:        Console.WriteLine("f2  ({0:f}). Format (G): {0:G}", f2);
14:    }
15: }
```

该程序清单的输出如下：

```
f1  (0.00). Format (G): 7.89E-7
f2  (1.20). Format (G): 1.2
```

分析：该程序清单初始化并打印两个变量。第9和10行创建了两个变量，一个是非常小的小数，另一个是简单的数字。

第 12 和 13 行将这些值写入到控制台。其中每一行的第一个占位符使用说明符 F 将浮点数显示为定点数；第二个占位符使用格式符 G 以通用格式打印相应的变量。从输出中可以知道，将 f1 以指数方法表示更简明，而 f2 以常规格式表示更简明。

12.2.1.5 格式化十六进制数

十六进制数是以 16 为基数的计数法的。这种计数法通常用于计算机中，附录 D 介绍了如何使用十六进制。字母 x（大写或小写）用于将数字表示为十六进制。十六进制说明符自动将值转换为十六进制。

12.2.1.6 保持精度（来回转换）

将数字从一种格式转换为另一种格式时，可能会降低精度。说明符 R（或 r）用于将字符串转换为数字时保持其精度。使用该说明符后，运行阶段环境将尽可能保持原有数字的精度。

12.2.1.7 使用模式定义（picture definition）创建自定义格式

有时候，需要更精确地控制数字的格式。例如，格式化驾驶证号或社会保险号时，可能想加入短划线；而对于电话号码，则想加入括号和短划线。表 12.2 列出了一些可结合使用说明符来自定义输出格式的格式化字符。程序清单 12.3 包含这些说明符的范例。

表 12.2 用于模式定义的格式说明符

说 明 符	描 述
0	零占位符，如果可能，则填充位
#	空占位符，如果可能，则填充位
.	显示一个句点，用作小数点
,	使用逗号将数字分组，也可用作倍增器（multiplier），见程序清单 12.3
%	将数字显示为百分比值（例如，1.00 被显示为 100%）
\	用于指示要打印的特殊字符。这可以是转移字符之一，如换行字符（\n）
'xyz'	显示单引号内的文本
"xyz"	显示双引号内的文本

程序清单 12.3 picts.cs：使用模式（picture）说明符

```

1: // picts.cs - Using picture specifiers
2: //-----
3:
4: using System;
5:
6: class myApp
7: {
8:     public static void Main()
9:     {
10:         int var1 = 1234;
11:         float var2 = 12.34F;
12:
13:         // Zero formatter
14:         Console.WriteLine("\nZero...");
15:         Console.WriteLine("{0} -->{0:00000000}", var1);
16:         Console.WriteLine("{0} -->{0:00000000}", var2);
17:     }

```

```

18:     // Space formatter
19:     Console.WriteLine("\nSpace...");
20:     Console.WriteLine("{0} -->{0:0###}<--", var1);
21:     Console.WriteLine("{0} -->{0:0###}<--", var2);
22:
23:     // Group separator and multiplier (,)
24:     Console.WriteLine("\nGroup Multiplier...");
25:     Console.WriteLine("{0} -->{0:0,,}<--", 1000000);
26:     Console.WriteLine("Group Separator...");
27:     Console.WriteLine("{0} -->{0:##,##,##0}<--", 2000000);
28:     Console.WriteLine("{0} -->{0:##,##,##0}<--", 3);
29:
30:     // Percentage formatter
31:     Console.WriteLine("\nPercentage...");
32:     Console.WriteLine("{0} -->{0:0%}<--", var1);
33:     Console.WriteLine("{0} -->{0:0%}<--", var2);
34:
35:     // Literal formatting
36:     Console.WriteLine("\nLiteral Formatting...");
37:     Console.WriteLine("{0} -->{0:'My Number: '0}<--", var1);
38:     Console.WriteLine("{0} -->{0:'My Number: '0}<--", var2);
39:     Console.WriteLine("\n{0} -->{0:Mine: 0}<--", var1);
40:     Console.WriteLine("{0} -->{0:Mine: 0}<--", var2);
41: }
42: }

```

该程序清单的输出如下:

```

Zero...
1234 -->0001234
12.34 -->0000012

Space...
1234 -->01234<--
12.34 -->00012<--

Group Multiplier...
1000000 -->1<--
Group Separator...
2000000 -->2,000,000<--
3 -->3<--

Percentage...
1234 -->123400%<--
12.34 -->1234%<--

Literal Formatting...
1234 -->My Number: 1234<--
12.34 -->My Number: 12<--

```

```
1234 -->Mine: 1234<--
12.34 -->Mine: 12<--
```

分析：该程序清单将表12.2列出的格式说明符用于两个变量。从注释和输出可以知道，这些说明符是如何起作用的。

12.2.1.8 格式化负数

有时候，需要以不同于正数的方式来处理负数。前面介绍的说明符同时适用于正数和负数。

可以将指定格式的占位符分为两部分或三部分。如果分为两部分，则前一部分用于正数和零，而后一部分用于负数。如果分为三部分，则第一部分用于正数，中间部分用于零，最后一部分用于负数。

使用分号将占位符分成多个部分，其中每部分都包含占位符编号。例如，对于正数，精度为3位，对于负数，精度为5位，对于0精度为1位，则可以使用下面的格式：

```
{0:D3;D5;'0'}
```

程序清单 12.4 包含这样的范例。

程序清单 12.4 threeways.cs

```
1: // threeway.cs - Controlling the formatting of numbers
2: //-----
3:
4: using System;
5:
6: class MyApp
7: {
8:     public static void Main()
9:     {
10:         Console.WriteLine("\nExample 1...");
11:         for ( int x = -100; x <= 100; x += 100 )
12:         {
13:             Console.WriteLine("{0:000;-00000;'0'}", x);
14:         }
15:
16:         Console.WriteLine("\nExample 2...");
17:         for ( int x = -100; x <= 100; x += 100 )
18:         {
19:             Console.WriteLine("{0:Pos: 0;Neg: -0;Zero}", x);
20:         }
21:
22:         Console.WriteLine("\nExample 3...");
23:         for ( int x = -100; x <= 100; x += 100 )
24:         {
25:             Console.WriteLine("{0:You Win!;You Lose!;You Broke Even!}", x);
26:         }
27:     }
28: }
```

该程序清单的输出如下：

```

Example 1...
-00100
0
100

Example 2...
Neg: -100
Zero
Pos: 100

Example 3...
You Lose!
You Broke Even!
You Win!

```

分析：该程序清单演示了如何将自定义格式分为三部分。它使用for循环来创建了一个负数，然后将其递增为0和正数。对于这三个值，可以使用同一个WriteLine来显示它们。这种操作执行了3次，完成三个不同的例子。

从第13行可以知道，对于正数，打印结果中将至少包含三位，因为第一部分包含三个0；对于负数，则打印结果中将包含负号，后面至少跟五个数字。这是因为格式中包含表示负号的短划线和5个零。如果值为零，则打印的将是0。

在第二个和第三个例子的格式中包含文本。它们之间的差别在于，第二个例子中还包含占位符0，以便打印实际的值；而第三个例子则没有，只打印文本。

从这三个例子可以知道，很容易根据变量的正负，提供不同的格式。

12.2.2 格式化日期和时间

日期和时间也可以被格式化。有很多说明符可用于格式化从星期几到完整的日期和时间字符串等。介绍这些格式字符之前，先来介绍如何获取日期和时间。

12.2.2.1 获取日期和时间

C#和.NET 框架提供了一个用于存储日期和时间的类——DateTime，它位于名称空间System中。DateTime类存储完整的日期和时间。

DateTime类有许多很有用的属性和方法，另外还有一些静态成员。您最有可能使用的两个静态属性是Now和Today。Now包含它被使用时的日期和时间，而Today返回当前的日期。由于它们是静态属性，因此可以通过类名（而不是实例名称）来取得它们的值：

```

DateTime.Now
DateTime.Today

```

有关所有方法和属性的信息，请参阅在线文档。下面是一些很有用的方法和属性：

- Date: 返回DateTime对象的日期部分；
- Month: 返回DateTime对象的月份部分；
- Day: 返回DateTime对象的天部分；
- Year: 返回DateTime对象的年份部分；
- DayOfWeek: 返回DateTime对象的星期部分（即星期几）；
- DayOfYear: 指出DateTime对象中日期属于该年份的多少天；

- `TimeOfDay`: 返回 `DateTime` 对象的时间部分;
- `Hour`: 返回 `DateTime` 对象的小时部分;
- `Minute`: 返回 `DateTime` 对象的分钟部分;
- `Second`: 返回 `DateTime` 对象的秒部分;
- `Millisecond`: 返回 `DateTime` 对象的毫秒部分;
- `Ticks`: 返回 `DateTime` 对象对应的 100 纳秒数。

注意: `DateTime.Today` 值提供当前的日期, 而不包含时间。

12.2.2.2 格式化日期和时间

用于日期、时间或两者的说明符很多。这包括以长格式或短格式显示信息。表 12.3 列出了这些日期和时间说明符。

表 12.3 日期和时间格式字符

说明符	描述	默认格式	输出范例
<code>d</code>	短日期	<code>mm/dd/yyyy</code>	5/6/2001
<code>D</code>	长日期	<code>day, month dd, yyyy</code>	Sunday, May 06, 2001
<code>f</code>	完整的日期短时间	<code>day, month dd, yyyy</code> <code>hh:mm AM/PM</code>	Sunday, May 06, 2001 12:30 PM
<code>F</code>	完整的日期完整的时间	<code>day, month dd, yyyy</code> <code>HH:mm:ss AM/PM</code>	Sunday, May 06, 2001 12:30:54 PM
<code>g</code>	短日期短时间	<code>mm/dd/yyyy HH:mm</code>	6/5/2001 12:30 PM
<code>G</code>	短日期长时间	<code>mm/dd/yyyy hh:mm:ss</code>	6/5/2001 12:30:54 PM
<code>M</code> 或 <code>m</code>	月份和天	<code>month dd</code>	May 06
<code>R</code> 或 <code>r</code>	RFC1123	<code>ddd, dd Month yyyy</code> <code>hh:mm:ss GMT</code>	Sun, 06 May 2001 12:30:54 GMT
<code>s</code>	合适的 (sortable)	<code>yyyy-mm-dd hh:mm:ss</code>	2001-05-06T12:30:54
<code>t</code>	短时间	<code>hh:mm AM/PM</code>	12:30 PM
<code>T</code>	长时间	<code>hh:mm:ss AM/PM</code>	12:30:54 PM
<code>u</code>	合适的 (通用的)	<code>yyyy-mm-dd hh:mm:ss</code>	2001-05-06 12:30:54Z
<code>U</code>	合适的 (通用的)	<code>Day, month dd, yy hh:mm:ss</code> <code>AM/PM</code>	Sunday, May 06, 2001 12:30:54 PM
<code>Y</code> 或 <code>y</code>	年份/月份	<code>month, yyyy</code>	May, 2001

警告：s 是一种用于打印合适日期的说明符，注意它是小写的。大写的 S 不是有效的说明符，使用它将引发异常。

日期和时间说明符使用起来很容易。程序清单 12.5 定义了一个日期变量，并以表 12.3 列出的各种格式打印该变量。

程序清单 12.5 dtformat.cs: 日期格式

```
1: // dtformat.cs - date/time formats
2: //-----
3:
4: using System;
5:
6: class MyApp
7: {
8:     public static void Main()
9:     {
10:         DateTime CurrTime = DateTime.Now;
11:
12:         Console.WriteLine("d: {0:d}", CurrTime );
13:         Console.WriteLine("D: {0:D}", CurrTime );
14:         Console.WriteLine("f: {0:f}", CurrTime );
15:         Console.WriteLine("F: {0:F}", CurrTime );
16:         Console.WriteLine("g: {0:g}", CurrTime );
17:         Console.WriteLine("G: {0:G}", CurrTime );
18:         Console.WriteLine("m: {0:m}", CurrTime );
19:         Console.WriteLine("M: {0:M}", CurrTime );
20:         Console.WriteLine("r: {0:r}", CurrTime );
21:         Console.WriteLine("R: {0:R}", CurrTime );
22:         Console.WriteLine("s: {0:s}", CurrTime );
23:         // Console.WriteLine("S: {0:S}", CurrTime ); // error!!!
24:         Console.WriteLine("t: {0:t}", CurrTime );
25:         Console.WriteLine("T: {0:T}", CurrTime );
26:         Console.WriteLine("u: {0:u}", CurrTime );
27:         Console.WriteLine("U: {0:U}", CurrTime );
28:         Console.WriteLine("y: {0:y}", CurrTime );
29:         Console.WriteLine("Y: {0:Y}", CurrTime );
30:     }
31: }
```

该程序清单的输出如下：

```
d: 5/6/2001
D: Sunday, May 06, 2001
f: Sunday, May 06, 2001 1:06 PM
F: Sunday, May 06, 2001 1:06:51 PM
g: 5/6/2001 1:06 PM
G: 5/6/2001 1:06:51 PM
m: May 06
M: May 06
r: Sun, 06 May 2001 13:06:51 GMT
```

```

R: Sun, 06 May 2001 13:06:51 GMT
s: 2001-05-06T13:06:51
t: 1:06 PM
T: 1:06:51 PM
u: 2001-05-06 13:06:51Z
U: Sunday, May 06, 2001 6:06:51 PM
y: May, 2001
Y: May, 2001

```

分析: 第10行声明了一个用于存储日期和时间的对象, 这是使用DateTime类实现的。该对象名为CurrentTime, 并将DateTime类的静态成员Now的值赋给它, Now提供当前的日期和时间。从输出可以知道, 作者是在五月份运行该程序清单的。第12~29行以各种格式打印该日期和时间。

第23行被注释掉, 它使用了非法的说明符S。如果将该行的注释标记删除, 该程序清单将引发异常。

12.2.3 显示枚举中的值

在第8天学习枚举时, 您发现, 使用Write和WriteLine输出枚举时, 显示的是枚举的描述性值, 而不是数字型值。通过格式化字符串, 可以控制输出是数字型值还是文本值, 还可以将其强制显示为十六进制。表12.4列出了用于枚举的格式化字符, 而程序清单12.6则使用了这些字符。

表 12.4 用于枚举的格式化字符

说 明 符	描 述
D或d	显示枚举元素的数值型值
G或g	显示枚举元素的字符串值
X或x	以十六进制格式显示枚举元素的数字型值

程序清单 12.6 enums.cs: 格式化枚举值

```

1: // enums.cs - enumerator formats
2: //-----
3:
4: using System;
5:
6: class MyApp
7: {
8:     enum Pet
9:     {
10:         Cat,
11:         Dog,
12:         Fish,
13:         Snake,
14:         Rat,
15:         Hamster,
16:         Bird
17:     }
18:

```

```
19: public static void Main()
20: {
21:     Pet myPet = Pet.Fish;
22:     Pet yourPet = Pet.Hamster;
23:
24:     Console.WriteLine("Using myPet: ");
25:     Console.WriteLine("d: {0:d}", myPet );
26:     Console.WriteLine("D: {0:D}", myPet );
27:     Console.WriteLine("g: {0:g}", myPet );
28:     Console.WriteLine("G: {0:G}", myPet );
29:     Console.WriteLine("x: {0:x}", myPet );
30:     Console.WriteLine("X: {0:X}", myPet );
31:
32:     Console.WriteLine("\nUsing yourPet: ");
33:     Console.WriteLine("d: {0:d}", yourPet );
34:     Console.WriteLine("D: {0:D}", yourPet );
35:     Console.WriteLine("g: {0:g}", yourPet );
36:     Console.WriteLine("G: {0:G}", yourPet );
37:     Console.WriteLine("x: {0:x}", yourPet );
38:     Console.WriteLine("X: {0:X}", yourPet );
39: }
40: }
```

该程序清单的输出如下:

```
Using myPet:
d: 2
D: 2
g: Fish
G: Fish
x: 00000002
X: 00000002

Using yourPet:
d: 5
D: 5
g: Hamster
G: Hamster
x: 00000005
X: 00000005
```

分析: 该程序清单声明了一个用于存储宠物的枚举。第21和22行创建了两个枚举对象myPet和yourPet, 用于说明格式说明符。第24~30行将说明符用于对象myPet; 而第32~38则用于yourPet。从输出可以知道, 说明符的大小写无关紧要。

12.3 使用字符串

介绍了关于格式化字符串的所有知识后, 我们回过头来更详细地介绍一些字符串方面的知识。

字符串是一种特殊的数据类型，可以存储文本信息。

您知道，`string` 是一个 C# 关键字，它指的是 `System` 名称空间中的 `String` 类，因此字符串具有 `String` 类的所有属性和方法。

存储在字符串中的值是不能修改的，当您调用字符串方法或修改字符串时，实际上将创建一个新的字符串。如果试图修改字符串中的字符，将发生错误。程序清单 12.7 是一个简单的例子，它证明了这一点。

程序清单 12.7 `str_err.cs`: 字符串不能被修改

```
1: // str_err.cs - Bad listing. Generates error
2: //-----
3: using System;
4:
5: class myApp
6: {
7:     public static void Main()
8:     {
9:         string str1 = "abcdefghijklmnop";
10:
11:         str1[5] = 'X';    // ERROR!!!
12:
13:         Console.WriteLine( str1 );
14:     }
15: }
```

编译该程序清单时，将出现如下所示的错误：

```
str_err.cs(11,6): error CS0200: Property or indexer 'string.this[int]' cannot be assigned
to - it is read only
```

分析：该程序清单证明了您不能修改字符串。字符串是一个由字符组成的数组。第6行试图将第6个字符改为大写的字母X。这将导致错误，因为您不能修改字符串的值。

您可能认为，如果字符串不能修改，则其用途将很有限。另外您可能会问，如何将方法和属性用于字符串。您将发现，修改字符串的方法实际上将创建一个新的字符串。另外，如果您确实需要修改字符串，则可以使用 C# 提供的另一个类：`StringBuilder`，这将在今天课程的后面介绍。

注意：字符串被认为是永远不变的，这意味着不能修改。

12.3.1 字符串方法

有很多非常有用的方法可用于字符串，表 12.5 列出了一些重要的字符串方法，并对其用途进行了描述。

表 12.5 常用的字符串方法

方 法	描 述
静 态 方 法	
<code>Compare</code>	比较两个字符串的值

第 12 天课程 输入和输出

续表

方 法	描 述
静 态 方 法	
CompareOrdinal	比较两个字符串的值,但不根据语言或其他国际化问题进行校正
Concat	将两个或更多的字符串合并成一个
Copy	使用已有的字符串创建一个新的字符串
Equals	比较两个字符串,以确定它们是否包含相同的值。如果是,则返回 true; 否则返回 false
Format	使用相应的字符串值替换格式说明符,说明符在今天课程的前面介绍过了
Join	将两个或更多的字符串合并为一个,在原来的字符串之间加上指定的“分隔符字符串”
每个实例都有的方法和属性	
Char	返回指定位置处的字符
Clone	返回存储的字符串的拷贝
CompareTo	将当前字符串同另一个字符串进行比较。如果当前字符串更小,则返回一个负数;如果等于,则返回 0; 如果更大,则返回一个正数
CopyTo	将字符串的一部分或全部复制到一个新的字符串或字符数组中
EndsWith	确定字符串是否以某个字符或字符串结尾,如果是则返回 true, 否则返回 false
Equals	比较两个字符串,以确定它们是否包含相同的值。如果是,则返回 true; 否则返回 false
IndexOf	返回字符串中第一次出现某个字符或字符串的索引(位置),如果没有这样的字符或字符串,则返回-1
Insert	将一个值插入到字符串中,这是通过返回一个新的字符串来实现的
LastIndexOf	返回字符串中最后一次出现某个字符或字符串的索引(位置),如果没有这样的字符或字符串,则返回-1
Length	返回字符串的长度,长度等于字符串包含的字符数
PadLeft	将字符串右对齐,并在左边填充指定的字符(或空格)
PadRight	将字符串左对齐,并在右边填充指定的字符(或空格)
Remove	从字符串的指定位置开始删除指定数目的字符
Split	执行的操作与 Join 相反,即根据指定的值,将字符串分成几个子串,指定的值被用作分拆点
StartsWith	确定字符串是否以某个字符串或字符串打头,如果是则返回 true, 否则返回 false
Substring	从字符串中指定位置开始返回一个子串,也可以指定返回多少个字符,但不是必须的
ToCharArray	将当前字符串中的字符复制到一个 char 数组中
ToLower	返回当前字符串的小写

续表

方 法	描 述
每个实例都有的方法和属性	
ToUpper	返回当前字符串的大写
Trim	从当前字符串的开头和结尾删除指定的字符串
TrimEnd	从当前字符串的结尾删除指定的字符串
TrimStart	从当前字符串的开头删除指定的字符串

在本书余下的内容中，将使用其中的很多方法和属性。

12.3.2 特殊的字符串格式符——@

前面的程序清单中使用了很多特殊字符。例如，要在字符串中使用双引号，可以使用转移字符。下面的代码打印“Hello World”（包括双引号）：

```
System.Console.WriteLine("\"Hello World\"");
```

要在字符串中包含反斜杠，也必须使用转移字符：

```
System.Console.WriteLine("My Code: C:\\Books\\TyCSharp\\Originals\\");
```

C#提供了一个特殊的格式字符@，用于省略转移字符。当该字符位于字符串的前面时，字符串的值将被看作是字面值。事实上，这种字符串被称为“逐字字符串字面值”（verbatim string literal）。下面的语句与前面打印目录的语句等效：

```
System.Console.WriteLine(@"My Code: C:\Books\TYCSharp\Original");
```

使用字符串格式符@时，您将发现，唯一需要注意的地方是双引号。如果在使用@格式化的字符串中使用双引号，则需要使用两个双引号。下述代码与打印“Hello World”的范例等效：

```
System.Console.WriteLine(@"""Hello World""");
```

注意：您可能在想，不是说使用两对双引号吗，为何使用了三对双引号？这是因为其中的一对用于将字符串括起。在这个例子中，第一个双引号表示字符串的开始位置，但由于它后面跟了一个双引号，因此系统知道需要显示一个双引号。第四个双引号本来表示字符串的结束位置，但由于它后面还有双引号，因此被转换为显示一个双引号。第六个双引号检查是否结束字符串，由于它后面没有跟双引号，因此字符串至此结束。

12.3.3 创建字符串

名称空间 System.Text 中的 StringBuilder 类用于创建用来存储可修改字符串的对象。使用 StringBuilder 类创建的对象的工作与字符串类似，差别在于 StringBuilder 的方法可直接操纵存储的值。表 12.6 列出了 StringBuilder 对象的方法和属性。

表 12.6 **StringBuilder 的方法和属性**

方法或属性	描 述
Append	将对象附加到当前 StringBuilder 的后面
AppendFormat	根据格式说明符将对象插入到字符串中
Capacity	设置或取得可以存储的字符数，Capacity 可以增加至等于 MaxCapacity
Chars	使用索引器表示法取得或设置给定索引位置的字符
EnsureCapacity	确保 StringBuilder 的容量至少大于指定的值。如果给 EnsureCapacity 传递了值，则 Capacity 属性将被设置为该值。如果 MaxCapacity 的值小于传递的值，将发生异常
Equals	确定当前 StringBuilder 是否等于传递的值
Insert	将对象放置到 StringBuilder 中的指定位置
Length	设置或取得当前存储在 StringBuilder 中的长度值。Length 不能大于 StringBuilder 的 Capacity。如果当前值小于 Length，则将截尾
MaxCapacity	取得 StringBuilder 的最大容量
Remove	从当前 StringBuilder 对象的指定位置开始，删除指定数目的字符。
Replace	将指定字符替换为新的字符
ToString	将 StringBuilder 转换为一个字符串

StringBuilder 类的使用方法与其他类相似。程序清单 12.8 使用了 StringBuilder 对象以及表 12.6 中的几个方法和属性。该程序清单让用户输入姓、名和中名，然后将它们组合成一个 StringBuilder 对象。

程序清单 12.8 **strbuilder.cs: 使用 StringBuilder 类**

```

1: // strbuilder.cs - String Builder
2: //-----
3: using System;
4: using System.Text;    //For StringBuilder
5:
6: class buildName
7: {
8:     public static void Main()
9:     {
10:         StringBuilder name = new StringBuilder();
11:         string buffer;
12:         int marker = 0;
13:
14:         Console.Write("\nEnter your first name: ");
15:         buffer = Console.ReadLine();
16:
17:         if ( buffer != null )
18:         {

```

```
19:     name.Append(buffer);
20:     marker = name.Length;
21: }
22:
23: Console.Write("\nEnter your last name: ");
24: buffer = Console.ReadLine();
25:
26: if ( buffer != null )
27: {
28:     name.Append(" ");
29:     name.Append(buffer);
30: }
31:
32: Console.Write("\nEnter your middle name: ");
33: buffer = Console.ReadLine();
34:
35: if ( buffer != null )
36: {
37:     name.Insert(marker+1, buffer);
38:     name.Insert(marker+buffer.Length+1, " ");
39: }
40:
41: Console.WriteLine("\n\nFull name: {0}", name);
42:
43: // Some stats....
44: Console.WriteLine("\n\nInfo about StringBuilder string:");
45: Console.WriteLine("value: {0}", name);
46: Console.WriteLine("Capacity: {0}", name.Capacity);
47: Console.WriteLine("Maximum Capacity: {0}", name.MaxCapacity);
48: Console.WriteLine("Length: {0}", name.Length);
49: }
50: }
```

该程序清单的输出如下:

Enter your first name: Bradley

Enter your last name: Jones

Enter your middle name: Lee

Full name: Bradley Lee Jones

Info about StringBuilder string:

value: Bradley Lee Jones

Capacity: 32

Maximum Capacity: 2147483647

Length: 17

分析：首先需要注意的是，第4行使用using语句将名称空间System.Text包含进来。没有这条语句，将需要以全限定名称System.Text.StringBuilder来引用StringBuilder类。

第10行创建了一个名为name的StringBuilder对象，用于存储该程序清单将创建的字符串。第11行创建了一个名为buffer的字符串，用于取得用户输入的信息。这种信息是在第15、24和33行使用Console类的ReadLine方法来读取的。首先读取的是姓，它被追加到StringBuilder对象name中，由于该对象为空，因此姓被放在开始位置。姓的长度被赋给变量marker，该变量被用来确定放置中名的位置。

接着取得的是名，它追加到name对象中（第29行），前面加入了一个空格（第28行）。最后，获得中名，并使用Insert方法将其插入到name对象的中间。前面保存的marker被用来确定插入中名的位置。

第41行显示得到的全名。第44~48行显示一些通用的信息。第45行打印了StringBuilder对象name的值；第46行打印name对象当前的容量；第47行打印最大容量；第48行打印name对象当前存储的值的长度。

12.4 从控制台获取信息

到目前为止，今天的课程主要介绍了如何格式化和显示信息。除了生成输出外，您还需要在获取输入（输入给程序的信息）方面有更灵活的方法。

本书前面使用过Console类的Read和ReadLine方法，接下来的几节将更详细地介绍如何获取输入并将其转换为更有用处的格式。

12.4.1 使用Read方法

System名称空间中的Console类包含两个可用于从用户那里获取信息的方法：ReadLine和Read。

Read方法每次从输入流中读取一个字符，并将该字符作为int值返回。如果读到了流的末尾，则返回-1。程序清单12.9使用Read方法来读取字符。

提示：要结束来自控制台的字符流，可以按Ctrl + z键。

程序清单 12.9 Read 方法

```
1: // readit.cs - Read information from Console
2: //-----
3: using System;
4: using System.Text;
5:
6: class MyApp
7: {
8:     public static void Main()
9:     {
10:         StringBuilder Input = new StringBuilder();
11:
12:         int ival;
13:         char ch = ' ';
14:
15:         Console.WriteLine("Enter text. When done, press CTRL+Z:");
16:
```

```

17:     while ( true )
18:     {
19:         ival = Console.Read();
20:         if ( ival == - 1 )
21:             break;
22:         ch = ( char ) ival;
23:
24:         Input.Append(ch);
25:     }
26:     Console.WriteLine("\n\n=====>\n");
27:     Console.Write( Input );
28:     Console.Write("\n\n");
29: }
30: }

```

该程序清单的输出如下:

```

Enter text. When done, press CTRL+Z:
Mary Had a little lamb,
Its fleece was white as snow.
Everywhere that Mary went,
that lamb was sure to go!
=====>
Mary Had a little lamb,
Its fleece was white as snow.
Everywhere that Mary went,
that lamb was sure to go!

```

分析: 从输出可以知道, 共输入了四行文本。输入完文本后, 按Ctrl + Z来结束输入。

第17~25行使用了一个While循环来读取字符。实际的读取工作是在第19行进行的。读取的值被赋给变量ival。如果ival等于-1, 则表明到达了输入的末尾, 因此使用break命令来退出循环。如果ival的值有效, 则将其强制转换为字符值, 如第22行所示。然后, 将其追加到字符串Input中。

完成字符的输入后, 第27行打印字符串Input。所有的字符都被存储到Input中, 包括回车和换行符。

注意: Windows和Web应用程序获取信息的方法与此不同。第15~17天将更详细地介绍在这些平台上如何获取信息。

12.4.2 使用 ReadLine 方法

本书前面已多次使用过ReadLine方法, 您应该知道其功能。ReadLine方法读取回车、换行符或两者之前的一行文本。如果达到流的末尾, 则返回null。

下面的程序清单与12.9类似。要结束该程序清单的运行, 可按Ctrl + z键。

程序清单 12.10 readln.cs: 使用 ReadLine 来读取字符

```

1: // readln.cs - Read information from Console
2: //-----
3: using System;
4: using System.Text;
5:

```

```

6: class myApp
7: {
8:     public static void Main()
9:     {
10:         StringBuilder Input = new StringBuilder();
11:         string buff;
12:
13:         Console.WriteLine("Enter text. When done, press Ctrl+Z:");
14:
15:         while ( (buff = Console.ReadLine()) != null )
16:         {
17:             Input.Append(buff);
18:             Input.Append("\n");
19:         }
20:         Console.WriteLine("\n\n=====>\n");
21:         Console.Write( Input );
22:         Console.Write("\n\n");
23:     }
24: }

```

该程序清单的输出如下：

Enter text. When done, press Ctrl+Z:

Twinkle, twinkle little star

How I wonder where you are

up above the sky so high

like a diamond in the sky

=====>

Twinkle, twinkle little star

How I wonder where you are

up above the sky so high

like a diamond in the sky

分析：该程序清单的功能与12.9相同，但它一次读取一行字符（第15行），而不是一个。如果读取的行为null，则输入结束。如果读取到了信息，则将其追加到Input字符串中（第17行）。由于读取的信息中不包括换行符，因此第18行在创建的字符串中加入了换行符，以便第21行的输出与输入的内容相同。

如果禁止使用 Ctrl + Z 呢？假设您希望输入一个空行来结束该程序。今天课程的练习 3 将要求您修改该程序清单，以便输入空行来结束程序。

12.4.3 Convert 类

使用 Read 和 ReadLine 方法读取信息的关键不是取得信息，而是将其转换为想要的格式。这可以是对字符串中的文本进行过滤，得到一个不同的字符串，或将其转换为一种不同的数据类型。

System 名称空间中有一个可用于将数据转换为其他数据类型的类：Convert 类。

Convert 类是一个被密封的类，包含大量的静态方法。这些方法可将数据转换为不同的数据类型

型。由于这些方法是静态的，因此使用格式如下：

```
Convert.method( Orig_val);
```

这里假设您已经使用 `using` 语句将名称空间 `System` 包含进来了。其中，`method` 是要使用的转换方法的名称；`Orig_val` 是要转换为新类型的值。这个类位于基类库中，知道这一点非常重要。这意味着可在其他编程语言中使用这个类。`Convert` 类的方法将数据转换为基数据类型，而不是 C# 数据类型。但不用着急，第 3 天的课程介绍过，每种 C# 数据类型都有相应的基类型。

表 12.7 列出了 `Convert` 类的一些方法，完整的方法列表请参阅 .NET 框架文档。程序清单 12.11 是一个使用 `Convert` 类方法的简短范例。该程序清单将 `ReadLine` 读取的字符串值转换为一个整数。

表 12.7 转换数据类型的方法

方 法	目 标 类 型
<code>ToBoolean</code>	布尔型
<code>ToByte</code>	8 位的无符号整数
<code>ToChar</code>	Unicode 字符
<code>ToDateTime</code>	<code>DateTime</code>
<code>ToDecimal</code>	<code>Decimal</code>
<code>ToDouble</code>	双精度数
<code>ToInt16</code>	16 位的符号整数
<code>ToInt32</code>	32 位的符号整数
<code>ToInt64</code>	64 位的符号整数
<code>ToSByte</code>	8 位的符号整数
<code>ToSingle</code>	单精度浮点数
<code>ToString</code>	字符串
<code>ToUInt16</code>	16 位的无符号整数
<code>ToUInt32</code>	32 位的无符号整数
<code>ToUInt64</code>	64 位的无符号整数

程序清单 12.11 conv.cs: 使用 `Convert` 的方法

```
1: // conv.cs - Converting to a data type
2: //-----
3: using System;
4: using System.Text;
5:
6: class myApp
```

```
7: /
8: public static void Main()
9: {
10:     string buff;
11:     int age;
12:
13:     Console.Write("Enter your age: ");
14:
15:     buff = Console.ReadLine();
16:
17:     try
18:     {
19:         age = Convert.ToInt32(buff);
20:
21:         if( age < 21 )
22:             Console.WriteLine("You are under 21.");
23:         else
24:             Console.Write("You are 21 or older.");
25:     }
26:     catch( ArgumentException )
27:     {
28:         Console.WriteLine("No value was entered... (equal to null)");
29:     }
30:     catch( OverflowException )
31:     {
32:         Console.WriteLine("You entered a number that is too big or too small.");
33:     }
34:     catch( FormatException )
35:     {
36:         Console.WriteLine("You didn't enter a valid number.");
37:     }
38:     catch( Exception e )
39:     {
40:         Console.WriteLine("Something went wrong with the conversion.");
41:         throw;
42:     }
43: }
44: }
```

下面是在命令行执行该程序清单多次得到的输出:

```
C:\Day12>conv
Enter your age: 12
You are under 21.
```

```
C:\Day12>conv
Enter your age: 21
You are 21 or older.
```

```
C:\Day12>conv
```

```
Enter your age: 65
You are 21 or older.

C:\Day12>conv
Enter your age: 9999999999999999
You entered a number that is too big or too small.

C:\\Day12>conv
Enter your age: abc
You didn't enter a valid number.

C:\Day12>conv
Enter your age: abc123
You didn't enter a valid number.

C:\Day12>conv
Enter your age: 123abc
You didn't enter a valid number.

C:\Day12>conv
Enter your age: 123 123
You didn't enter a valid number.
```

分析：您首先注意到的是，该程序清单使用了异常处理。大凡可能发生异常时，都应该使用异常处理。如果输入的信息不合适，则第19行使用的转换方法 `ToInt32` 将可能引发异常。第26、30和34行捕获三种不同的异常，第38行捕获预料之外的其他异常。如果没有出现异常，则根据年龄是否小于21岁，显示一条消息。

该程序清单让您知道如何从终端用户那里获取信息、将其转换为更有用的格式、并验证它以确保其合法。

12.5 总 结

今天的课程介绍了大量的内容。请给包含方法表的页做上标记，以便您在后面编程时参考。

今天介绍了如何格式化信息，使之更好看。除了学习如何格式化常规数据类型外，您还学习了如何取得并格式化日期和时间。

今天课程的后半部分重点介绍了字符串以及可用于操纵它们的方法。由于字符串是永远不变的——不可修改，因此还介绍了 `StringBuilder` 类。通过这个类，您学习了如何操纵字符串信息。

最后，重点介绍了如何从控制台获取信息。您温习了 `Read` 和 `ReadLine` 方法，并将其与学到的格式化字符串和转换类 `Convert` 结合起来使用。现在，您知道如何从控制台读取信息，并将其转换为有用的格式（包括不同的数据类型）。

12.6 问与答

问：您说字符串不能被修改，而很多字符串方法看起来却修改了字符串，原因何在？

答：直接操纵字符串的方法并不修改原来的字符串，而是创建一个修改后的新字符串，并用它替换原来的字符串。对于 `StringBuilder` 类，则可以操纵原来的字符串。

问：您说 `Convert` 类处理的是基数据类型，我不明白您的意思，请解释。

答：您应该复习第 3 天的课程。当您编译程序时，C#数据类型将被转换为运行阶段环境中的数据类型。例如 C#类型 `int` 将被转换为 `System.Int32`。实际上，在程序中，`int` 和 `System.Int32` 可以互换

12.7 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

12.7.1 小测验

1. 哪个方法可用于将数据类型 `int` 转换为字符串，并进行格式化？
2. `string` 类中的哪个方法可用于将信息格式化为一个新的字符串？
3. 哪个说明符可用于指示将数字格式化为金额？
4. 哪个说明符可用于指示将数字格式化为包含逗号和一位小数（如 123,890.5）？
5. 如果 `x` 的值为 123456789.876，则下述代码的输出是什么？

```
Console.WriteLine("X is {0:'a value of '#, .#'. '}", x);
```

6. 在下述情况下，应使用什么样的说明符：如果为正数，则将其显示为至少包含 5 位的 decimal 数；如果为负，则至少包含 8 位；如果为 0，则显示文本<empty>。
7. 如何取得当前的日期？
8. 字符串和 `StringBuilder` 对象之间的关键区别在哪里？
9. 哪个特殊字符用作字符串格式符，它对字符串有何影响？
10. 如何从字符串得到数字型值？

12.7.2 练习

1. 编写一行代码，将数字格式化为至少包含三位和两位小数。
2. 编写以“星期几，月份，日和四位年份”格式（如 Monday, January 1, 2002）打印日期值的代码。
3. 修改程序清单 12.10，以便程序在用户输入空行后结束。
4. 修改程序清单 12.11。如果用户输入的信息中包含空格，则对信息进行裁剪，只是用空格前面的部分，并对其进行转换。例如，如果用户输入 123 456，则只使用 123。
5. 选做题：编写一个这样的程序：让用户输入其全名、年龄和电话号码，以特定的格式显示这些信息，并显示用户的姓名的首字母。

第 13 天课程

接 口

今天，将加深您对面向对象编程的理解，这包括通过使用接口来扩大继承和多态的用途。您将学习如何从多个来源继承特征以及如何对大量不同的数据类型执行方法。具体地说，今天将学习以下内容：

- 接口简介；
- 接口的基本结构；
- 定义和使用接口；
- 实现多个接口；
- 使用已有的接口派生出新的接口；
- 如何对类隐藏接口。

第 6 天介绍了关于类的知识，第 11 天介绍了如何使用一个类派生出另一个类。今天将介绍如何从多个来源继承特征，派生出新的类。

13.1 接 口

请看下面的类，这个类有何特点呢？

```
public abstract class cShape
{
    public abstract long Area();
    public abstract long Circumference();
    public abstract int sides();
}
```

上述类是一个抽象类，其所有的方法都是抽象的。抽象类已经在第 11 天介绍过了。

抽象类指的是至少包含一个抽象方法的类。而抽象方法指的是被继承时，必须覆盖的方法。

接口是另一种类似于类的引用类型，它与上面的 `cShape` 类极其类似。接口的用途是定义要声明的类中将包含什么，但不定义实际的功能。接口类似于抽象方法，与上面的类极其类似。

事实上，通过删除类和方法的限定符，并把关键字 `class` 改为 `interface`，可将 `cShape` 类改为接口：

```
public interface IShape
```



```

{
    long Area();
    long Circumference();
    int sides();
}

```

13.1.1 类和接口之比较

接口类似于纯粹的抽象类，它与类之间的差别很多。

首先（也是最主要的），接口不提供任何实现代码，这些代码是由实现接口提供的。接口提供关于将发生的情况的规范或指南，但不提供细节。

接口不同于类，其所有的成员都被视为公有的。如果试图为接口的成员声明一个不同的作用域限定符，将出错。

接口只包含方法、属性、事件和索引器（indexer），而不包含数据成员、构造函数和析构函数，也不能包含静态成员。

抽象类也具有上述特点，但在功能上不同于接口。阅读完今天的课程后，您便会理解这一点。

13.1.2 使用接口

接口的功能好像没有类那么强大，但它具备一些类无法实现的功能。类只能从另一个类派生而来，但类可以实现多个接口。另外，结构不能继承另一个结构或方法，但可以实现接口。后面将更详细地介绍如何实现多个接口。

注意：C#不像C++和其他面向对象语言那样支持多重继承。由于多重继承复杂的实现所带来的麻烦，多重继承被有意地删除了。C#通过允许实现多个接口提供了多重继承的功能和好处。

13.1.3 为何使用接口

使用接口有好处，从前几节可以知道一些。

首先，可以将接口作为一种给结构提供继承特性的途径。另外，可以在一个类中实现多个接口，从而获得抽象类无法获得的功能。

使用接口最大的价值之一是，可以给类添加通过其他方法无法实现的特征。如果将同样的特征添加到其他类中，便可以对其具备的功能做出假设。实际上，通过使用类，可以避免做这样的假设。

使用接口而不是类带来的另一个好处是，强制新的类实现接口定义的所有特征。如果继承带虚拟成员的基类，则可能不为虚拟成员提供实现代码。这使得新类和使用新类的程序可能出错。

13.2 定义接口

前面介绍过，接口是关于类需要实现的功能的指南，其基本结构如下：

```

interface IName
{
    members;
}

```

其中 IName 是接口的名称。可以为接口指定任何名称，但作者建议以 I 打头，以表明它是一个接口，这与大多数程序员的通常做法一致。

`members` 是接口的成员，可以是前面指出的任何类型：属性、事件、方法（虚拟方法）和索引器。

事件和索引器将在明天介绍。指定方法时，无需使用作用域限定符，正如前面指出的，方法将被视为公有的。

另外不需要实现方法体。大多数情况下，只需指明返回类型和方法名，后面加上括号和分号：

```
interface IFormatForPrint
{
    void FormatForPrint(PrintClass PrinterType);
    int  NotifyPrintComplete();
}
```

上述接口定义了两个方法：`FormatForPrint` 和 `NotifyPrintComplete`。前者接受一个 `PrintClass` 对象，但不返回任何值；后者返回一个整数。实现该接口的类必须实现这两个方法。

13.2.1 定义带方法成员的接口

了解足够的理论后，我们来看一些代码。程序清单 13.1 定义了一个名为 `IShape` 的接口，然后两个类（`Circle` 和 `Square`）实现了该接口。

接口 `IShape` 的定义如下：

```
public interface IShape
{
    double Area();
    double Circumference();
    int    Sides();
}
```

通过使用该接口，您保证了两件事。首先，`Circle` 和 `Square` 将完整地实现该接口的方法。这里是保证这两个类都将包含方法 `Area`、`Sides` 和 `Circumference` 的实现。同样重要的是，您还确保这两个类都将具备 `IShape` 的特征。第 12 天已经介绍过这一点的价值所在，而程序清单 13.1 再次说明了这种价值。

程序清单 13.1 `shape.cs`：使用 `IShape` 接口

```
1: // shape.cs -
2: //-----
3:
4: using System;
5:
6: public interface IShape
7: {
8:     double Area();
9:     double Circumference();
10:    int    Sides();
11: }
12:
13: public class Circle : IShape
14: {
15:     public int x;
```

```
16: public int y;
17: public double radius;
18: private const float PI = 3.14159F;
19:
20: public double Area()
21: {
22:     double theArea;
23:     theArea = PI * radius * radius;
24:     return theArea;
25: }
26:
27: public double Circumference()
28: {
29:     return ((double) (2 * PI * radius));
30: }
31:
32: public int Sides()
33: {
34:     return 1;
35: }
36:
37: public Circle()
38: {
39:     x = 0;
40:     y = 0;
41:     radius = 0.0;
42: }
43: }
44:
45: public class Square : IShape
46: {
47:     public int side;
48:
49:     public double Area()
50:     {
51:         return ((double) (side * side));
52:     }
53:
54:     public double Circumference()
55:     {
56:         return ((double) (4 * side));
57:     }
58:
59:     public int Sides()
60:     {
61:         return 4;
62:     }
63:
64:     public Square()
65:     {
```

```

66:     side = 0;
67: }
68: }
69:
70: public class MyApp
71: {
72:     public static void Main()
73:     {
74:         Circle myCircle = new Circle();
75:         myCircle.radius = 5;
76:
77:         Square mySquare = new Square();
78:         mySquare.side = 4;
79:
80:         Console.WriteLine("Displaying Circle information:");
81:         displayInfo(myCircle);
82:
83:         Console.WriteLine("\nDisplaying Square information:");
84:         displayInfo(mySquare);
85:     }
86:
87:     static void displayInfo( IShape myShape )
88:     {
89:         Console.WriteLine("Area: {0}", myShape.Area());
90:         Console.WriteLine("Sides: {0}", myShape.Sides());
91:         Console.WriteLine("Circumference: {0}", myShape.Circumference());
92:     }
93: }

```

该程序清单的输出如下:

```

Displaying Circle information:
Area: 78.5397529602051
Sides: 1
Circumference: 31.415901184082

```

```

Displaying Square information:
Area: 16
Sides: 4
Circumference: 16

```

分析: 该程序清单的第6~11行定义了接口 `IShape`, 该接口与前面介绍的完全相同。从第13行, 您知道接口是如何被实现的:

```
public class Circle : IShape
```

实现接口的方式与继承类相同——将其放在新类名的后面, 中间用冒号隔开。

`Circle` 类包含很多数据成员以及 `IShape` 中各方法的代码。`Circle` 类定义了 `IShape` 接口中的所有三个方法, 每个方法的参数类型和返回类型都与接口中的方法名相同。

第 45 行的 `Square` 类也是如此, 它也实现了 `IShape` 接口, 因此也包含该接口中三个方法的定义。

从第 70 行开始是应用程序类，其中的 Main 方法创建了一个 Circle 对象和一个 Square 对象，并给它们赋值。第 81 行调用了 displayInfo 方法，并将 Circle 对象 myCircle 传递给它；第 84 行再次调用了该方法，但传递的是 Square 对象 mySquare。

该方法被重载为能够接受对象 Circle 和 Square 吗？不是！第 11 天介绍的是类，而这里针对的是接口。displayInfo 方法接受一个 IShape 值。从技术上讲，并没有 IShape 这样的对象，但有具备 IShape 特征的对象。该方法是多态的，它能够接受任何实现了 IShape 接口的对象，因此能够使用 IShape 接口中定义的方法。

注意：您也可以在 displayInfo 方法中使用关键字 is 和 as 来确定是否可以使用不同的类方法。有关如何使用这些关键字，请参阅第 11 天的课程。

13.2.2 在接口中指定属性

接口中也可以包含关于属性的规范。和其他接口成员一样，接口中也不包含属性的具体实现代码。在接口中声明属性的格式如下：

```
modifier(s) datatype name
{
    get;
    set;
}
```

程序清单 13.2 演示了如何在接口中定义属性，然后如何在类中使用该属性。该程序清单并没有什么使用价值，只是说明了这一概念而已。

程序清单 13.2 props.cs：在接口中定义属性

```
1: // props.cs - Using properties in an interface
2: //-----
3:
4: using System;
5:
6: public interface IShape
7: {
8:     int Sides
9:     {
10:         get;
11:         set;
12:     }
13:
14:     double Area();
15: }
16:
17: public class Square : IShape
18: {
19:     private int sides;
20:     public int SideLengthn;
21:
22:     public double Area()
23:     {
```

```
24:     return ((double) (SideLength * SideLength));
25: }
26:
27: public int Sides
28: {
29:     get { return sides; }
30:     set { sides = value; }
31: }
32:
33: public Square()
34: {
35:     Sides = 4;
36: }
37: }
38:
39: public class MyApp
40: {
41:     public static void Main()
42:     {
43:         Square mySquare = new Square();
44:         mySquare.SideLength = 5;
45:
46:         Console.WriteLine("\nDisplaying Square information:");
47:         Console.WriteLine("Area: {0}", mySquare.Area());
48:         Console.WriteLine("Sides: {0}", mySquare.Sides);
49:     }
50: }
```

该程序清单的输出如下:

```
Displaying Square information:
Area: 25
Sides: 4
```

警告: 别忘了, C#是区分大小写的。使用名称相同, 但字母的大小写不同的变量是完全合法的, 虽然令人迷惑。在这个程序清单中, 一个例子是使用side和Side, 将属性名设置为与对应的数据名相同, 但大小写不同, 是一种常见的做法。

分析: 该程序清单的重点在于使用属性, 而不是前一个程序清单中的其他代码。现在是通过属性而不是方法来访问形状的边数。第8~12行的IShape接口中包含对属性Sides的声明, 其数据类型为int。该属性包含get和set方法, 但需要注意的是, 并不要求在接口中全部指定它们, 而可以毋庸置疑地只指定get或set。如果在接口中指定了get和set, 则实现该接口的类必须实现这两种方法。

注意: 对于这里的IShape接口, 可以只指定Sides的get方法。对于要设置边数的形状, 可以在构造函数中设置, 其值永远不变。get方法仍可以使用。即使接口中没有包含set方法, 类仍可以实现该方法。

从第17行开始是 Square 类中对 IShape 接口的实现。get 和 set 属性的定义位于第 27~31 行。Square 类的实现代码很简单, Sides 属性设置了数据成员 sides 的值。

对于通过接口实现的属性, 其用法和其他属性完全相同。该程序清单的多行中使用了 Sides 属

性，这包括第 48 行，它取得该属性的值。该属性的值是在构造函数（第 35 行）中设置的。

注意：许多人仍然这么说，即某个类继承了某个接口。从某种意义上说，确实如此，但更准确地说，应该是类实现了接口。

13.3 使用多个接口

实现接口而不是继承类的好处之一是可以同时实现多个接口。这让您能够实现多重继承，同时避免了一些缺点。

要实现多个接口，需要将这些接口用逗号分开。例如，要在 `Square` 类中包含接口 `IShape` 和 `IShapeDisplay`，可以使用下面的代码：

```
class Square : IShape, IShapeDisplay
{
    ...
}
```

然后，需要实现这两个接口的所有元素。程序清单 13.3 演示了如何使用多个接口。

程序清单 13.3 multi.cs：在同一个类中实现多个接口

```
1: // multi.cs -
2: //-----
3:
4: using System;
5:
6: public interface IShape
7: {
8:     // Cut out other methods to simplify example.
9:     double Area();
10:    int Sides { get; }
11: }
12:
13: public interface IShapeDisplay
14: {
15:     void Display();
16: }
17:
18: public class Square : IShape, IShapeDisplay
19: {
20:     private int sides;
21:     public int SideLength;
22:
23:     public int Sides
24:     {
25:         get { return sides; }
26:     }
27:
28:     public double Area()
29:     {
```

```
30:     return ((double) (SideLength * SideLength));
31: }
32:
33: public double Circumference()
34: {
35:     return ((double) (Sides * SideLength));
36: }
37:
38: public Square()
39: {
40:     sides = 4;
41: }
42:
43: public void Display()
44: {
45:     Console.WriteLine("\nDisplaying Square information:");
46:     Console.WriteLine("Side length: {0}", this.SideLength);
47:     Console.WriteLine("Sides: {0}", this.Sides);
48:     Console.WriteLine("Area: {0}", this.Area());
49: }
50: }
51:
52: public class MyApp
53: {
54:     public static void Main()
55:     {
56:         Square mySquare = new Square();
57:         mySquare.SideLength = 7;
58:
59:         mySquare.Display();
60:     }
61: }
```

该程序清单的输出如下:

```
Displaying Square information:
Side length: 7
Sides: 4
Area: 49
```

分析: 该程序清单声明并使用了两个接口。从第18行可以知道, Square类将实现两个接口。由于该类包含这两个接口, 因此必须实现这两个接口的所有成员。从第23~49行的代码可以知道, 所有的成员都实现了。

13.4 显式接口成员

至此, 实现接口一切都很顺利。当实现这样的接口——其成员的名称与已有名称发生冲突时, 将出现什么样的情况呢? 例如, 如果程序清单 13.3 中的两个接口都包含名为 Display 的方法, 将出

现什么情况呢?

如果类包含两个或更多的接口,而这些接口包含名称相同的成员,则该成员只需实现一次。该方法实现将满足各个接口的需要。

有时候,您可能想分别为两个接口实现方法。在这种情况下,需要使用显式接口实现。显式接口实现是通过在定义成员时包含接口名和成员名来实现的。在调用方法时,也必须进行强制转换,如程序清单 13.4 所示。

程序清单 13.4 explicit.cs

```

1: // explicit.cs -
2: //-----
3:
4: using System;
5:
6: public interface IShape
7: {
8:     double Area();
9:     int Sides { get; }
10:    void Display();
11: }
12:
13: public interface IShapeDisplay
14: {
15:     void Display();
16: }
17:
18: public class Square : IShape, IShapeDisplay
19: {
20:     private int sides;
21:     public int SideLength;
22:
23:     public int Sides
24:     {
25:         get { return sides; }
26:     }
27:
28:     public double Area()
29:     {
30:         return ((double) (SideLength * SideLength));
31:     }
32:
33:     public double Circumference()
34:     {
35:         return ((double) (Sides * SideLength));
36:     }
37:
38:     public Square()
39:     {
40:         sides = 4;

```

```
41:     \
42:
43:     void IShape.Display()
44:     {
45:         Console.WriteLine("\nDisplaying Square Shape's information:");
46:         Console.WriteLine("Side length: {0}", this.SideLength);
47:         Console.WriteLine("Sides: {0}", this.Sides);
48:         Console.WriteLine("Area: {0}", this.Area());
49:     }
50:     void IShapeDisplay.Display()
51:     {
52:         Console.WriteLine("\nThis method could draw the shape...");
53:     }
54:
55: }
56:
57: public class myApp
58: {
59:     public static void Main()
60:     {
61:         Square mySquare = new Square();
62:         mySquare.SideLength = 7;
63:
64:         IShape ish = (IShape) mySquare;
65:         IShapeDisplay ishd = (IShapeDisplay) mySquare;
66:
67:         ish.Display();
68:         ishd.Display();
69:     }
70: }
```

该程序清单的输出如下：

```
Displaying Square Shape's information:
Side length: 7
Sides: 4
Area: 49
```

```
This method could draw the shape...
```

分析：该程序清单稍微复杂些，但它说明了您可以在同一个类中显式地声明并使用不同接口中的名称相同的方法。

该程序清单中包含两个名为 `Display` 的方法，`Square` 类对它们显式地进行了定义。从第 43 和 50 行可以知道，显式定义使用了方法的显式名称。显式名称由句点分隔的接口名和方法名组成。

要使用这些显式接口，不仅仅需要调用方法，还需要完成其他工作。如果使用标准类名调用方法，实际将调用哪个方法呢？要使用某个方法，必须将类强制转换为相应的接口。这里需要将类强制转换为接口 `IShape` 或 `IShapeDisplay`。第 64 行声明了一个名为 `ish` 的 `IShape` 变量，并将 `mySquare` 赋给它。强制转换确保对象 `mySquare` 被看作是一个 `IShape`。

第 65 行将 `mySquare` 强制转换为 `IShapeDisplay`, 并将其赋给一个名为 `ishd` 的 `IShapeDisplay` 变量。从 67 和 68 行可以知道, 这些接口变量可以用来调用相应的 `Display` 方法。

该程序清单说明, 多个接口可以包含名称相同的方法。通过使用显式定义和强制转换, 可以确保正确的方法被调用。为何要这样做呢? 就该程序清单而言, 可以使用 `IShapeDisplay` 方法来确保所有的类都包含一个具备图形显示功能的方法。`IShape` 接口中的 `Display` 方法则用于提供详细的文本信息。通过实现这两个方法, 类将具备这两种显示功能。

13.5 从已有的接口派生出新的接口

和类一样, 接口也可以从另一个接口派生而来。继承接口的方式与继承类相似, 下面的代码片段演示了如何扩展前面创建的 `IShape` 接口:

```
public interface IShape
{
    long Area();
    long Circumference();
    int Sides { get; set; };
}
interface I3DShape : IShape
{
    int Depth { get; set; }
}
```

`I3DShape` 包含 `IShape` 接口的所有成员和新添加的成员, 这里新添加的是 `Depth` 属性成员。然后, 您便可以像使用其他接口一样来使用 `I3DShape` 接口。该接口包含的成员方法有 `Area`、`Circumference`、`Sides` 和 `Depth`。

13.6 隐藏接口成员

可以在基类中实现接口成员, 但不让基类访问它。这样可以接口, 而又避免成员给类带来的混乱。

要隐藏接口成员, 可以在类中显式地定义该成员。程序清单 13.5 是一个隐藏接口成员的例子。

程序清单 13.5 `hide.cs`: 对类隐藏接口成员

```
1: // hide.cs -
2: //-----
3:
4: using System;
5:
6: public interface IShape
7: {
8:     // members left out to simplify example...
9:     int ShapeShifter( int val );
10:    int Sides { get; set; }
11: }
```

```
12:
13: public class Shape : IShape
14: {
15:     private int sides;
16:
17:     public int Sides
18:     {
19:         get { return sides; }
20:         set { sides = value; }
21:     }
22:
23:     int IShape.ShapeShifter( int val )
24:     {
25:         Console.WriteLine("Shifting Shape...");
26:         val += 1;
27:         return val;
28:     }
29:
30:     public Shape()
31:     {
32:         Sides = 5;
33:     }
34: }
35:
36: public class MyApp
37: {
38:     public static void Main()
39:     {
40:         Shape myShape = new Shape();
41:
42:         Console.WriteLine("My shape has been created.");
43:         Console.WriteLine("Using get accessor. Sides = {0}", myShape.Sides);
44:
45:         // myShape.Sides = myShape.ShapeShifter(myShape.Sides); // error
46:
47:         IShape tmp = (IShape) myShape;
48:         myShape.Sides = tmp.ShapeShifter( myShape.Sides);
49:
50:         Console.WriteLine("ShapeShifter called. Sides = {0}", myShape.Sides);
51:     }
52: }
```

该程序清单的输出如下:

```
My shape has been created.
Using get accessor. Sides = 5
Shifting Shape....
ShapeShifter called. Sides = 6
```

分析：该程序清单包含了一个简化的、今天一直在使用的 IShape 接口。该程序清单重点演示了对类隐藏接口的成员，这里对 Shape 类隐藏了 ShapeShifter 方法，第 45 行被注释掉，它试图将 ShapeShifter 方法作为 Shape 类的成员来使用。如果删除该行的注释标记，并重新编译该程序，将出现下面的错误：

```
hide2.cs(45,21): error CS0117: 'Shape' does not contain a definition for 'ShapeShifter'
```

从上述错误可以知道，Shape 对象不能直接访问方法 ShapeShifter——该方法被隐藏了。

这是如何实现的呢？这是通过显式地定义接口成员来实现的。第 23 行中 ShapeShifter 方法的定义中包含这样的显式定义，其中包含了接口的名称。

调用显式定义的成员时，需要执行第 47 和 48 行那样的操作。您需要声明一个接口变量，然后将要使用的对象强制转换为接口类型。第 47 行创建了一个名为 tmp 的 IShape 接口变量，然后将 myShape 对象转换为相应的接口类型，并赋给该变量。第 48 行则使用 IShape 接口变量来访问 ShapeShifter 方法。从第 50 行的输出可以知道，相应的方法被调用。

tem 变量之所以能够访问该方法，是因为其类型与第 23 行中的显式声明相同。

13.7 总结

今天的课程是到目前为止最短的，但包含了大量重要的信息。您学习了接口——一种让您能够规定哪些必须实现的结构。接口可用于确保不同的类包含类似的实现。您学习了大量关于接口的知识，包括如何使用接口、如何扩展接口，以及如何在基类中实现它们的一些成员，同时隐藏这些成员。

13.8 问与答

问：理解接口很重要吗？

答：是的。您将发现，C#编程中都要使用接口。类库中提供的许多预定义的方法都使用了接口。

问：您说关键字 as 和 is 也可用于接口，但却没有举例。对于接口，这些关键字的用法与类中的情况有何不同？

答：将关键字 as 和 is 用于接口的方式与用于类几乎完全相同。由于用法是类似的，因此对于接口，编码范例也与第 11 天列举的几乎完全相同。

13.9 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

13.9.1 小测验

1. 接口是引用类型还是值类型？
2. 接口有何用途？
3. 接口成员被声明为公有的、私有的还是保护的？
4. 接口和类之间的主要差别在哪里？

5. 结构可使用什么样的继承?
6. 接口中可以包含哪些类型?
7. 如何声明一个这样的公有类: 它名为 `Aclass`, 从名为 `baseClass` 类派生而来, 同时实现一个名为 `ImyInterface` 的接口?
8. 可以同时继承多少个类?
9. 可以同时继承 (实现) 多少个接口?
10. 显式接口实现是如何完成的?

13.9.2 练习

1. 为名为 `Iid` 的接口编写代码, 该接口只包含一个名为 `ID` 的属性成员。
2. 编写声明一个名为 `Iposition` 的接口的代码。该接口包含一个接受一个 `Point` 值, 并返回一个布尔值的方法。

3. 下面的代码片段可能有问题。如果确实有问题, 是什么问题?

```
public interface IDimensions
{
    long Width;
    long Height;
    double Area();
    double Circumference();
    int Sides();
}
```

4. 在一个名为 `Rectangle` 的类中实现程序清单 13.1 声明的接口 `IShape`。

第 14 天课程

索引器、代表和事件

您已经学习了许多与 C# 编程相关的基本主题。今天您将学习其他几个主题，它们是您全面理解 C# 编程的基础。今天的课程包含以下内容：

- 索引器；
- 创建自己的索引器；
- 代表；
- 事件编程技术；
- 创建自己的事件和事件处理程序；
- 多点传送（multicast）。

14.1 使用索引器

新术语：第 8 天介绍了数组，今天将介绍索引器。索引器能够将索引用于对象来获得该对象中存储的值。这实际上让您能够像对待数组那样对待对象。

索引器类似于属性。和属性一样，定义索引器时，您也使用 `get` 和 `set`；不同的是，您取得的是对象中的值，而不是特定的数据成员。定义属性时，您需要定义属性的名称；而定义索引器时，您不是创建一个名称，而是使用关键字 `this`，它引用对象实例，因此实际上使用的是对象名称。定义索引器的格式如下：

```
public dataType this[int index]
{
    get
    {
        // Do whatever you want...
        return aValue;
    }
    set
    {
        // Do whatever you want
        // Generally you should set a value within the class
        // based on the index and the value they assign.
    }
}
```

```

}
```

创建索引器后，您便可以对对象使用方括号表示法（[]）来设置或取得对象中的值。从上面的格式可知，索引器将设置和返回的数据类型为 `datatype`。在 `get` 部分，将返回一个数据类型 `datatype` 的值；在 `set` 部分，则可以对数据类型为 `datatype` 的值执行某些操作。与属性和函数成员一样，您也可以使用关键字 `value`，这是作为参数传递给 `set` 方法的值。程序清单 14.1 是一个使用索引器的简单范例。

程序清单 14.1 `indexer.cs`：使用索引器

```

1: // indx2.cs - Using an indexer
2: //-----
3:
4: using System;
5:
6: public class SpellingList
7: {
8:     protected string[] words = new string[size];
9:     static public int size = 10;
10:
11:     public SpellingList()
12:     {
13:         for (int x = 0; x < size; x++)
14:             words[x] = String.Format("Word{0}", x);
15:     }
16:
17:     public string this[int index]
18:     {
19:         get
20:         {
21:             string tmp;
22:
23:             if( index >= 0 && index <= size-1 )
24:                 tmp = words[index];
25:             else
26:                 tmp = "";
27:
28:             return ( tmp );
29:         }
30:         set
31:         {
32:             if( index >= 0 && index <= size-1 )
33:                 words[index] = value;
34:         }
35:     }
36: }
37:
38: public class TestApp
39: {
```



```

40:     public static void Main()
41:     {
42:         SpellingList myList = new SpellingList();
43:
44:         myList[3] = "=====";
45:         myList[4] = "Brad";
46:         myList[5] = "was";
47:         myList[6] = "Here!";
48:         myList[7] = "=====";
49:
50:         for ( int x = 0; x < SpellingList.size; x++ )
51:             Console.WriteLine(myList[x]);
52:     }
53: }

```

该程序清单的输出如下：

```

Word0
Word1
Word2
=====
Brad
Was
Here!
=====
Word8
Word9

```

分析：该程序清单创建了一个用于 SpellingList 类的索引器。SpellingList 类包含一个名为 words 的字符串数组，可用于存储一系列的单词。该字符串数组的长度被设置为第 9 行声明的变量的值。

第 11 ~ 15 行是 SpellingList 类的构造函数，它设置数组中每个元素的初始值。您也可以从用户那里取得这些单词，或从文件中读取。该构造函数将每个数组元素的值设置为 Word###，其中###为数组元素的编号。

接下来来看第 38 ~ 53 行的 TestApp 类，从中可以知道 SpellingList 类是如何被使用的。第 42 行使用 SpellingList 类来实例化将来存储单词的 myList 对象，这将导致构造函数被执行。第 44 ~ 48 行修改了其中的一些值。

看到第 44 ~ 48 行时，如果您想到了使用数组的方法，您可能对此有疑问。要存取单词，通常需要访问对象的数据成员。当数据成员为数组时，则给第四个元素赋值的代码将如下：

```
MyList.words[3] = "=====";
```

但第 44 行访问的是对象的第四个元素，而设置的却是 words 数组的第四个元素。

第 17 ~ 35 行为 SpellingList 类创建了一个索引器，它让您能够使用对象名来访问 words 数组中的元素。

第 17 行是该索引器的定义，您知道这是一个索引器，而不是属性，因为其中使用的是关键字 this，而不是一个名称。另外，该行还提供一个名为 index 的索引，并指定索引器的返回值为一个字符串。

第 19~29 行是索引器的 `get` 部分,它根据 `index` 的值返回一个值。在这个例子中,返回的是 `words` 数组的一个元素。您可以根据需要返回任何值,但返回的值必须有意义。第 23 行执行检查,确保 `index` 的值是有效的。如果不检查 `index` 的值,则可能引发异常。在该程序清单中,如果 `index` 的值超出了范围,则返回一个空值,如第 26 行所示。如果 `index` 的值有效,则返回存储在 `words` 数组中 `index` 位置处的值。

索引器的 `set` 部分位于第 30~34 行,它用于设置对象中的信息。和属性一样,关键字 `value` 的为赋给的值。在这些代码中,也对 `index` 的值进行检查,以确保它是有效的。如果有效,则 `words` 数组中 `index` 位置处的单词被更新为赋给的值。

我们再来看一下 `testApp` 类。第 44~48 行使用索引器的 `set` 部分来赋值。第 44 行将调用索引器的 `set` 部分,传递的 `index` 值为 3,而 `value` 将为 `====`。在第 45 行, `value` 的值将为 `Brad`,而 `index` 的值为 4。第 51 行调用索引器的 `get` 部分,同时给 `index` 传递的值为 `x`;而返回的值将是索引器的 `get` 部分返回的字符串。

注意: 有时候应该使用索引器,有时候则不应该使用。当索引器能够提供代码的可读性,并使之更容易理解时,应该使用它。例如,您可以创建一个可以东西压入以及从中弹出的栈,此时如果使用索引器,则不用删除栈中的元素就可以存取它们。

14.2 代 表

新术语: 下面介绍一个更高级的主题——代表。代表是一种引用类型,它定义了方法调用的特征标。这样代表便可以接受并执行具有这种特征标格式的方法。人们经常将代表与接口进行比较。昨天的课程介绍过,接口是一种引用类型,它定义了类的设计方案 (`layout`),但本身并没有定义任何功能;而代表定义了方法的设计方案,但本身并不定义方法,而是接受并使用与其设计方案 (特征标) 匹配的方法。

举一个例子将有助于更清楚地说明代表。这里的例子是一个对两个数字进行排序 (可以是升序,也可以降序) 的程序。该程序中包含对数字进行排序的代码,但让用户指定排序方式。根据用户指定的排序方式 (升序或降序),将调用不同的方法。虽然将调用不同的方法,但程序将只调用一个代表。代表将执行合适的方法。

声明代表的格式如下:

```
public delegate returnType DelegateName(parameters);
```

其中 `public` 可替换为合适的存取限定符;而 `delegate` 是一个关键字,用于指示这是一个代表。定义的其他部分用于指定该代表将使用的方法的特征标。从前面的课程知道,特征标包含方法返回的数据类型 (`returnType`) 以及方法将接受的参数类型和个数 (`parameters`)。代表的名称与常规的方法名称类似。由于代表将用于执行多个返回类型和参数匹配的方法,因此您无法知道具体的方法名称。

这里创建一个名为 `Sort` 的代表,它可以接受多个排序方法。这些方法不返回任何值,因此将它们返回值设置为 `void`。这些将用来排序的方法接受两个按引用传递的 `int` 变量,这使排序函数在必要时可以交换这些值。因此,代表的定义如下:

```
public delegate void Sort(ref int a, ref int b);
```

请注意最后的分号，虽然上述代码看起来像一个方法定义，但不是。这里没有主体 (body)，因为代表只是可以被执行的方法的模板。方法被委托给该代表来执行。

代表是多个方法的模板。例如代表 Sort 可使用任何不返回值的，并接受两个按引用传递 int 参数的方法。下面是一个 Sort 代表可使用的方法：

```
public static void Ascending( ref int first, ref int second )
{
    if (first > second )
    {
        int tmp = first;
        first = second;
        second = tmp;
    }
}
```

该方法 (Ascending) 的返回类型为 void，它接受两个 ref int 参数。这与 Sort 代表的特征标匹配，因此该代表可以使用它。该方法接受两个值，并检查第一个是否比第二个大。如果是这样，则交换两个变量的值。由于这些值是按引用传递的，因此原来的两个变量的值也将被交换。

第二个方法名为 Descending：

```
public static void Descending( ref int first, ref int second )
{
    if (first < second )
    {
        int tmp = first;
        first = second;
        second = tmp;
    }
}
```

该方法与 Ascending 类似，只是将大的值放在第一个变量中而已。您还可以声明其他供该代表使用的排序方法，只要其特征标与该代表匹配即可。另外，其他的程序也可以使用 Sort 代表，但其相应的方法中应包含自己的逻辑。

至此，声明了代表及其使用的多个方法，接下来需要做什么呢？

您需要将方法与代表关联起来，为此需要实例化代表对象。代表对象的声明方式与其他对象类似，只是将委托给代表的方法的名称作为参数。例如，要声明一个可使用 Ascending 方法的代表对象，则代码如下：

```
Sort up = new Sort(Ascending);
```

这创建了一个名为 up 的代表对象，然后便可以使用它。up 与 Ascending 方法关联在一起。接下来创建一个与 Descending 方法关联在一起的 Sort 代表对象，该代表对象名为 down：

```
Sort down = new Sort(Descending);
```

至此，您声明了代表，创建了代表可使用的对象，并将这些方法和代表对象关联起来了。如何使被委托的方法执行呢？创建一个接受一个代表对象的方法。该通用方法便可以执行被委托给代表的方法：

```
public void DoSort(Sort ar)
```

```

{
    ar(ref val1, ref val2);
}

```

正如您看到的，DoSort 方法接受一个名为 ar 的代表对象，然后执行 ar。需要注意的是，ar 的特征标与代表相同。如果代表有返回类型，则 ar 也将有返回类型。ar 调用的方法也与代表匹配。实际上，DoSort 方法执行的是作为 Sort 代表对象传递的那个方法。就这个例子而言，如果传递的是 up，则 ar(...) 等同于调用 Ascending 方法；如果传递的是 down，则 ar(...) 等同于调用 Descending 方法。

至此，您明白了使用代表时涉及到的各个关键部分，程序清单 14.2 将整个范例整合成了一个可运行的应用程序。

程序清单 14.2 deleg1.cs: 使用简单的代表

```

1: // deleg1.cs - Using a delegates
2: //-----
3:
4: using System;
5:
6: public class SortClass
7: {
8:     static public int val1;
9:     static public int val2;
10:
11:     public delegate void Sort(ref int a, ref int b);
12:
13:     public void DoSort(Sort ar)
14:     {
15:         ar(ref val1, ref val2);
16:     }
17: }
18:
19: public class SortProgram
20: {
21:     public static void Ascending( ref int first, ref int second )
22:     {
23:         if (first > second )
24:         {
25:             int tmp = first;
26:             first = second;
27:             second = tmp;
28:         }
29:     }
30:
31:     public static void Descending( ref int first, ref int second )
32:     {
33:         if (first < second )
34:         {
35:             int tmp = first;
36:             first = second;

```

```
37:         second = tmp;
38:     }
39: }
40:
41: public static void Main()
42: {
43:     SortClass.Sort up = new SortClass.Sort(Ascending);
44:     SortClass.Sort down = new SortClass.Sort(Descending);
45:
46:     SortClass doIT = new SortClass();
47:
48:     SortClass.val1 = 310;
49:     SortClass.val2 = 220;
50:
51:     Console.WriteLine("Before Sort: val1 = {0}, val2 = {1}",
52:         SortClass.val1, SortClass.val2);
53:     doIT.DoSort(up);
54:     Console.WriteLine("After Sort: val1 = {0}, val2 = {1}",
55:         SortClass.val1, SortClass.val2);
56:
57:     Console.WriteLine("Before Sort: val1 = {0}, val2 = {1}",
58:         SortClass.val1, SortClass.val2);
59:     doIT.DoSort(down);
60:     Console.WriteLine("After Sort: val1 = {0}, val2 = {1}",
61:         SortClass.val1, SortClass.val2);
62: }
63: }
```

该程序清单的输出如下:

```
Before Sort: val1 = 310, val2 = 220
After Sort: val1 = 220, val2 = 310
Before Sort: val1 = 220, val2 = 310
After Sort: val1 = 310, val2 = 220
```

分析: 该程序清单首先声明了一个 `SortClass` 类, 用于包含 `Sort` 代表。这个类包含两个公有的静态变量 (如第 8 和 9 行所示), 用于存储要排序的值。这里之所以将其声明为静态的, 是为了简化该程序清单的编码。第 11 行是代表的定义, 接下来为方法 `DoSort` (如第 13 ~ 16 行所示), 它将执行被委托的方法。正如您看到的, `DoSort` 方法接受一个代表对象, 第 15 行通过使用与第 11 行的代表定义中相同的特征标, 将该代表对象作为一个方法调用。

`SortProgram` 使用了 `Sort` 代表。这个类定义了将被委托的方法——`Ascending` 和 `Descending`, 同时包含了实现关键逻辑的 `Main` 方法。第 43 和 44 行创建了两个代表对象, 它们是前面介绍过的对象 `up` 和 `down`。

第 46 行创建了一个 `SortClass` 对象, 为使用 `DoSort` 方法, 这是必须的。第 48 和 49 行设置了要排序的变量的值。第 51 行将这些值打印到控制台。然后, 第 53 行调用 `DoSort` 方法, 并给它传递一个代表对象, 这里传递的是 `up`。这将导致 `Ascending` 方法被调用。从第 57 行的输出可以知道, 排序完成了。然后, 第 59 行再次调用 `DoSort` 方法, 但传递的是 `down` 对象。这导致 `Descending` 方法被调

用，从输出中可以知道这一点。

实际上，可以将 DoSort 方法声明为静态的，这样就不用在第 46 行声明一个名为 doIT 的 SortClass 对象，而可以使用类名（而不是对象名）来访问 DoSort 方法：

```
SortClass.DoSort (...);
```

这个例子在使用代表时，使用的是硬编码值和具体的调用。当您编写更为动态的程序时，代表的价值将更为明显。例如，可以将程序清单 14.2 修改成使用文件中的数据或用户输入的值。这样排序的价值将更高。另外，还可以让用户来指定按升序、降序还是其他方法来排序。然后，根据用户指定的排序方式，调用 DoSort，并传递相应的代表对象。

注意：在今天的练习中，将要求您创建一个这样的代表，即可使用对整数数组进行排序的方法，附录 A 将提供答案，您可以在答案的基础上做进一步的改进，将数组的类型声明为对象，然后创建一个可用于对任何数据类型的数据进行排序的代表。

14.3 事件

您将发现，代表主要是在处理事件时使用。事件是类发出的通知，指出发生了某种事情。这样其他的类可以根据通知执行某种操作。

最常见的事件处理的例子是 Microsoft Windows。在 Windows 中，将显示对话框或窗口。这样用户执行大量不同的操作——可以单击按钮、选择菜单、输入文本等。每当用户执行这种操作时，事件便发生了。然后，Windows 中的事件处理程序将根据发生的事件做出相应的反应。例如，如果用户单击按钮，将发生 ButtonClick 事件。这样 ButtonClick 事件处理程序将采取所需的行动。

14.4 创建事件

创建并使用事件的步骤包括为事件建立代表、创建一个类来给事件处理程序传递参数、声明事件对应的代码、创建事件发生时将执行的代码（处理程序）以及使事件发生。

14.4.1 事件的代表

要使用事件，首选需要为它创建一个代表。为事件创建的代表格式如下：

```
delegate void EventHandlerName(object source, xxxEventArgs e);
```

其中，EventHandlerName 是事件处理程序对应的代表的名称。事件的代表总是接受两个参数，第一个参数 object source 是引发事件的源，第二个参数 xxxEventArgs e 是一个类，它包含事件处理程序可以使用的数据。这个类是从 EventArgs 类派生而来的，后者位于名称空间 System 中。

下面的代码行为事件创建一个代表。该事件检查赋给的字符，如果是特定的字符，则事件发生：

```
delegate void CharEventHandler(object source, CharEventArgs e);
```

上述代码声明了一个名为 CharEventHandler 的代表。从该声明可以知道，需要从 EventArgs 派生出一个名为 CharEventArgs 的类。

14.4.2 EventArgs 类

EventArgs 类用于将数据传递给事件处理程序，它可以派生出这样的新类，即包含用于存储所

需值的数据成员。派生类的格式如下：

```
public class xxxEventArgs : EventArgs
{
    // Data members

    public xxxEventArgs( type name )
    {
        //Set up values
    }
}
```

正如您看到的，xxxEventArgs 是从 EventArgs 派生而来的。您可以将 xxxEventArgs 重命名为您希望的任何名称。使用以 EventArgs 结尾的名称使这种类的用途更为明显。

提示：虽然可以给 xxxEventArgs 类指定任何名称，但应该以 EventArgs 结尾，这指明了这种类的用途，也与其他人的编程方式更为一致。

然后，您便可以在派生类中加入数据成员，并在构造函数中加入初始化数据成员值的逻辑。这个类将被传递给事件处理程序，其中应该包含事件处理程序需要的所有数据。

在前一节的范例中，创建的代表名为 CharEventHandler，它传递一个 CharEventArgs 对象。CharEventArgs 的代码如下：

```
public class CharEventArgs : EventArgs
{
    public char CurrChar;
    public CharEventArgs(char CurrChar)
    {
        this.CurrChar = CurrChar;
    }
}
```

正如您看到的，CharEventArgs 是一个从 EventArgs 派生而来的新类。不管要处理的是什么事，都需要按上述方式从 EventArgs 派生出一个类。上述派生类包含一个字符变量 CurrChar，事件处理程序中的代码将可以使用该变量的值；它还包含一个构造函数，当这个类的对象被创建时，构造函数将接受一个字符并执行。传递给构造函数的字符将被赋给类中的数据成员。

14.4.3 事件类的代码

可以创建一个引发事件的类，它包含事件的声明。事件声明的格式如下：

```
public event xxxEventHandler EventName;
```

其中 xxxEventHandler 是为事件创建的代表，EventName 是要声明的事件的名称。概括地说，这行代码使用关键字 event 创建一个名为 EventName 的事件实例，其类型为 xxxEventHandler。EventName 将被用于给代表指定方法以及执行方法。

下面是一个事件类的例子：

```
1: class CharChecker
2: {
3:     char curr_char;
4:     public event CharEventHandler TestChar;
```

```
5:     public char Curr_Char
6:     {
7:         get { return curr_char; }
8:         set
9:         {
10:            if (TestChar != null )
11:            {
12:                CharEventArgs args = new CharEventArgs(value);
13:                TestChar(this, args);
14:                curr_char = args.CurrChar;
15:            }
16:        }
17:    }
18: }
```

这个类包含在条件满足时将引发事件的代码。您编写的代码可能与此不同，但有几个方面是类似的。第 4 行使用前面创建的代表 `CharEventHandler` 创建了一个事件对象。该事件对象将被用来执行指定的事件处理程序。接下来（第 5~17 行）是属性 `Curr_Char` 的定义。正如您看到的，`get` 属性返回该类中数据成员 `curr_char` 的值（第 7 行）。

第 8~16 行的 `set` 属性很独特。它首先检查 `TestChar` 对象是否等于 `null`（第 10 行）。记住，`TestChar` 被声明为一个事件对象，仅当没有对应的事件处理程序时，这个对象才会为 `null`，这将在下一节做更详细的介绍。如果有事件处理程序，则第 12 行创建一个 `CharEventArgs` 对象。前一节介绍过，该对象将包含事件处理程序需要的所有值。传递给 `set` 方法的值被传递给 `CharEventArgs` 的构造函数，前一节介绍过，这是一个事件处理程序可以使用的字符。

第 13 行调用事件代表，这是通过使用第 4 行创建的事件对象来实现的。由于这是一个事件对象，因此将检查与该对象关联在一起的所有方法。正如您看到的，给该对象传递了两个值，第一个是 `this`，它指的是调用事件的对象，第二个是 `args`，这是前一行声明的 `CharEventArgs` 对象。

第 14 行是该事件所特有的，它将 `CharEventArgs` 对象中包含的字符赋给数据成员 `curr_char`。如果第 13 行调用的事件处理程序修改了数据，则这一行确保该事件类获得更新后的值——这正是 `set` 属性的用途所在。

14.4.4 创建事件处理程序

至此，您创建了代表、用于将信息传递给事件处理程序的结构以及引发事件的代码。现在需要创建事件处理程序。事件处理程序是一个代码片段，在事件发生时将被通知。事件处理程序是使用代表的格式创建的一个方法，其格式如下：

```
void handlername(object source, xxxEventArgs argName)
{
    //Event Handler code
}
```

其中 `handlername` 是事件发生时将被调用的方法的名称。现在，您对传递给该方法的两个参数应该很熟悉。第一个是引发事件的对象；第二个是一个从 `EventArgs` 派生而来的类，它包含事件处理程序要使用的值。`Event Handler code` 可以是任何代码。如果事件是单击按钮，则这些代码将在按钮被单击时执行。如果是 `Cancel` 按钮，则这些代码将执行取消的逻辑。如果是 `OK` 按钮，则这些代

码将执行一切顺利时的逻辑。

回到关于字符事件的例子。可以声明一个这样的事件处理程序，即如果输入的是字母 A，则将其替换为 X。这很可笑，但易于理解：

```
static void Drop_A(object source, CharEventArgs e)
{
    if(e.CurrChar == 'a' || e.CurrChar == 'A' )
    {
        Console.WriteLine("Don't like 'a!'");
        e.CurrChar = 'X';
    }
}
```

正如您看到的，该事件处理程序接受了一个 `CharEventArgs` 参数。然后从该对象中取出 `CurrChar` 的值，并对其进行检查。如果用户输入的是 A 或 a，则显示一条消息，并将当前的字符改为 X；如果是其他字符，则不执行任何操作。

14.4.5 将事件处理程序和事件关联起来

现在您几乎有了所需的一切，应该将事件处理程序和事件关联起来，以便事件发生时执行该处理程序。这应该在主程序中进行。

要将事件处理程序与事件关联起来，必须首先声明一个包含事件的对象。就关于字符的范例而言，这可以通过声明一个 `CharChecker` 对象来实现：

```
CharChecker tester = new CharChecker();
```

正如您看到的，这种对象的实例化方式与其他对象相同。创建这样的对象后，便可以使用事件了。每当 `CharChecker` 对象的 `set` 逻辑被调用时，便将按其中的逻辑运行，包括创建事件对象和引发事件。

但现在并未将事件处理程序和事件关联起来。要将事件处理程序和该对象关联起来，需要使用 `+=` 运算符，其格式如下：

```
ObjectWithEventName.EventObj += new EventDelegateName(EventName);
```

其中 `ObjectWithEventName` 是您使用事件类声明的对象，在这个例子中是 `tester`；`EventObj` 是您在事件类中声明的事件对象，这里为 `TestChar`；运算符 `+=` 是一个指示器，指出接下来要将一个事件处理程序加入到事件中。关键字 `new` 指出应创建接下来的事件处理程序。最后事件处理程序的名称被传递给代表 `EventDelegateName`。就这个例子而言，最终的语句如下：

```
tester.TestChar += new CharEventHandler(Drop_A);
```

14.4.6 将所有的东西组合起来

这大量的工作要做，但完成后，便可以使用像下面那么简单的代码行来引发该事件：

```
tester.Curr_Char = 'B';
```

程序清单 14.3 将所有的东西组合了起来。

程序清单 14.3 使用事件和事件处理程序

```
1: // events.cs - Using events
2: //-----
```

```
3:
4: using System;
5:
6: delegate void CharEventHandler(object source, CharEventArgs e);
7:
8: public class CharEventArgs : EventArgs
9: {
10:     public char CurrChar;
11:     public CharEventArgs(char CurrChar)
12:     {
13:         this.CurrChar = CurrChar;
14:     }
15: }
16:
17: class CharChecker
18: {
19:     char curr_char;
20:     public event CharEventHandler TestChar;
21:     public char Curr_Char
22:     {
23:         get { return curr_char; }
24:         set
25:         {
26:             if (TestChar != null )
27:             {
28:                 CharEventArgs args = new CharEventArgs(value);
29:                 TestChar(this, args);
30:                 curr_char = args.CurrChar;
31:             }
32:         }
33:     }
34: }
35:
36: class MyApp
37: {
38:     static void Main()
39:     {
40:         CharChecker tester = new CharChecker();
41:
42:         tester.TestChar += new CharEventHandler(Drop_A);
43:
44:         tester.Curr_Char = 'B';
45:         Console.WriteLine("{0}", tester.Curr_Char);
46:
47:         tester.Curr_Char = 'r';
48:         Console.WriteLine("{0}", tester.Curr_Char);
49:
50:         tester.Curr_Char = 'a';
51:         Console.WriteLine("{0}", tester.Curr_Char);
52:
```

```

53:     tester.Curr_Char = 'd';
54:     Console.WriteLine("{0}", tester.Curr_Char);
55:
56: }
57:
58: static void Drop_A(object source, CharEventArgs e)
59: {
60:     if(e.CurrChar == 'a' || e.CurrChar == 'A' )
61:     {
62:         Console.WriteLine("Don't like 'a!'");
63:         e.CurrChar = 'X';
64:     }
65: }
66: }

```

该程序清单的输出如下：

```

B
r
Don't like 'a!'
X
d

```

分析：该程序清单包含了前面各节讨论的代码。只有Main例程中的第44~54行是新的，这些代码将字符赋给tester对象的Curr_Char。如果赋给的是A或a，则打印一条消息，并将其修改为X。通过调用Console.WriteLine显示Curr_Char的值，来说明修改后的情况。

事件类可以是您想创建的任何类，例如，可以将这个例子中的CharChecker类修改为存储全名或其他文本信息。换句话说，这个例子的功能很少，但您可以通过修改代码来完成更多的功能。

14.4.7 多个事件处理程序

可以通过多点传送（multicasting）为同一个事件声明多个事件处理程序。新加入的处理程序必须有同样的格式，即接受一个object对象和一个从EventArgs派生的对象，并且返回值void。要加入其他的事件处理程序，应以前面介绍的方式使用运算符+=。程序清单 14.4 对 14.3 做了修改，加入了另一个事件处理程序。

程序清单 14.4 Event2.cs: 多个事件处理程序

```

1: // events2.cs - Using multiple event handlers
2: //-----
3:
4: using System;
5:
6: delegate void CharEventHandler(object source, CharEventArgs e);
7:
8: public class CharEventArgs : EventArgs
9: {
10:     public char CurrChar;
11:     public CharEventArgs(char CurrChar)
12:     {

```

```
13:         this.CurrChar = CurrChar;
14:     }
15: }
16:
17: class CharChecker
18: {
19:     char curr_char;
20:     public event CharEventHandler TestChar;
21:     public char Curr_Char
22:     {
23:         get { return curr_char; }
24:         set
25:         {
26:             if (TestChar != null )
27:             {
28:                 CharEventArgs args = new CharEventArgs(value);
29:                 TestChar(this, args);
30:                 curr_char = args.CurrChar;
31:             }
32:         }
33:     }
34: }
35:
36: class MyApp
37: {
38:     static void Main()
39:     {
40:         CharChecker tester = new CharChecker();
41:
42:         tester.TestChar += new CharEventHandler(Drop_A);
43:         tester.TestChar += new CharEventHandler(Change_D);
44:
45:         tester.Curr_Char = 'B';
46:         Console.WriteLine("{0}", tester.Curr_Char);
47:
48:         tester.Curr_Char = 'r';
49:         Console.WriteLine("{0}", tester.Curr_Char);
50:
51:         tester.Curr_Char = 'a';
52:         Console.WriteLine("{0}", tester.Curr_Char);
53:
54:         tester.Curr_Char = 'd';
55:         Console.WriteLine("{0}", tester.Curr_Char);
56:     }
57:
58:     static void Drop_A(object source, CharEventArgs e)
59:     {
60:         if(e.CurrChar == 'a' || e.CurrChar == 'A' )
61:         {
62:             Console.WriteLine("Don't like 'a!'");
```

```

63:         e.CurrChar = 'X';
64:     }
65: }
66:
67: // new event handler....
68: static void Change_D(object source, CharEventArgs e)
69: {
70:     if(e.CurrChar == 'd' || e.CurrChar == 'D' )
71:     {
72:         Console.WriteLine("D's are good!");
73:         e.CurrChar = 'Z';
74:     }
75: }
76: }

```

该程序清单的输出如下:

```

B
r
Don't like 'a'!
X
D's are good!
Z

```

分析: 第68~75行新添加一个名为Change_D的事件处理程序。正如您看到的,其格式与要求的相同——返回类型为void,接受两个正确的参数。该事件处理程序检查赋给的字母是否是D或d,如果是,则显示一条消息,然后将其改为Z。这不很有趣,但很有效。

第43行将该事件处理程序加入到事件中。正如您看到的,加入的方式与原来的事件处理程序Drop_A相同。现在,赋给字母时,两个事件处理程序都将执行。

14.4.8 删除事件处理程序

可以添加事件处理程序,当然也可删除。要删除事件处理程序,应使用运算符-=,而不是+=。程序清单14.5列出了CharChecker程序的新Main例程。

警告: 该程序清单不完整,它只包含Main方法。可以用这些代码替换程序清单14.4中的Main方法,也可以从www.TeachYourselfCSharp.com下载该程序清单的全部源代码。

程序清单 14.5 events3.cs: 删除事件处理程序

```

1: static void Main()
2: {
3:     CharChecker tester = new CharChecker();
4:
5:     tester.TestChar += new CharEventHandler(Drop_A);
6:     tester.TestChar += new CharEventHandler(Change_D);
7:
8:     tester.Curr_Char = 'B';
9:     Console.WriteLine("{0}", tester.Curr_Char);
10:
11:     tester.Curr_Char = 'r';

```

```
12:     Console.WriteLine("{0}", tester.Curr_Char);
13:
14:     tester.Curr_Char = 'a';
15:     Console.WriteLine("{0}", tester.Curr_Char);
16:
17:     tester.Curr_Char = 'd';
18:     Console.WriteLine("{0}", tester.Curr_Char);
19:
20:     // Remove event handler...
21:     Console.WriteLine("\nRemoving event handler...");
22:     tester.TestChar -= new CharEventHandler(Change_D);
23:
24:     // Try D-a-d...
25:
26:     tester.Curr_Char = 'D';
27:     Console.WriteLine("{0}", tester.Curr_Char);
28:
29:     tester.Curr_Char = 'a';
30:     Console.WriteLine("{0}", tester.Curr_Char);
31:
32:     tester.Curr_Char = 'd';
33:     Console.WriteLine("{0}", tester.Curr_Char);
34: }
```

该程序清单的输出如下：

```
B
r
Don't like 'a'!
X
D's are good!
Z

Removing event handler...
D
Don't like 'a'!
x
d
```

分析：从输出中可以知道，当第22行执行后，事件处理程序Change_D便不再处于活动状态但事件处理程序Drop_A仍有效。

警告：如果给事件指定了多个处理程序，则无法保证首先执行哪一个处理程序。对于程序清单14.5而言，则无法保证Drop_A在Change_D之前执行。

另外，事件处理程序和事件可能会引发异常以及出现其他代码可能出现的任何问题。如果发生异常，则不能保证其他事件处理程序一定会执行。

14.5 总 结

今天介绍了 C# 中一些比较复杂的主题。首先介绍了索引器。索引器可用于类，这样便可以使用索引表示法访问类了。这使得类就像数组。

接着介绍了代表。代表类似接口：定义了存取方式，但不提供实现。代表建立了使用方法的格式。通过使用代表，只需单个方法调用，便可以动态地调用不同的方法。

最后介绍了事件。可以创建代码来使事件发生，更重要的是，可以创建代码（事件处理程序）来在事件发生时做出反应。

14.6 问与答

问：今天介绍的概念比较难，理解它们有多重要呢？

答：在不理解今天介绍的概念的情况下，使用 C# 也可以完成很多工作，但有更多的工作您将无法完成。如果要编写 Windows 或其他图形环境应用程序，则事件是至关重要的。正如今天介绍的，代表对于使用事件至关重要。

许多 C# 编辑器（如 Visual Studio.NET）能够自动为您创建大量的代码，这很有帮助。例如，Visual Studio.NET 会添加许多标准事件的代码。

问：在今天的课程中，事件是在属性中声明的，必须这样吗？

答：不。可以在属性或方法中声明事件调用。

问：可以给事件指定多个事件处理程序，那么可以给同一个代表指定多个方法吗？

答：可以。可以给代表指定多个方法，这样一个调用便能执行多个方法。这也被称为多点传送。

问：函数指针是什么？

答：在诸如 C 和 C++ 等语言中，有一种名叫函数指针的结构。函数指针的功能与代表相同，但代表是类型安全的，也是可靠的。除了用来引用方法外，代表也可被事件使用。

14.7 作 业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

14.7.1 小测验

1. 您呆在起居室，此时电话响了，您起身接电话。电话铃声与下面哪个概念最为类似？

- a. 索引器；
- b. 代表；
- c. 事件；
- d. 事件处理程序；
- e. 异常处理程序。

2. 接电话与下面的哪个概念最为类似？

- a. 索引器；
- b. 代表；

- c. 事件;
 - d. 事件处理程序;
 - e. 异常处理程序。
3. 索引器的主要功能是什么?
 4. 使用哪个关键字来声明索引器?
 5. 索引器的定义与下面的哪种定义类似?
 - a. 类定义;
 - b. 对象定义;
 - c. 属性定义;
 - d. 代表定义;
 - e. 事件定义。
 6. 创建和使用事件包含哪些步骤?
 7. 哪种运算符用于添加事件处理程序?
 8. 给同一个事件指定多个事件处理程序被称为什么?
 9. 下面哪些是正确的 (注意: 可能没有一个是正确的, 也可能都是正确的)?
 - 事件是一个基于代表的实例;
 - 代表是一个基于事件的实例。
 10. 可以在类中的什么位置实例化事件?

14.7.2 练习

1. 给下面的类添加一个索引器, 并在一个简单的程序中使用该类。

```
public class SimpleClass
{
    int[] numbers;

    public SimpleClass(int size)
    {
        numbers = new int[size];           // declare size elements
        for ( int x = 0; x < size; x++ )    // initialize values to 0.
            numbers[x] = 0;
    }
}
```

2. 改写程序清单 14.1, 不使用索引器。
3. 修改程序清单 14.2, 以便不必声明一个 `doIT` 对象。
4. 编写一个使用代表的程序, 对整型数组中的元素进行排序。您可以以程序清单 14.2 为基础。
5. 在程序清单 14.5 的代码中添加一个事件处理程序, 该事件处理程序将小写的元音字母改为大写。

第2周复习

您已经完成了学习 C# 的第二周课程，现在您知道了 C# 中的许多重要的基本主题。

下面的程序清单将其中的许多概念整合到一个程序中，该程序的功能要比各个课程中的范例强一些。该程序很长，但不太有趣。

该程序清单是一个功能有限的“21 点”扑克游戏。它显示您手中的牌以及庄家（计算机）手中的第一张牌。“21 点”的思想是，使您手中的牌的总点数尽可能接近 21 点，但不要超过。花牌（J、Q 和 K）为 10 点，其他牌的点数与其基数相同。A 的点数可以是 1 点也可以是 11 点，这由您决定。

庄家（计算机）手中牌的总点数不能少于 17 点，否则必须在要一张牌。如果庄家总点数超过 21 点，则爆了。如果您超过 21 点，则您爆了，计算机（庄家）自动获胜。

程序清单 WF2.1 21 点游戏

```
1: // cards.cs -
2: //     Blackjack
3: //-----
4:
5: using System;
6:
7: public enum CardSuit
8: {
9:     Zero_Error,
10:    clubs,
11:    diamonds,
12:    hearts,
13:    spades
14: }
15:
16: public enum CardValue
17: {
18:     Zero_Error,
19:     Ace,
20:     two,
21:     three,
22:     four,
23:     five,
```

```
24:     six,
25:     seven,
26:     eight,
27:     nine,
28:     ten,
29:     Jack,
30:     Queen,
31:     King
32: }
33:
34: // Structure: card
35: //=====
36: struct card
37: {
38:     public CardSuit suit; // 1 - 4
39:     public CardValue val; // 1 - 13
40:
41:     public int CardValue
42:     {
43:         get
44:         {
45:             int retval;
46:
47:             if( (int) this.val >= 10)
48:                 retval = 10;
49:             else
50:                 if( (int) this.val == 1 )
51:                     retval = 11;
52:                 else
53:                     retval = (int) this.val;
54:
55:             return retval;
56:         }
57:     }
58:
59:     public override string ToString()
60:     {
61:         return (string.Format("{0} of {1}", this.val.ToString("G"),
62:                                 this.suit.ToString("G")));
63:     }
64: }
65:
66: // Class: deck
67: //=====
68: class deck
69: {
70:     public card [] cards = new card[53] ;
71:     int next;
72:
73:     // deck()
```

```

74:  // Constructor for setting up a regular deck
75:  //=====
76:  public Deck()
77:  {
78:      next = 1;  // initialize pointer to point to first card.
79:
80:      // Initialize the cards in the deck
81:      cards[0].val = 0;  // card 0 is set to 0.
82:      cards[0].suit = 0;
83:
84:      int currcard = 0;
85:      for( int suitctr = 1; suitctr < 5; suitctr++ )
86:      {
87:          for( int valctr = 1; valctr < 14; valctr++ )
88:          {
89:              currcard = (valctr) + ((suitctr - 1) * 13);
90:              cards[currcard].val = (CardValue) valctr;
91:              cards[currcard].suit = (CardSuit) suitctr;
92:          }
93:      }
94:  }
95:
96:  // shuffle()
97:  // Randomizes a deck's cards
98:  //=====
99:  public void shuffle()
100:  {
101:      Random rnd = new Random();
102:      int sort1;
103:      int sort2;
104:      Card tmpcard = new Card();
105:
106:      for( int ctr = 0; ctr < 100; ctr++ )
107:      {
108:          sort1 = (int) ((rnd.NextDouble() * 52) + 1);
109:          sort2 = (int) ((rnd.NextDouble() * 52) + 1);
110:
111:          tmpcard = this.cards[sort1];
112:          this.cards[sort1] = this.cards[sort2];
113:          this.cards[sort2] = tmpcard;
114:      }
115:
116:      this.next = 1;  // reset pointer to first card
117:  }
118:
119:  // dealCard()
120:  // Returns next card in deck
121:  //=====
122:  public Card dealCard()
123:  {

```

```
124:     if( next > 52 )
125:     {
126:         // At end of deck
127:         return (this.cards[0]);
128:     }
129:     else
130:     {
131:         // Returns current card and increments next
132:         return this.cards[next++];
133:     }
134: }
135: }
136:
137: // Class: CardGame
138: //=====
139:
140: class CardGame
141: {
142:     static deck mydeck = new deck();
143:     static card [] pHand = new card[10];
144:     static card [] cHand = new card[10];
145:
146:     public static void Main()
147:     {
148:         int pCardCtr = 0;
149:         int pTotal = 0;
150:         int cTotal = 0;
151:
152:         bool playing = true;
153:
154:         while ( playing == true )
155:         {
156:             //CLEAR HANDS
157:             pTotal = 0;
158:             cTotal = 0;
159:             pCardCtr = 0;
160:
161:             for ( int ctr = 0; ctr < 10; ctr++)
162:             {
163:                 pHand[ctr].val = 0;
164:                 pHand[ctr].suit = 0;
165:             }
166:
167:             Console.WriteLine("\nShuffling cards...");
168:             mydeck.shuffle();
169:
170:             Console.WriteLine("Dealing cards...");
171:
172:             pHand[0] = mydeck.dealCard();
173:             cHand[0] = mydeck.dealCard();
```

第2周复习

```
174:         pHand[1] = mydeck.dealCard();
175:         cHand[1] = mydeck.dealCard();
176:
177:         // Set computer total equal to its first card...
178:         cTotal = cHand[0].CardValue;
179:
180:         bool playersTurn = true;
181:
182:         do
183:         {
184:             Console.WriteLine("\nPlayer\'s Hand:");
185:             pCardCtr = 0;
186:             pTotal = 0;
187:
188:             do
189:             {
190:                 Console.WriteLine(" Card {0}: {1}",
191:                                     pCardCtr + 1,
192:                                     pHand[pCardCtr].ToString());
193:
194:                 // Add card value to player total
195:                 pTotal += pHand[pCardCtr].CardValue;
196:
197:                 pCardCtr++;
198:
199:             } while ((int) pHand[pCardCtr].val != 0);
200:
201:             Console.WriteLine("Dealer\'s Hand:");
202:
203:             Console.WriteLine(" Card 1: {0}",
204:                                 cHand[0].ToString());
205:
206:
207:             Console.WriteLine("-----");
208:             Console.WriteLine("Player Total = {0} \nDealer Total = {1}",
209:                                 pTotal, cTotal);
210:
211:
212:             if( pTotal <= 21 )
213:             {
214:                 playersTurn = GetPlayerOption(pCardCtr);
215:             }
216:             else
217:             {
218:                 playersTurn = false;
219:             }
220:
221:         } while(playersTurn == true);
222:
223:         // Player's turn is done
```

```
224:
225:     if ( pTotal > 21 )
226:     {
227:         Console.WriteLine("\n\n**** BUSTED ****\n");
228:     }
229:     else // Determine computer's score
230:     {
231:         // Tally Computer's current total...
232:         cTotal += cHand[1].CardValue;
233:
234:         int cCardCtr = 2;
235:
236:         Console.WriteLine("\n\nPlayer's Total: {0}", pTotal);
237:         Console.WriteLine("\nComputer: ");
238:         Console.WriteLine(" {0}", cHand[0].ToString());
239:         Console.WriteLine(" {0} TOTAL: {1}",
240:             cHand[1].ToString(),
241:             cTotal);
242:
243:         while ( cTotal < 17 ) // Less than 17, must draw
244:         {
245:             cHand[cCardCtr] = mydeck.dealCard();
246:             cTotal += cHand[cCardCtr].CardValue;
247:             Console.WriteLine(" {0} TOTAL: {1}",
248:                 cHand[cCardCtr].ToString(),
249:                 cTotal);
250:             cCardCtr++;
251:         }
252:
253:         if (cTotal > 21 )
254:         {
255:             Console.WriteLine("\n\nComputer Busted!");
256:             Console.WriteLine("YOU WON!!!");
257:         }
258:         else
259:         {
260:             if( pTotal > cTotal)
261:             {
262:                 Console.WriteLine("\n\nYOU WON!!!");
263:             }
264:             else
265:             if( pTotal == cTotal )
266:             {
267:                 Console.WriteLine("\n\nIt's a push");
268:             }
269:             else
270:             {
271:                 Console.WriteLine("\n\nSorry, The Computer won");
272:             }
273:         }
274:     }
```

第2周复习

```
274:         }
275:
276:         Console.WriteLine("\n\nDo you want to play again? ");
277:         string answer = Console.ReadLine();
278:
279:         try
280:         {
281:             if( answer[0] != 'y' && answer[0] != 'Y' )
282:             {
283:                 //Quitting
284:                 playing = false;
285:             }
286:         }
287:         catch( System.IndexOutOfRangeException )
288:         {
289:             // Didn't enter a value so quit
290:             playing = false;
291:         }
292:     }
293: }
294:
295: // GetPlayerOption()
296: // Returns true to hit, false to stay
297: //=====
298:
299: static bool GetPlayerOption( int cardctr )
300: {
301:     string buffer;
302:     bool cont = true;
303:     bool retval = true;
304:
305:     while(cont == true)
306:     {
307:         Console.WriteLine("\n\nH = Hit, S = Stay ");
308:         buffer = Console.ReadLine();
309:
310:         try
311:         {
312:             if ( buffer[0] == 'h' || buffer[0] == 'H' )
313:             {
314:                 pHand[cardctr] = mydeck.dealCard();
315:                 cont = false;
316:             }
317:             else if( buffer[0] == 's' || buffer[0] == 'S' )
318:             {
319:                 // Turn is over, return false...
320:                 retval = false;
321:                 cont = false;
322:             }
323:             else
```

```

324:         {
325:             Console.WriteLine("\n*** Please enter an H or S and press
&ENTER...");
326:         }
327:     }
328:     catch( System.IndexOutOfRangeException )
329:     {
330:         // Didn't enter a value, so ask again
331:         cont = true;
332:     }
333: }
334:     return retval;
335: }
336: }
337: //----- END OF LISTING -----//

```

该程序清单的输出如下:

```

Shuffling cards...
Dealing cards...

Player's Hand:
    Card 1: four of clubs
    Card 2: six of hearts
Dealer's Hand:
    Card 1: Jack of hearts
-----
Player Total = 10
Dealer Total = 10

H = Hit, S = Stay h

Player's Hand:
    Card 1: four of clubs
    Card 2: six of hearts
    Card 3: King of diamonds
Dealer's Hand:
    Card 1: Jack of hearts
-----
Player Total = 20
Dealer Total = 10

H = Hit, S = Stay s

Player's Total: 20

Computer:
    Jack of hearts
    seven of diamonds TOTAL: 17

```


第2周复习

```
YOU WON!!!

Do you want to play again?

Shuffling cards...
Dealing cards...

Player's Hand:
  Card 1: three of clubs
  Card 2: Jack of spades
Dealer's Hand:
  Card 1: seven of clubs
-----
Player Total = 13
Dealer Total = 7

H = Hit, S = Stay h

Player's Hand:
  Card 1: three of clubs
  Card 2: Jack of spades
  Card 3: five of hearts
Dealer's Hand:
  Card 1: seven of clubs
-----
Player Total = 18
Dealer Total = 7

H = Hit, S = Stay s

Player's Total: 18

Computer:
  seven of clubs
  two of diamonds TOTAL: 9
  three of diamonds TOTAL: 12
  five of clubs TOTAL: 17

YOU WON!!!

Do you want to play again?
```

分析: 这里的输出是从一副洗过后的、标准的52张扑克牌中选择得到的, 因此您运行该程序时, 输出将不同。该程序并不是一个完整的21点游戏。例如, 它不指出您是不是blackjack (两张牌就21点), 也不保存历史记录——您赢过计算机多少次。您可以对其进行改进。

该程序清单使用了前面的 14 天中介绍过的许多概念。接下来的几节将对其中的一些部分进行分析。

用于表示扑克牌的枚举

第 8 天介绍了枚举。该程序使用枚举来简化对各张牌的处理。这里使用了两个枚举。第一个位于第 7~14 行，用于存储各种花色，为了简化，以数字方式表示各种花色，并将第一个位置设置为 `Zero_Error`。这样各种花色（从梅花开始）将分别指定 1~4 的值。也可以删除第 9 行，并将第 10 行改成如下所示，这样各种花色对应的值不变：

```
clubs = 1,
```

作者这里选择包含零位置，并将其作为错误值，如果需要，可以使用这种结构来创建其他扑克游戏。

第二个枚举用于表示牌的点数，它名为 `CardValue`，是在第 16~32 行定义的。这使得能够表示每一张牌。同样，这里也略过了零，并提供了一个占位符。这样 A 的编号将为 1，2 的编号为 2，等等。同样，将 A 的值设置为 1，并删除第 18 行，也可获得相同的编号方式。

```
Ace = 1
```

card 结构

`card` 是在第 36~64 行定义的，它被定义为结构，而不是类。也可以将 `card` 定义为一个类，但由于它较小，因此使用结构的效率将更高。

`card` 结构包含为数不多的几个成员。第 38 和 39 行创建了两个分别用于存储花色和值的成员变量。这些变量的类型是前面创建的枚举。另外，`card` 结构还包含一个属性，让您能够获得牌的点数，这里花牌的点数为 10（第 47~48 行），A 为 11（第 50 和 51 行），其他牌的点数与其编号相同（第 52 和 53 行）。

`card` 结构的最后一个成员是 `ToString` 方法。前一周介绍过，所有的类都是从基类 `Object` 派生而来的。`Object` 类包含大量派生类可以使用的方法，其中的一个是 `ToString`。第 11 天介绍过，可以通过关键字 `override`，用您自己的功能覆盖已有的方法。第 59~63 行覆盖了基类的 `ToString` 方法。

覆盖方法使用格式字符“G”，以可读性更强的方式打印各张牌的值，它打印的是枚举的文本值。第 12 天介绍了如何将格式字符用于枚举和其他数据类型。

deck 类

要玩牌，必须有一副牌。一个类被用来定义一副牌。如果有人问您应用哪种数据类型来存储一副牌，您可能说数组。虽然可以创建一个由 `card` 组成的数组（实际使用的是 `deck` 类），`deck` 类不仅仅是存储牌的信息。

对于一副牌，使用类更合适。因为除了存储牌外，还需要创建一些方法来处理这些牌。该程序清单中的 `deck` 类包含洗牌和发牌的方法。这个类还记录当前的位置以及其他信息。

`Deck` 类包含一个由 `card` 组成的数组（第 70 行），该数组中的各个 `card` 结构是在 `deck` 类的构造函数中初始化的（第 76~94 行）。这种初始化工作是通过遍历各种花色和牌值来完成的。实际赋值工作是在第 90 和 91 行完成的。数字型值被强制转换为 `CardValue` 或 `CardSuit` 类型，然后赋给 `deck` 类中 `cards` 数组中的 `card` 结构。

被赋值的 `card` 结构在数组中的位置是使用 `currcard` 来记录的。第 89 行的计算看起来好像有些奇怪,但它用来生成 1~52 的值。如果您理解了这一行,便会知道,每循环一次,计算出得到 `currcard` 的值就增加 1。

`deck` 类还包含用于洗牌的方法(第 99~118 行),第 108 和 109 行取得两个 1~52 的随机数。然后,第 111~113 行将这两个数对应的 `cards` 数组位置处的两张牌互换。这种操作的执行次数取决于从第 106 行开始的 `for` 循环。这里是 100 次,这足以将整幅牌洗乱。

CardGame 类

该程序的主要部分叫 `CardGame`。前面指出过,这是一个简化的 21 点游戏。您可以使用 `deck` 类及其方法来创建其他的扑克牌游戏,甚至可以创建使用多副牌的程序。

该程序清单包含大量的注释和显示语句,以帮助您理解其中的代码。对于 `CardGame`,作者将介绍一个主要的地方。第 143 和 144 行创建了用于存储玩家和计算机手中的牌的数组,它们被声明为最多可存储 10 张牌。很少出现手中有 5 张牌的情况,需要 10 张牌的概率非常低,因此这足够了。

`CardGame` 类中的大部分代码都简单易懂。第 279~291 行加入了异常处理。第 279 行的 `try` 语句检查数组 `answer` 的第一个字符,看它是否是 Y 或 y。如果是,则说明玩家还想再玩;如果不是,则认为玩家不想玩了。但如果用户在按 `Enter` 键之前,没有输入任何值,将会如何呢?在这种情况下,数组 `answer` 的第一个元素将不包含任何值,此时当您试图访问第一个字符时,将引发异常。这是一种 `IndexOutOfRangeException` 异常,第 287 行的 `catch` 语句将捕获这种异常。在询问玩家是否还要牌时,使用的逻辑与此类似。

查看整副牌

也可以通过遍历来查看整副牌。该程序清单没有这样做,但只需几行代码便可完成这样的工作:

```
deck aDeck = new deck();
card aHand;

for( int ctr = 1; ctr < 53; ctr++)
{
    aHand = aDeck.dealCard();
    Console.WriteLine(aHand.ToString());
}
```

上述代码声明了一个名为 `aDeck` 的 `deck` 对象,还声明了一个临时的、名为 `aHand` 的 `card` 变量,它保存从 `aDeck` 中发出的牌。然后使用一个 `for` 循环,将牌发给临时变量 `aHand`,并将其显示到屏幕上,来遍历整副牌。上述代码打印整副牌中的各张牌,而不论牌是否洗过。

总 结

该程序清单只使用了过去几天学习的一些复杂主题。使用 C# 语言中的基本结构,可以完成大量的工作。您还将发现,该程序清单中的部分代码是可以重用的,包括 `card` 和 `deck` 类。您可以使

用这些类来创建其他的扑克牌游戏。另外，将该程序清单与下一周将学习的知识结合起来，可以创建出图形界面，这样玩起来将更有趣，更容易。

虽然该程序清单不完美，但它确实使用不多的代码完成了大量的工作。您将发现，在编程时，您将想包含大量的注释，包括 XML 文档注释；同时还想包含比该程序清单更多的异常处理。

第三周课程



至此，您完成了两周的课程，还有一周的课程需要学习。在第 2 周，您学习了大量关于 C# 的细节。第 3 周将介绍一些您可使用的已有的代码，然后简要地介绍多个高级主题。

具体地说，本周的第 1 天将介绍基类库 (BCL)。BCL 中包含一组预定义的类和类型，您可以在程序中使用它们。第 15 天课程“使用 .NET 基类”将介绍如何操纵计算机目录、使用数学函数和文件。这些工作都是依赖 BCL 的帮助来完成的。

第 16 天课程“创建 Windows 窗体”和第 17 天课程“创建 Windows 应用程序”将介绍基于窗体编程的知识。您将学习如何创建并定制基本窗体以及如何在窗体中添加基本控件和功能，其中包括添加菜单和对话框。这两天的课程并不能包揽一切，仅仅介绍基于窗体编程的知识就需要一篇篇幅比本书还长的图书！这两天的课程旨在介绍一些如何将 C# 用于基于 Windows 编程的基本知识。

第 18 天课程“Web 开发”简要介绍 C# 在基于 Web 编程方面的用途。这里假设您具备一些 Web 方面的经验，如果您没有这方面的经验，将发现本天的课程难以理解。但也不必烦恼，市面上有专门介绍这方面主题的图书。

第 19 天课程“编译指令和调试技术”从使用基类库回到 C# 编程。您将学习一些关于调试 C# 应用程序的知识，另外，还将学习如何使用编译指令来决定程序中的哪些代码需要编译，哪些不编译。您将知道哪些编译指令可用来完成这种伪预处理。

第 20 天课程“重载运算符”介绍运算符重载，许多人认为这一主题非常复杂，但在 C# 中实现起来比较简单。前面介绍过如何重载方法，而第 20 天将介绍如何重载运算符！

第 21 天课程“反射”。在此之前，您已经对 C# 中的大部分关键主题有了基本的了解。这里将介绍 C# 中的几个高级主题，包括属性和反射。阅读完本书后，您将具备丰富的、开发 C# 应用程序的知识。

第 15 天课程

使用.NET 基类

在前面的 14 天中，介绍了如何创建自己的类型，包括类、接口、枚举等等。在此期间，您使用了 C# 类库中的大量类和类型。本周将重点介绍这些已有的基类。今天将介绍以下内容：

- 已有的基类库；
 - 名称空间及其组织结构；
 - 通过使用组件定时器、目录信息、系统环境、数学函数、文件和数据认识许多标准类型。
- 今天和接下来的两天将深入介绍微软公司编写的，并在一组库中提供的许多类和其他类型。

15.1 .NET 框架中的类

.NET 框架包含大量的类、枚举、结构、接口和其他数据类型。实际上，它们数以千计。您可以在 C# 程序中使用这些类。

今天将介绍其中的几种类型，今天的课程的大部分内容被组织成小型的范例程序清单，这些程序清单演示了如何使用大量不同的类。您很容易在自己的程序中扩展这些范例。

15.1.1 通用语言规范

.NET 框架中的类遵循了通用语言规范（Common Language Specification，或称 CLS）。在本书的开头，讨论运行阶段环境（runtime）时，已经提到过 CLS。

CLS 是一套规则，运行在 .NET 平台上的所有语言都必须遵守。这套规则还包含通用类型系统，这在第 3 天讨论基本数据类型时介绍过。由于遵守这套规则，所以不管程序使用的是什么语言语法，通用运行阶段环境都能执行它。

遵循 CLS 的优点在于，使用一种语言编写的代码可以被另一种语言调用。由于 .NET 框架中的函数遵循了 CLS，因此它们不但可被 C# 使用，还可以被任何 CLS-顺应的语言（如 Visual Basic.NET 和 JScript.NET）使用。

注意：有 20 多种语言可以使用 .NET 框架中的代码。每种语言调用 .NET 框架中代码的方式可能稍有不同，但代码实现的功能完全相同。

15.1.2 用于组织类型的名称空间

.NET 框架中的代码是以名称空间的方式进行组织的。NET 框架中包含成百上千的名称空间，它们被用来组织成千上万的类和其他类型。

一些名称空间被存储在其他名称空间中。例如，您使用过的 `DateTime` 类型位于名称空间 `System` 中，而 `Random` 类型也位于该名称空间中。许多输入/输出类型被存储在 `System` 名称空间中的 `System.IO` 名称空间中。许多操纵 XML 数据的函数则位于名称空间 `System.XML` 中。关于 .NET 框架中完整的名称空间列表，可参阅在线文档。

15.1.3 ECMA 标准

并非名称空间中的所有类型都需要与其他语言兼容。另外，其他公司开发的 C# 开发工具也不一定包含相应的代码例程。

在开发 C# 时，微软公司向同一个被授权将 C# 标准化的标准委员会提交了大量的类型。通过将这类提交给标准委员会，其他开发商便可以开发使用同样的名称空间和类型的 C# 工具和编译器。这样，使用微软公司的工具创建的代码将与其他公司的工具兼容。

注意：编写本书时，只有微软公司有 C# 编译器和运行阶段环境。通过将 C# 语言和基类库提交给标准委员会，其他人和公司便能够开发 C# 工具——包括编译器和运行阶段环境。

被标准化的类位于 `System` 名称空间中。其他名称空间中包含的类则没有标准化。如果类不是标准的一部分，则不一定所有支持 C# 的操作系统和运行阶段环境都支持它。例如，微软公司的 SDK 中包含多个名称空间，这包括 `Microsoft.VisualBasic`、`Microsoft.CSharp`、`Microsoft.JScript` 和 `Microsoft.Win32`。这些名称空间不是 ECMA 标准的一部分，因此并非所有的开发环境中都有这些名称空间。

注意：关于 ECMA 和 C# 标准的信息，可以在 [Msdn.Microsoft.com/net/ecma](http://msdn.microsoft.com/net/ecma) 找到。

15.1.4 查看 .NET 框架类

基类库中包含成千上万的类和其他类型，要介绍所有这些类型，需要几本这么厚的书。在编写自己的程序之前，应花些时间来查看在线文档，看是不是已经有类似的功能。今天介绍的所有类和其他类型都是提交给 ECMA 的标准的一部分。

注意：您不但可以直接使用类库中的类型，还可以扩展它们。

15.2 使用定时器

程序清单 15.1 是一个非常小的程序，设计得并不好。该程序很简单，没有任何新的东西。

程序清单 15.1 `timer.cs`：显示时间

```
1: // Timer01.cs - Displaying Date and Time
2: //     Not a great way to do the time.
3: //     Press Ctrl+C to end program.
4: //-----
5: using System;
6:
7: class myApp
8: {
9:     public static void Main()
10:    {
11:        while (true)
12:        {
```

```

13:         Console.Write("\r{0}", DateTime.Now);
14:     }
15: }
16: }

```

该程序清单的输出如下:

5/26/2001 9:34:19 PM

分析: 从输出可以知道, 该程序的执行时间是5月26日晚上9:34。该程序清单提供了一个命令行时钟。该时钟好像是每秒更新一次, 但实际上要频繁得多, 不过仅当显示的值发生变化时, 您才会认为时间变了, 而这正好是每秒一次。该程序将不断运行, 直到您按下Ctrl + C键终止它。

今天课程的重点是使用基类库中的类和其他类型。第13行调用使用了DateTime, DateTime是一个结构, 位于基类库的System名称空间中。该结构包含一个名为Now的静态属性, 它返回当前的时间。DateTime还包含很多其他的数据成员和方法, 有关信息, 请参阅.NET框架类库文档。

新术语: 在屏幕上显示日期的更佳方式是使用定时器。定时器使得在特定的时间或经过一定的时间后, 某种处理(以代表的方式)被调用。.NET框架包含一个定时器类, 它位于名称空间System.Timers中, 这个类名为Timer。程序清单15.2使用Timer类重写了程序清单15.1。

程序清单 15.2 Timer02.cs: 使用定时器定期更新显示的日期和时间

```

1: // Timer02.cs - Displaying Date and Time
2: //     Using the Timer class.
3: //     Press Ctrl+C or 'q' followed by Enter to end program.
4: //-----
5: using System;
6: using System.Timers;
7:
8: class MyApp
9: {
10:     public static void Main()
11:     {
12:         Timer myTimer = new Timer();
13:         myTimer.Elapsed += new ElapsedEventHandler( DisplayTimeEvent );
14:         myTimer.Interval = 1000;
15:         myTimer.Start();
16:
17:
18:         while ( Console.Read() != 'q' )
19:         {
20:             ; // do nothing...
21:         }
22:     }
23:
24:     public static void DisplayTimeEvent( object source, ElapsedEventArgs e )
25:     {
26:         Console.Write("\r{0}", DateTime.Now);
27:     }
28: }

```


该程序清单的输出如下：

5/26/2001 10:04:13 PM

分析：该程序清单的输出与15.1类似，但其运行方式更佳。它不是不断地更新显示的日期和时间，而是每隔1000次滴答（即1秒）更新一次。

仔细查看该程序清单，您将知道定时器是如何工作的。第 12 行新建了一个 Timer 对象。第 14 行设置了时间间隔。第 13 行指定了定时器的时间间隔被设置后将执行的方法，这里是 DisplayTimeEvent，该方法是在第 24 ~ 27 行定义的。

第 15 行调用了 Start 方法，该方法开始记时。Timer 类的另一个成员是 AutoReset，如果将其默认值 true 修改为 false，则 Timer 事件将只发生一次。如果保留 AutoReset 的默认值 true 或将其设置为 true，则每当经过指定的时间间隔后，Timer 都将引发一个事件，从而执行该方法。

第 18 ~ 21 行包含一个循环，该循环将不断运行，直到用户输入字母 q 并按下 Enter 键。然后到达例程的结尾，程序将结束。该程序中的循环不执行任何操作，如果愿意，您也可以在循环中完成其他处理工作。在该循环中无需调用 DisplayTimeEvent，因为每当相应的时间过去后，该方法都将自动被调用。

该定时器用于将时间显示到屏幕上。定时器和定时器事件也可用于许多其他的程序中。您可以创建一个在指定的时间启动程序的定时器。也可以创建一个每隔一定时间对重要数据进行复制的备份程序。还可以创建一个这样的例程，即如果用户在一定的时间内没有任何活动，则注销用户或结束程序。使用定时器的方式数不胜数。

注意：程序清单15.2使用的事件名称与第14天介绍的稍有不同。这种名称是第14天介绍的例程的定制版本。

15.3 获取目录和系统环境信息

对于程序而言，有大量关于运行该程序的计算机的信息可用，如何选择使用这些信息，取决于程序员。程序清单 15.3 显示了计算机及其环境的信息，这是通过使用 Environment 类实现的，这个类包含大量您将感兴趣的静态数据成员。

程序清单 15.3 env01.cs: 使用 Environment 类

```
1: // env01.cs - Displaying information with the
2: //           Environment class
3: //-----
4: using System;
5:
6: class MyApp
7: {
8:     public static void Main()
9:     {
10:        // Some Properties...
11:        Console.WriteLine("=====");
12:        Console.WriteLine(" Command: {0}", Environment.CommandLine);
13:        Console.WriteLine("Curr Dir: {0}", Environment.CurrentDirectory);
```

```

14: Console.WriteLine(" Sys Dir: {0}", Environment.SystemDirectory);
15: Console.WriteLine(" Version: {0}", Environment.Version);
16: Console.WriteLine(" OS Vers: {0}", Environment.OSVersion);
17: Console.WriteLine(" Machine: {0}", Environment.MachineName);
18: Console.WriteLine(" Memory: {0}", Environment.WorkingSet);
19:
20: // Some methods...
21: Console.WriteLine("=====");
22: string [] args = Environment.GetCommandLineArgs();
23: for ( int x = 0; x < args.Length; x++ )
24: {
25:     Console.WriteLine("Arg {0}: {1}", x, args[x]);
26: }
27:
28: Console.WriteLine("=====");
29: string [] drives = Environment.GetLogicalDrives();
30: for ( int x = 0; x < drives.Length; x++ )
31: {
32:     Console.WriteLine("Drive {0}: {1}", x, drives[x]);
33: }
34:
35: Console.WriteLine("=====");
36: Console.WriteLine("Path: {0}",
37:     Environment.GetEnvironmentVariable("Path"));
38: Console.WriteLine("=====");
39:
40: }
41: }

```

在作者的笔记本电脑中运行该程序清单时，得到的输出如下：

```

=====
Command: C:\MYDOCU~1\BOOKS\98-CODE\DAY15\ENV01.EXE
Curr Dir: C:\My Documents\Books\98-code\Day15
Sys Dir: C:\WINDOWS\SYSTEM
Version: 1.0.2914.16
OS Vers: Microsoft Windows 98 4.90.73010104.0
Machine: HP-P111
Memory: 0
=====
Arg 0: C:\MYDOCU~1\BOOKS\98-CODE\DAY15\ENV01.EXE
=====
Drive 0: A:\
Drive 1: C:\
Drive 2: D:\
=====
Path:
C:\WINDOWS;C:\WINDOWS\COMMAND;C:\WINDOWS\MICROS~1.NET\FRAMEW~1\V10~1.291
;
=====

```

在作者的台式机中运行该程序清单时，得到的输出如下：

```
=====
Command: ENV01
Curr Dir: C:\WORKAREA\DAY15
Sys Dir: C:\WINDOWS\System32
Version: 1.0.2914.16
OS Vers: Microsoft Windows NT 5.1.2462.0
Machine: HPBETA30
Memory: 3629056
=====

Arg 0: ENV01
Arg 1: aaa
Arg 2: bbbbbb
Arg 3: ccccc

=====

Drive 0: A:\
Drive 1: C:\
Drive 2: D:\
Drive 3: E:\

=====

Path: C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\PRO-
GRA~1\MICROS
~2\80\Tools\BINN
=====
```

分析：Environment类的工作原理非常简单，它包含大量的静态成员，提供关于用户系统的信息。作者在两台不同的机器上运行了该应用程序。第一台机器是作者的笔记本电脑，它安装的是Windows ME（虽然输出中指出的是Windows 98），包含三个驱动器：A、C和D。输出中指出了当前目录和系统目录以及驱动器和其他目录信息。

第二台机器安装的是Windows XP（NT 5），输出中还包含关于该机器的其他信息。不同的是，在这台机器上运行时，提供了三个命令行参数：aaa、bbbbbb和ccccc。

大多数信息都可以使用Environment类的静态成员来获取。第12~18行使用了多个静态成员。这个类包含一些返回字符串数组的方法，其中包括返回命令行参数的方法GetCommandLineArgs和GetLogicalDrives。该程序清单使用简单的循环来打印这些字符串数组中的值。第22~26行打印命令行参数，而第29~33行打印有效的驱动器。

Environment类还包含其他一些您可能感兴趣的方法。GetEnvironmentVariable取得当前系统的环境变量及其值，可以使用该方法来取得当前系统的某个环境变量的值。

15.4 使用数学函数

基本的数学运算符（如加、减、求模）提供的功能有限。您迟早将发现您需要更强大的数学函数。C#可以使用基类中的一组数学函数，这些函数位于名称空间System.Math中。表15.1列出了大量数学函数。

`Math` 类是密封的。前面介绍过，密封类不能被继承；另外，其所有的方法和数据成员都是静态的。因此不能创建 `Math` 对象，而只能通过类名来使用其数据成员和方法。

表 15.1 `Math` 类中的数学函数

方 法	描 述
<code>Abs</code>	返回绝对值
<code>Ceiling</code>	返回不小于给定数字的最小整数
<code>Exp</code>	返回 E 的幂，是 <code>Log</code> 的逆运算
<code>Floor</code>	返回不大于给定数值的最大整数
<code>IEEERemainder</code>	返回两个数相除的结果，这种除法运算遵循 ANSI/IEEE 标准 754-1985 的 5.1 节规定的求余运算
<code>Log</code>	返回自然对数
<code>Log10</code>	返回以 10 为底的对数
<code>Max</code>	返回两个值中较大的一个
<code>Min</code>	返回两个值中较小的一个
<code>Pow</code>	返回给定值的给定次幂
<code>Round</code>	返回四舍五入后的值，您可以指定精度。数字 5 将被舍为 0
<code>Sign</code>	返回值的正负情况。对于负数，返回 -1；对于 0，返回 0；对于正数，返回 1
<code>Sqrt</code>	返回平方根
<code>Acos</code>	返回反余弦
<code>Asin</code>	返回反正弦
<code>Atan</code>	返回反正切
<code>Atan2</code>	返回两个数的商的反正切
<code>Cos</code>	返回余弦
<code>Cosh</code>	返回双曲余弦
<code>Sin</code>	返回正弦
<code>Sinh</code>	返回双曲正弦
<code>Tan</code>	返回正切
<code>Tanh</code>	返回双曲正切

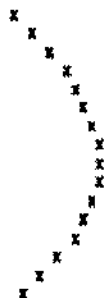
`Math` 类还包含两个常量：`PI` 和 `E`，前者返回圆周率的值——3.14159265358979323846；后者返回欧拉常数——2.7182818284590452354。

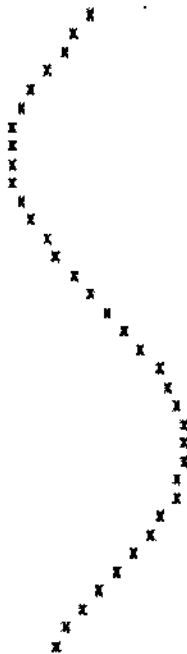
表 15.1 中的大多数数学函数都很容易理解。程序清单 15.4 演示了两个函数的用法。

程序清单 15.4 Math.cs: 使用一些数学函数

```
1: // math.cs - Using a Math routine
2: //-----
3: using System;
4:
5: class myMathApp
6: {
7:     public static void Main()
8:     {
9:         int val2;
10:        char disp;
11:
12:        for (double ctr = 0.0; ctr <= 10; ctr += .2)
13:        {
14:            val2 = (int) Math.Round( { 10 * Math.Sin(ctr)} ) ;
15:            for( int ctr2 = -10; ctr2 <= 10; ctr2++ )
16:            {
17:                if (ctr2 == val2)
18:                    disp = 'X';
19:                else
20:                    disp = ' ';
21:
22:                Console.Write("{0}", disp);
23:            }
24:            Console.WriteLine(" ");
25:        }
26:    }
27: }
```

该程序清单的输出如下:





该程序清单演示了如何使用 Sin 方法。第 12~25 行的 for 语句将一个 double 变量从 0.0 增加到 10.0, 每次增加 0.2。第 14 行使用 Math.Sin 函数计算该变量的正弦值。由于正弦值在 -1.0 到 1.0 之间, 为简化显示, 将正弦值转换为 -10 到 10 之间的值。这种转换是通过将正弦值乘以 10, 然后使用 Math.Round 取整实现的。

执行乘法和四舍五入运算后, val2 的值将位于 -10 到 10 之间。第 15 行的 for 循环显示一行字符, 在该行中, 除了 val2 对应的位置为 X 外, 其他位置都为空格。第 24 行打印另一个空格, 以开始新的一行。最后的结果是, 显示了一条近似的正弦曲线。

15.5 使用文件

读写文件信息的功能将使程序的用途更多。另外, 很多时候, 您希望能够使用已有的文件。接下来的几节将简要地介绍一些基本的文件操纵特性, 然后解释一个重要的文件概念——流。

15.5.1 复制文件

基类库中有一个名为 File 的文件类, 它位于名称空间 System.IO 中。File 类包含大量的静态方法, 可用于操纵文件。实际上, File 中的所有方法都是静态的。表 15.2 列出了许多重要的方法。

表 15.2 File 类中的方法

方 法	描 述
AppendText	将文本追加到文件中
Copy	使用已有的文件新建一个文件

续表

方 法	描 述
Create	在指定位置创建一个文件
CreateText	新建一个可存储文本的文件
Delete	删除指定位置的一个文件。该文件必须存在, 否则将引发异常
Exists	确定指定的位置是否存在指定的文件
GetAttributes	返回指定文件的属性信息, 包含文件是否是压缩的、是否是目录名、是否是隐藏或只读的、是否是系统文件、是否是临时文件等
GetCreationTime	返回文件创建的日期和时间
GetLastAccessTime	返回文件最后一个被访问的日期和时间
GetLastWriteTime	返回文件最后一次被写入的日期和时间
Move	让文件可以移到新的位置并可以重命名
Open	打开指定位置的一个文件。打开文件后, 您便可以将信息写入文件或读取文件中的信息
OpenRead	创建一个只读的文件
OpenText	打开一个文件, 然后便可以将其作为文本读取
OpenWrite	打开指定的文件, 以便写入
SetAttributes	设置指定文件的属性
SetCreationTime	设置文件的创建日期和时间
SetLastAccessTime	设置文件最后一次被访问的日期和时间
SetLastWriteTime	设置文件最后一次被写入的日期和时间

程序清单 15.5 是一个使用 File 类来创建文件副本的小型程序。

程序清单 15.5 Filecopy.cs: 复制文件

```

1: // filecopy.cs - Copies a file
2: //-----
3: using System;
4: using System.IO;
5:
6: class MyApp
7: {
8:     public static void Main()
9:     {
10:         string[] CLA = Environment.GetCommandLineArgs();
11:
12:         if ( CLA.Length < 3 )
13:         {

```

```
14:         Console.WriteLine("Format: {0} orig-file new-file", CLA[0]);
15:     }
16:     else
17:     {
18:         string origfile = CLA[1];
19:         string newfile = CLA[2];
20:
21:         Console.Write("Copy...");
22:
23:         try
24:         {
25:             File.Copy(origfile, newfile);
26:         }
27:
28:         catch (System.IO.FileNotFoundException)
29:         {
30:             Console.WriteLine("\n{0} does not exist!", origfile);
31:             return;
32:         }
33:
34:         catch (System.IO.IOException)
35:         {
36:             Console.WriteLine("\n{0} already exists!", newfile);
37:             return;
38:         }
39:
40:         catch (Exception e)
41:         {
42:             Console.WriteLine("\nAn exception was thrown trying to copy file.");
43:             Console.WriteLine();
44:             return;
45:         }
46:
47:         Console.WriteLine("...Done");
48:     }
49: }
50: }
```

该程序清单的输出如下：

Copy.....Done

分析：上述输出是使用下面的命令运行该程序得到的结果：

filecopy filecopy.cs filecopy.bak

上述命令执行之前，filecopy.cs 已经存在，而 filecopy.bak 不存在。程序执行后，filecopy.bak 将被创建，因此将存在。如果再次执行上述命令（此时 filecopy.bak 已经存在），输出将如下：

Copy...
filecopy.bak already exists!

如果执行该程序时，没有提供任何参数，或者只提供一个参数，则输出将如下：

```
Format: C:\MyDocu-1\BOOKS\98-CODE\DAY15\FILECOPY.EXE orig-file new-file
```

最后，有必要看一下如果要复制的文件不存在，则该程序的输出：

```
Copy...  
BadFileName does not exist!
```

从上述输出可以知道，程序清单 15.5 做了大量的工作，以便对各种可能出现的情况都做出反应。这是通过编程逻辑和异常处理实现的。

来看看程序清单。第 4 行将名称空间 `System.IO` 包含了进来，这样使用 `File` 类时无需使用全限定名。第 10 行是 `Main` 方法的第一个关键行，它使用前面介绍过的 `Environment` 类的方法获取命令行参数。

第 12 行检查命令行参数变量 `CLA` 中是否至少包含三个值。如果不是，则说明用户没有提供足够的信息。记住，使用 `GetCommandLineArgs` 时，程序名将被视为第 1 个参数，命令行中的其他值依次被视为第 2、3... 个参数。这意味着需要三个值来获取程序名、原来的文件和新创建的文件。如果少于三个值，则第 14 行告诉将用法告诉给用户，其中包含 `GetCommandLineArgs` 方法获取的程序名称。

提示：使用 `GetCommandLineArgs` 的价值在于，提供了用户实际执行的程序的名称。这样在“用法”消息中包含的将是实际的程序名而不是硬代码值。这样做的好处是，当程序 `filecopy` 被重命名后，提供的用法信息仍将是正确的——它提供的是实际被执行的程序的名称。

如果用户提供了必要的参数，则对文件进行处理。第 18 和 19 行将命令行中的信息赋给文件变量，这些变量的名称更容易理解。从技术上讲，不需要这样做，但这可以提高程序中其他代码的可读性。

第 21 行向用户显示一条简单消息，指出已经开始复制。第 25 行使用 `File` 类的 `Copy` 方法复制文件。正如您看到的，复制文件非常简单。

虽然 `Copy` 的用法很简单，但注意到该程序清单的做法很重要。复制是在异常处理逻辑中进行的，这包括第 23 行的 `try` 语句以及接下来的三个 `catch` 语句。由于操作文件时，很多情况下都会发生错误，因此确保程序对这些情况做出合适的反应至关重要。为此，最佳的方式是在程序中包含异常处理。

对于 `File` 类的大部分方法可能发生的重要错误，都定义了异常。查看某个类的在线文档时，您将发现其中包含为其方法定义的各种异常。在可能发生异常时包含异常处理是一个非常好的编程习惯。

第 28 行是 `Copy` 方法调用对应的第一个异常处理程序，该异常在要复制的文件不存在时引发，它被相应地命名为 `FileNotFoundException`。

注意：在该程序清单中，引用异常时，使用的是全限定名。由于已经包含了名称空间 `System.IO`，因此您可以删除异常名中的 `System.IO`。

第 34 行捕获 `IOException` 异常。该异常在许多其他异常发生时引发，这包括目录不存在（`DirectoryNotFoundException`）、到达文件末尾（`EndOfStreamException`）、装载文件时出现问题（`FileLoadException`）或文件找不到。最后一种异常将被第一个 `catch` 语句捕获。当要创建的文件已经存在时，也将引发 `IOException` 异常。

最后，第 40 行使用标准的通用异常捕获其他意想不到的错误。由于不知道引发这种异常的原因，因此这里提供一条通用性消息。

如果未出现异常，文件将成功复制。第 47 行显示一条消息，指出这一点。

应该	不应该
使用 File 类的方法时，一定要加入异常处理	使用命令行参数时，不要认为用户将提供您所需的一切

15.5.2 获取文件信息

除了 File 类外，还可以使用 FileInfo 类来处理文件。程序清单 15.6 演示了如何使用 FileInfo 类，该程序接受一个文件名，然后显示该文件的大小以及该文件涉及到的一些重要的日期。输出是关于文件 filesize.cs 的。

程序清单 15.6 filesize.cs: 使用 FileInfo 类

```

1: // fileinfo.cs ~
2: //-----
3: using System;
4: using System.IO;
5:
6: class myApp
7: {
8:     public static void Main()
9:     {
10:         string[] CLA = Environment.GetCommandLineArgs();
11:
12:         FileInfo fiExe = new FileInfo(CLA[0]);
13:
14:         if ( CLA.Length < 2 )
15:         {
16:             Console.WriteLine("Format: {0} filename", fiExe.Name);
17:         }
18:         else
19:         {
20:             try
21:             {
22:                 FileInfo fiFile = new FileInfo(CLA[1]);
23:
24:                 if(fiFile.Exists)
25:                 {
26:                     Console.WriteLine("=====");
27:                     Console.WriteLine("{0} - {1}", fiFile.Name, fiFile.Length);
28:                     Console.WriteLine("=====");
29:                     Console.WriteLine("Last Access: {0}", fiFile.LastAccessTime);
30:                     Console.WriteLine("Last Write: {0}", fiFile.LastWriteTime);
31:                     Console.WriteLine("Creation: {0}", fiFile.CreationTime);
32:                     Console.WriteLine("=====");
33:                 }
34:                 else
35:                 {

```

```

36:         Console.WriteLine("{0} doesn't exist!", fiFile.Name);
37:     }
38: }
39:
40:     catch (System.IO.FileNotFoundException)
41:     {
42:         Console.WriteLine("\n{0} does not exist!", CLA[1]);
43:         return;
44:     }
45:     catch (Exception e)
46:     {
47:         Console.WriteLine("\nAn exception was thrown trying to copy file.");
48:         Console.WriteLine();
49:         return;
50:     }
51: }
52: }
53: }

```

该程序清单的输出如下：

```

=====
filesize.cs - 1547
=====
Last Access: 5/27/2001 12:00:00 AM
Last Write: 5/27/2001 2:57:34 PM
Creation: 5/27/2001 2:57:32 PM
=====

```

该程序清单与前一个类似，使用 `FileInfo` 类创建了一个与指定的文件相关联的对象。第 12 行创建了一个名为 `fiExe` 的 `FileInfo` 对象，并将其与被执行的程序（`filesize.exe`）关联起来。如果用户在命令行中没有输入参数，在第 14 行打印用法信息，其中包含 `fiExe` 的值。

第 22 行创建了另一个 `FileInfo` 对象，并将其与传递给程序的参数关联起来。第 26~32 行显示关于该文件（命令行参数指定的文件）的信息。

15.6 使用数据文件

能够获取关于文件的信息以及复制文件已经很不错了，但读写文件的用途更大。

15.6.1 理解流

术语文件通常与存储在磁盘驱动器或内存中的信息相关。使用文件时，通常要通过流。许多人不知道流和文件之间的差别，流指的是信息流，它不一定与文件相关，也不一定非得是文本。

流可用于向内存、网络、Web、字符串等发送信息或接收来自这些地方的信息。流也可用于向文件发送信息和接收来自文件的信息。

15.6.2 读取文件的步骤

读写文件时，需要按一定的步骤进行。首先需要打开文件。如果是创建新文件，则在创建该文

件的同时，应该打开它。文件打开后，您需要使用流来将信息加入到文件中或从文件中取出信息。创建流时，需要指明信息的流动方向。将流与文件关联起来后，便可以开始读写数据了。如果是读取文件中的信息，则可能需要检查是否到了文件的末尾。读写完信息后，需要关闭文件。

读写文件的基本步骤

1. 打开/创建文件；
2. 建立从文件中取出信息/向文件写入信息的流；
3. 将信息加入到文件中或从文件中读取信息；
4. 关闭流/文件。

15.6.3 用于创建和打开文件的方法

存在各种类型的流。您使用的流和方法将随文件中的数据类型而异。本节将重点介绍读写文本信息；下一节将介绍如何读写二进制信息。二进制信息能够存储数字型值和其他任何数据类型。

要打开磁盘文件以读写文本，可以使用 `File` 或 `FileInfo` 类。这两个类中包含多个可用于读写文件的方法，其中包括：

- `AppendText`：打开一个可追加文本的文件（创建一个用于追加文本的 `StreamWriter`）；
- `Create`：新建一个文件；
- `CreateText`：创建并打开一个可存储文本的文件（实际上是创建一个 `StreamWriter`）；
- `Open`：打开一个文件，以便进行读写（实际上是打开一个 `FileStream`）；
- `OpenRead`：打开一个文件，以便读取；
- `OpenText`：打开一个已有的文件，以便读取（实际上是创建一个 `StreamReader`）；
- `OpenWrite`：打开一个文件，以便读写。

如果它们包含类似的方法，您如何确定何时应该使用 `File` 类，而不是 `FileInfo` 类呢？这两个类是不同的。`File` 类包含的方法都是静态的，另外还将自动检查文件的权限；`FileInfo` 类用于创建 `FileInfo` 实例。如果在程序中只需打开文件一次，则应该使用 `File` 类；如果要多次使用同一个文件，则应该使用 `FileInfo` 类。拿不准时，应该使用 `FileInfo` 类。

15.6.3.1 写文本文件

学习如何使用文件的最佳方式是看代码，程序清单 15.7 创建一个文本文件，然后将信息写入到该文件中。

程序清单 15.7 Writing.cs: 写文本文件

```
1: // writing.cs - Writing to a text file.
2: // Exception handling left out to keep listing short.
3: //-----
4: using System;
5: using System.IO;
6:
7: public class WritingApp
8: {
9:     public static void Main(String[] args)
10:    {
11:        if( args.Length < 1 )
12:        {
```

```
13:         Console.WriteLine("Must include file name.");
14:     }
15:     else
16:     {
17:         StreamWriter myFile = File.CreateText(args[0]);
18:
19:         myFile.WriteLine("Mary Had a Little Lamb,");
20:         myFile.WriteLine("Whose Fleece Was White as Snow.");
21:
22:         for ( int ctr = 0; ctr < 10; ctr++ )
23:             myFile.WriteLine ("{0}", ctr);
24:
25:         myFile.WriteLine("Everywhere that Mary Went,");
26:         myFile.WriteLine("That Lamb was sure to go.");
27:
28:         myFile.Close();
29:     }
30: }
31: }
```

输出: 运行该程序清单时, 除非提供文件名, 否则将看不到任何输出。您需要提供一个文件名作为参数。这样, 该文件将被创建, 并将包含以下内容:

```
Mary Had a Little Lamb,
Whose Fleece Was White as Snow.
0
1
2
3
4
5
6
7
8
9
Everywhere that Mary Went,
That Lamb was sure to go.
```

分析: 该程序清单中没有包含异常处理, 这意味着它可能引发没有被处理的异常。之所以没有包含异常处理, 是为了让您能够将重点放在操纵文件的方法上, 同时缩短程序清单。

来看一下程序清单。第 11 ~ 14 行检查用户是否将文件名用作了命令行参数。如果没有, 则显示一条错误消息; 如果有, 执行第 17 行以后的代码。

第 17 行调用了 `File` 类的 `CreateText` 方法来新建一个名为 `myFile` 的 `StreamWriter` 对象, 传递的参数是要创建的文件的名称, 这样将创建一个可存储文本的文件。要存储的文本将通过名为 `myFile` 的 `StreamWriter` 发送给该文件。图 15.1 说明了该语句的功能。

警告: 如果用户提供的命令行参数指定的文件已存在, 则原来的文件将被覆盖。

建立流并将其指向文件后, 便可以将信息写入到流中, 从而写入到文件中。第 19 行表明将信息写入到流的方式与写入到控制台相同, 当然使用的是流名——这里为 `myFile`, 而不是 `Console`。第

19 和 20 行调用 `WriteLine` 方法将句子写入到流中。第 22 ~ 23 行将数字写入到流中，这些数字是作为文本写入的。最后第 25 和 26 行将另外两行文本写入到文件中。

写完文件后，需要关闭流。第 28 行通过调用 `Close` 方法将流关闭。

使用文件的步骤与该范例完全相同。



图 15.1 使用流来写文件

15.6.3.2 从文件中读取文本

读取文本文件中的信息与写入信息极其类似。程序清单 15.8 可用于读取程序清单 15.7 创建的文件，该程序读取的是文本数据。

程序清单 15.8 reading.cs: 读取文本文件

```

1: // reading.cs - Read text from a file.
2: // Exception handling left out to keep listing short.
3: //-----
4: using System;
5: using System.IO;
6:
7: public class ReadingApp
8: {
9:     public static void Main(String[] args)
10:    {
11:        if( args.Length < 1 )
12:        {
13:            Console.WriteLine("Must include file name.");
14:        }
15:        else
16:        {
17:            string buffer;
18:
19:            StreamReader myFile = File.OpenText(args[0]);
20:
21:            while ( (buffer = myFile.ReadLine()) != null )
22:            {
23:                Console.WriteLine(buffer);
24:            }
25:
26:            myFile.Close();
27:        }
28:    }
29: }

```

该程序清单的输出如下：

```
Mary Had a Little Lamb,
Whose Fleece Was White as Snow.
0
1
2
3
4
5
6
7
8
9
Everywhere that Mary Went,
That Lamb was sure to go.
```

分析：第17行声明了一个字符串变量buffer，用于存储从文件中读取的信息。第19行与前一个程序清单的第17行类似，只不过使用的是File类的OpenText方法，而不是CreateText方法，它打开传递给程序的文件名（arg[0]）对应的文件。同样，将该文件与一个流关联起来。第21行使用一个while循环遍历该文件。ReadLine方法用于从myFile流中读取一行文本，直到读取的一行为null为止。null表明达到了文件的末尾。

每读取一行，便将该行打印到控制台（第23行）。读取文件中的所有行后，第26行将该文件关闭。

15.6.3.3 将二进制信息写入到文件中

使用文本文件时，必须将所有的数字转换为文本。很多时候，如果能够直接将数值写入到文件中，并从文件中直接读取将更佳。例如，如果您将一系列的整数作为整数写入到文件中，便可以将它们作为整数从文件中取出。如果将其作为文本写入，则必须从文件中读取文本并将其从字符串转换为整数。可以将二进制流（BinaryStream）与文件关联起来，然后通过该流读写二进制信息，而无需额外完成转换的步骤。

程序清单 15.9 将二进制数据写入到文件中。虽然这里是 100 个整数写入到文件中，但写入其他数据类型也将这样容易。

注意：二进制信息指的是保持其数据类型的存储格式，而不被转换为文本的信息。

程序清单 15.9 binwrite.cs：将二进制信息写入到文件中

```
1: // binWrite.cs -
2: // Exception handling left out to keep listing short.
3: //-----
4: using System;
5: using System.IO;
6:
7: class MyStream
8: {
9:     public static void Main(String[] args)
10:    {
```

```

11:     if( args.Length < 1 )
12:     {
13:         Console.WriteLine("Must include file name.");
14:     }
15:     else
16:     {
17:         FileStream myFile = new FileStream(args[0], FileMode.CreateNew);
18:         BinaryWriter bwFile = new BinaryWriter(myFile);
19:
20:         // Write data to Test.data.
21:         for (int i = 0; i < 100 ; i++)
22:         {
23:             bwFile.Write(i );
24:         }
25:
26:         bwFile.Close();
27:         myFile.Close();
28:     }
29: }
30: }

```

分析：执行该程序时，需要提供一个文件名命令行参数。如果提供了，输出将不会被写入到控制台，信息将被写入到文件中。如果您查看该文件，显示的将是扩展字符，而不是数字。

该程序清单中也没有包含异常处理。如果您试图将信息写入到一个已有的文件中，第 17 行将引发异常。在该程序清单中，打开文件的方式不同于打开文本文件。第 17 行创建了一个名为 myFile 的 FileStream 对象，并使用 FileStream 的构造函数将该文件流与一个文件关联起来。该构造函数的第一个参数是要创建的文件的名称（args[0]）；第二个参数是文件的打开模式，它是枚举 FileMode 的一个值，这里是 CreateNew，这意味着将新建一个文件。表 15.3 列出了 FileMode 枚举中的其他模式值。

表 15.3 **FileMode 枚举的值**

值	定 义
Append	打开已有的文件或新建一个文件
Create	新建一个文件，如果文件已经存在，则删除该文件，然后新建这样的文件
CreateNew	新建一个文件，如果文件已经存在，将引发异常
Open	打开已有的文件
OpenOrCreate	打开一个文件，如果该文件不存在，则创建它
Truncate	打开一个已有的文件并删除其内容

创建 FileStream 后，需要将其设置为能够操纵二进制数据。第 18 行通过将其与一种能够用于将二进制数据写入到流中的类型——BinaryWriter 关联起来，来完成了这项工作，它创建了一个名为 bwFile 的 BinaryWriter，并将 myFile 传递给 BinaryWriter 的构造函数，从而将 myFile 与 bwFile 关联起来。

第 23 行表明, 可以使用 Write 方法直接将信息写入到 BinaryWriter 对象 bwFile 中。被写入的数据可以是特定的数据类型, 这里是整数。写完文件后, 需要将打开的流关闭。

15.6.3.4 读取文件中的二进制信息

将二进制数据写入到文件中后, 您和可能想读取该文件。程序清单 15.10 包含一个从文件中读取二进制信息的程序。

程序清单 15.10 binread.cs: 读取二进制信息

```

1: // binRead.cs -
2: // Exception handling left out to keep listing short.
3: //-----
4: using System;
5: using System.IO;
6:
7: class MyStream
8: {
9:     public static void Main(String[] args)
10:    {
11:        if( args.Length < 1 )
12:        {
13:            Console.WriteLine("Must include file name.");
14:        }
15:        else
16:        {
17:            FileStream myFile = new FileStream(args[0], FileMode.Open);
18:            BinaryReader brFile = new BinaryReader(myFile);
19:
20:            // Read data
21:            Console.WriteLine("Reading file....");
22:            while( brFile.PeekChar() != -1 )
23:            {
24:                Console.Write("<{0}> ", brFile.ReadInt32());
25:            }
26:
27:            Console.WriteLine("....Done Reading.");
28:
29:            brFile.Close();
30:            myFile.Close();
31:        }
32:    }
33: }
```

该程序清单的输出如下:

Reading file.....

```

<0> <1> <2> <3> <4> <5> <6> <7> <8> <9> <10> <11> <12> <13> <14> <15> <16> <17>
<18> <19> <20> <21> <22> <23> <24> <25> <26> <27> <28> <29> <30> <31> <32> <33>
<34> <35> <36> <37> <38> <39> <40> <41> <42> <43> <44> <45> <46> <47> <48> <49>
<50> <51> <52> <53> <54> <55> <56> <57> <58> <59> <60> <61> <62> <63> <64> <65>
<66> <67> <68> <69> <70> <71> <72> <73> <74> <75> <76> <77> <78> <79> <80> <81>
```

```
<82> <83> <84> <85> <86> <87> <88> <89> <90> <91> <92> <93> <94> <95> <96> <97>
<98> <99> ....Done Reading.
```

分析：使用该程序可以读取前一个程序清单写入的数据。第17行创建了一个FileStream对象，这里使用的文件模式是Open，然后第18行将该对象与一个BinaryReader关联起来，以读取二进制信息。

第22行与以前稍微有些不同，它使用了BinaryReader类的PeekChar方法。该方法查看流中的下一个字符。如果下一个字符是文件末尾，则返回-1；否则，返回下一个字符。在这样做的时候，在流中的位置保持不变。该方法让您能够查看下一个字符。

只要下一个字符不是文件末尾，便执行第24行，从BinaryStream对象brFile中读取一个整数。用于读取整数的方法ReadInt32使用的是.NET框架中的一种类型的名称，而不是C#的名称。记住，C#可以使用.NET框架中的所有类——它们不属于C#语言，除了C#语言之外，其他语言也可以使用这些类。

对于其他每种基数据类型，BinaryReader类也包含相应的、与ReadInt32类似的方法，这些方法的用法与ReadInt32相同。

15.6.4 使用其他文件类型

前几节介绍了如何读写基本文本和二进制数据，还有用于读取其他类型数据（包括XML）的类。另外，今天介绍的类和其他类型还具备大量其他的功能。

包含类和其他类型的名称空间支持更高级的文件访问。虽然这些类通过较少的代码便可以提供更强大的功能，但应该知道使用它们的代价——这些类可能没有遵循.NET标准，因此可能是不可移植的。

15.7 总 结

今天介绍了.NET框架中的一些基类。编写本书时，今天介绍的类已被提供，以便标准化。这意味着，其可移植性将与C#程序一样高。

今天首先介绍了定时器——用于在特定的时间过去后引发事件；然后介绍了如何获取关于当前目录和文件以及系统本身的信息。您经常需要用到数学函数，而今天介绍了Math类中的大量数学函数。

最后，介绍了文件的存取，讨论了如何读写文本文件和二进制文件。

15.8 问与答

问：我尝试使用帮助文档中介绍的类，但编译时被告知缺少一个组合体（assembly），我该怎么办？

答：如果代码没有任何错误，并仍然出现缺少文件或组合体的错误，则可能需要在编译命令中包含对.NET框架中某个组合体的引用，方法是，使用开关/r:和要包含的名称空间所在的磁盘文件名。帮助文档将告诉您各个类需要哪个文件。例如，System.TextReader类型位于组合体mscorlib.dll中。假设要编译的文件为xxx.cs，而该文件需要使用上述组合体，则命令如下：

```
csc /r:mscorlib.dll xxx.cs
```

问: 今天介绍过, 可以使用 `Environment` 类的 `GetCommandLineArgs` 方法来获取命令行参数; 而本书前面介绍过, 可以在 `Main` 方法中使用字符串参数来获取命令行值。请问哪种方法更好?

答: 这两种方法都管用。差别在于, 使用 `GetCommandLineArgs` 还可以取得被执行的程序的名称。`Main` 参数的第一个值是第一个参数, 而不是被执行的程序的名称。

问: XML 和 ADO 属于标准类吗?

答: 不属于。有许多用于 XML 的类正在被标准化, 但 ADO 是微软公司的技术, 它不是标准的一部分。

15.9 作业

下面的小测验帮助您巩固所学的知识, 练习则让您实际应用所学的知识。在阅读下一课时之前, 应尽可能理解这些小测验和练习的答案, 答案见附录 A。

15.9.1 小测验

1. 一秒钟包含多少次滴答 (tick) ?
2. 定时器使用下面哪种东西?
 - a. 代表;
 - b. 事件;
 - c. 异常。
3. 哪个标准化组织负责 C# 和基类库的标准化工作?
4. 使用 `Environment.GetCommandLineArgs` 和 `Main(string args[])` 之间的差别何在?
5. 何时创建 `Math` 类的一个实例 (`Math` 对象) ?
6. 哪个类或方法可用于确定文件是否存在?
7. 文件和流之间的区别何在?
8. 哪个 `FileMode` 值可用于新建一个文件?
9. 哪些类用于 XML?

15.9.2 练习

1. 创建一个程序, 它使用二进制文件方法来写文件。创建一个用于存储人的姓名、年龄、会员资格的结构。将这些信息写入到文件中 (提示: 年龄可以是整数, 会员资格可以是布尔型)。
2. 修改程序清单 15.4, 在其中使用 `Math` 类中的 `Cos` 方法。
3. 创建一个程序, 从控制台读取文本, 并将其写入到文件中。用户输入一个空行, 表示结束输入。

4. 下面的程序有问题吗?

```
1: using System;
2: using System.IO;
3:
4: class MyStream
5: {
6:     public static void Main(String[] args)
7:     {
8:         FileStream myFile = new FileStream(args[0], FileMode.Open);
```

```
9:         BinaryReader brFile = new BinaryReader(myFile);
10:         while( brFile.PeekChar() != -1 )
11:         {
12:             Console.Write("<{0}> ", brFile.ReadInt32());
13:         }
14:     }
15: }
```

第 16 天课程

创建 Windows 窗体

微软公司的基类库中包含大量用于创建和处理基于窗体 (form) 的 Windows 应用程序 (包括创建窗体和控件) 的类。今天将介绍以下内容:

- 如何创建 Windows 窗体;
- 定制窗体的外观;
- 将控件加入到 Windows 窗体中;
- 使用文本框、标签等;
- 通过设置控件的属性定制其外观;
- 将事件和控件关联起来。

16.1 使用 Windows 和窗体

当今的大多数操纵系统都使用事件驱动编程技术和窗体来与用户交互。如果您从事过 Microsoft Windows 环境下的开发工作, 则很可能使用过 Win32 库中的一组函数, 这些函数帮助您创建窗口。昨天介绍了基类库 (BCL), BCL 中包含一组用于开发类似的窗口和窗体的类。基类库中的类的优点在于, 它可以被 .NET 框架中的任何编程语言使用。另外, 它们使得开发基于窗体的应用程序非常容易。

16.2 创建 Windows 窗体

要创建 Windows 窗体应用程序, 可以从 Forms 类派生出新的类。Forms 类位于名称空间 System.Windows.Forms 中。程序清单 16.1 列出了创建最小的 Windows 窗体应用程序所需的代码。

程序清单 16.1 firstfm.cs: 一个简单的 Windows 窗体应用程序

```
1: // firstfm.cs - A super simplistic windows form application
2: //-----
3:
4: using System.Windows.Forms;
5:
6: public class frmHelloApp : Form
```

```

7: {
8:     public static void Main( string[] args )
9:     {
10:         frmHelloApp.frmHello = new frmHelloApp();
11:         Application.Run(frmHello);
12:     }
13: }

```

如果考虑到其功能，则该程序清单是非常短的。要知道其功能，需要编译它。下一节将介绍如何编译它。

16.2.1 编译选项

编译程序清单 16.1 的方法与以前不同，需要在编译命令中包含对基类的引用，这在昨天做了简要的介绍。

Forms 类位于一个名为 System.Windows.Forms.dll 的组合体 (assembly) 中，编译程序时，可能需要包含对该组合体的引用。仅仅在程序清单的开始位置使用 using 语句，并不能真正地将任何文件包含到程序中，而只是提供了到文件中名称空间的某一点的引用。正如您学习并已经明白的，这只是让您能够使用简短的名称，而不是全限定名称。大多数常用的 Windows 窗体控件和窗体功能都位于组合体 System.Windows.Forms.dll 中。

为确保该组合体被使用，需要在编译程序时使用引用命令行参数——/reference:filename，其中 filename 是组合体的名称。使用组合体 Forms 来编译程序清单 16.1 中的 firstfrm.cs 的命令如下：

```
csc /reference:System.Windows.Forms.dll firstfrm.cs
```

也可以使用 /reference: 的缩写 /r。执行上述编译命令后，该程序便可以执行。

如果从命令提示符执行应用程序 firstfrm，将显示如图 16.1 所示的窗口。

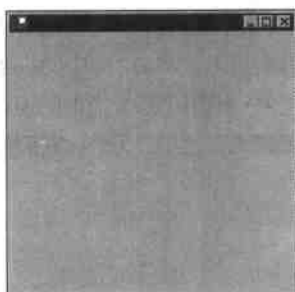


图 16.1 应用程序 firstfrm 的窗体

这与您期望的完全相同，但如果直接在诸如 Microsoft Windows 等操作系统中执行该程序，结果将稍有不同。出现的将是命令行窗口和 Windows 窗体（见图 16.2），其中的命令行窗口并不是您创建的。

要防止出现命令行窗口，需要告诉编译器您希望创建一个在 Microsoft Windows 操作系统中运行的程序，方法是使用 /target: 标记和 winexe 选项。您可以使用缩写 /t。使用下面的命令重新编译程序 firstfrm.cs，得到的结果将是您希望的：

```
csc /r:System.Windows.Forms.dll /t:winexe firstfrm.cs
```

当执行该程序时，它不会首先创建一个命令窗口。

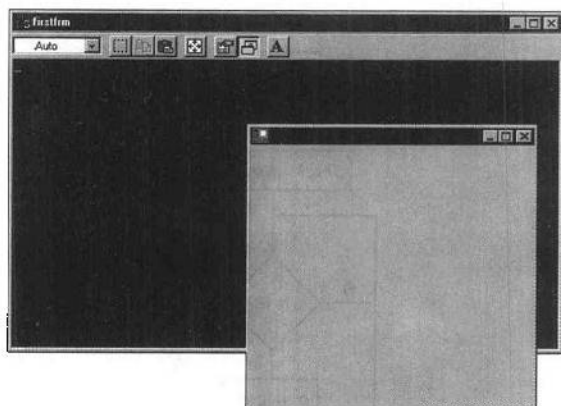


图 16.2 应用程序 `frmstfrm` 实际显示的内容

注意：需要知道的是，当您编译程序时，将自动包含某些组合体。例如，默认情况下，诸如 Microsoft Visual C# 等开发工具包含一些组合体。如果一个组合体也没有包含进来，则编译时将出错，指出可能缺少一个组合体。

16.2.2 分析第一个 Windows 窗体应用程序

编译并执行 Windows 窗体应用程序后，应该开始了解其中的代码。我们回过头来看看程序清单 16.1 中的代码。

第 4 行包含了名称空间 `System.Windows.Forms`，这样便可以通过简短的名称来引用 `Form` 和 `Application` 类。第 6 行将该应用程序放在一个名为 `frmHelloApp` 的类中，这个类是从 `Form` 类派生而来的，后者提供了 Windows 窗体的所有基本功能。

注意：今天的课程后面将介绍，名称空间 `System.Windows.Forms` 还包含使 Windows 窗体的用途更大的控件、事件、属性和其他代码。

仅仅通过一行代码（第 6 行），您实际上创建了一个窗体应用程序。第 10 行实例化这个类的一个对象。第 11 行调用 `Application` 类的 `Run` 方法，这将在稍后做更详细的介绍。现在您只需知道，它是应用程序显示窗体，并不断运行，直到用户关闭窗体。也可以调用 `Form` 类的 `Show` 方法，做法是将第 11 行修改成如下所示：

```
frmHello.Show();
```

虽然这看起来更为简单直观，但应用程序将由于故障而结束。程序显示窗体，然后进入下一行——程序的末尾。由于到达了程序末尾，因此处理将结束，而窗体将关闭。这并不是您期望的结果。`Application` 类可用于避免这个问题。

注意：后面将介绍显示窗体，然后等待的窗体方法。

16.2.3 `Application.Run` 方法

Windows 应用程序是事件驱动的程序，通常显示一个包含控件的窗体。然后程序进入一个循环，直到用户在窗体上或 Windows 环境中执行某种操作。当发生某种事情时，消息将被创建，这些消息将导致事件发生。如果消息有对应的事件处理程序，则该事件处理程序将被执行；如果没有，循环

将继续。图 16.3 说明了这种循环。

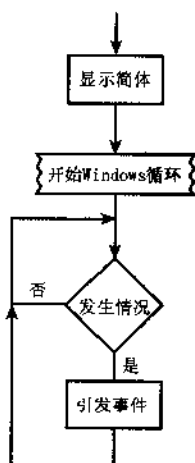


图 16.3 标准 Windows 程序的流程

正如您看到的，循环似乎永远不会终止。实际上，有一种事件可以终止程序。从 Form 派生而来的基本窗体包含关闭按钮，其命令菜单中包含 Close 选项。这些控件都可以引发关闭窗体并结束循环的事件。

现在您应该猜到了 Application 类（更准确地说是 Application 类的 Run 方法）的功能。Run 方法负责创建循环，并使程序不断运行下去，直到结束程序循环的事件发生。对于程序清单 16.1 而言，单击窗体上的关闭按钮或选择命令菜单中的 Close 选项将引发一个结束循环并关闭窗体的事件。

Application.Run 方法还为您显示窗体。程序清单 16.1 的第 11 行接受一个窗体对象 frmHello，这是一个从 Form 类派生而来对象。Application.Run 方法将显示该窗体，然后开始循环。

注意：Application 类的 Run 方法创建的循环实际上是处理被创建的消息。这些消息可以是操作系统、应用程序或其他正在运行的应用程序创建的，循环将处理这些消息。例如，当您单击按钮时，将有大量的消息被创建，这包括鼠标被按下、松开以及单击按钮等消息。如果消息与某个事件处理程序匹配，则该事件处理程序将被执行；如果没有定义相应的事件处理程序，则该消息将被忽略。

16.3 定制窗体的外观

前一个程序清单包含一个基本窗体，与 Form 类相关的属性、方法和事件非常多，本书无法一一介绍，但有必要简要地介绍其中的一些。关于 Form 功能的完整列表可参阅在线文档。

16.3.1 窗体的标题栏

程序清单 16.1 包含一个基本的空窗体。接下来的几个程序清单将继续使用空窗体，但在今天的每个程序清单中，都将介绍如何控制窗体的某些方面。

程序清单 16.1 中的窗体包含一些可用的控件，包括控制菜单、最大化按钮、最小化按钮和关闭按钮。可以通过设置属性来控制窗体的这些控件是否可用：

- **ControlBox**：决定是否显示控制按钮；
- **HelpButton**：决定窗体的标题栏是否出现帮助按钮，仅当 MaximizeBox 和 MinimizeBox 为 false

时才会显示;

- **MaximizeBox**: 决定是否包含最大化按钮;
- **MinimizeBox**: 决定是否包含最小化按钮;
- **Text**: 窗体的标题 (caption)。

其中的一些属性值会影响其他属性。例如, 仅当属性 **MaximizeBox** 和 **MinimizeBox** 都为 **false** 时, 才会显示帮助按钮。程序清单 16.2 演示了如何设置上述属性的值, 其输出如图 16.4 所示。请输入该程序清单, 然后编译并运行它。别忘了在编译时包含 `/t:winexe` 标记。

程序清单 16.2 form2.cs: 设置窗体的标题栏

```
1: // form2.cs - Caption Bar properties
2: //-----
3:
4: using System.Windows.Forms;
5:
6: public class frmHelloApp : Form
7: {
8:     public static void Main( string[] args )
9:     {
10:         frmHelloApp frmHello = new frmHelloApp();
11:
12:         // Caption bar properties
13:         frmHello.MinimizeBox = true;
14:         frmHello.MaximizeBox = false;
15:         frmHello.HelpButton = true;
16:         frmHello.ControlBox = true;
17:         frmHello.Text = @"My Form's Caption";
18:
19:         Application.Run(frmHello);
20:     }
21: }
```

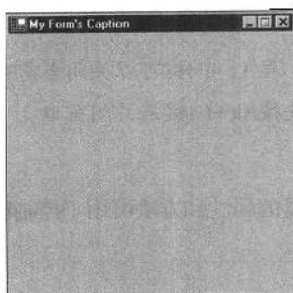


图 16.4 程序清单 16.2 的输出

分析: 该程序清单很容易理解。第6行创建了一个名为 `frmHelloApp` 的新类, 它是从 `Form` 类派生而来的。第10行实例化一个 `frmHelloApp` 对象, 第13~17行设置了该窗体的许多标题栏属性。第19行调用 `Application` 类的 `Run` 方法来显示该窗体。从图16.4可知, 其中包含最大化和最小化按钮, 但最大化按钮不可用, 这是因为第14行将 `MaximizeBox` 属性设置为了 `false`。如果将属性 `MaximizeBox` 和 `MinimizeBox` 都设置为 `false`, 则这两个按钮将不会出现。

第 15 行将 `HelpButton` 属性设置为 `false`。仅当属性 `MaximizeBox` 和 `MinimizeBox` 都被设置为 `false` 时,才可能显示帮助按钮。这意味着第 15 行将被忽略。请修改第 13 行,将属性 `MinimizeBox` 也设置为 `false`,然后重新编译和运行该程序,将得到如图 16.5 所示的结果。

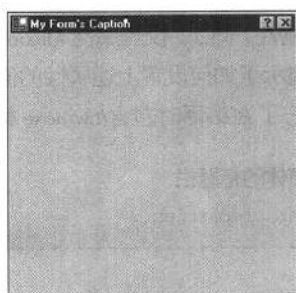


图 16.5 包含帮助按钮

从上述结果可知,此时第 14 行的设置没有被忽略。

还需要注意另一种组合情况。如果将 `ControlBox` 设置为 `false`,则关闭按钮和控制按钮都将被隐藏。另外,如果属性 `ControlBox`、`MinimizeBox` 和 `MaximizeBox` 都被设置为 `false`,同时 `Text` 属性没有被设置,则整个标题栏都将隐藏。请删除程序清单中的第 17 行,并将第 13~16 行的各个属性都设置为 `false`,然后重新编译并运行该程序,将得到如图 16.6 所示的结果。

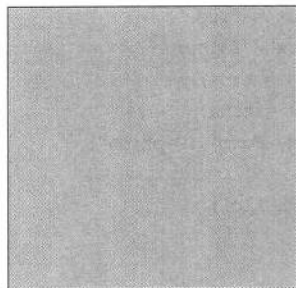


图 16.6 没有标题栏

注意: 在 Microsoft Windows 中,按 `Alt+F4` 键可以关闭当前窗口。如果隐藏控制按钮,则关闭按钮也不会出现。在这种情况下,需要按 `Alt+F4` 键来关闭窗口。

16.3.2 窗体的大小

接下来控制窗体的大小。有大量的属性和方法可用于控制窗体的形状和大小。表 16.1 列出了其中的一些。

表 16.1 Form 类的大小控制属性

属 性	描 述
<code>AutoScale</code>	窗体根据字体和/或其中的控件自动调整其大小
<code>AutoScaleBaseSize</code>	用于自动调整窗体大小的基数
<code>AutoScroll</code>	窗体将能够自动滚动

续表

属 性	描 述
AutoScrollMargin	自动滚动的最大值
AutoScrollMinSize	自动滚动的最小值
AutoScrollPosition	自动滚动的位置
ClientSize	窗体客户区域的大小
DefaultSize	一种保护属性, 设置窗体的默认大小
DesktopBounds	窗体的大小和位置
DesktopLocation	窗体的位置
Height	窗体的高度
MaximizeSize	被最大化时, 窗体的大小
MinimizeSize	被最小化时, 窗体的大小
Size	窗体的大小, 设置或获得包含 x 和 y 的 Size 对象
SizeGripStyle	设置窗体大小调整指示器的显示方式, 取值为 SizeGripStyle 枚举中的值, 包括 Auto (必要时自动显示)、Hide (隐藏) 和 Show (始终显示)
StartPosition	窗体的起始位置, 取值为 FormStartPosition 枚举中的值。可能的取值包括 CenterParent (位于父窗体的中央)、CenterScreen (位于屏幕的中央)、Manual (位置和大小取决于起始位置)、WindowsDefaultBounds (位于默认位置) 和 WindowsDefaultLocation (位于默认位置, 大小取决于指定的大小值)
Width	窗体的宽度

上表只列出了可用于控制窗体大小的属性和方法的一部分。程序清单 16.3 在一个简单的应用程序中使用了其中的一些属性, 该程序的结果如图 16.7 所示。

程序清单 16.3 form3.cs: 控制窗体的大小

```

1: // form3.cs - Form Size
2: //-----
3:
4: using System.Windows.Forms;
5: using System.Drawing;
6:
7: public class frmHelloApp : Form
8: {
9:     public static void Main( string[] args )
10:    {
11:        frmHelloApp myForm = new frmHelloApp();
12:        myForm.Text = "Form Sizing";
13:
14:        myForm.Width = 400;

```

```

15:      myForm.Height = 100;
16:
17:      Point FormLoc = new Point(200,350);
18:      myForm.StartPosition = FormStartPosition.Manual;
19:      myForm.DesktopLocation = FormLoc;
20:
21:
22:      Application.Run(myForm);
23:  }
24: }

```

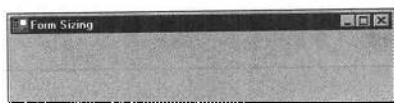


图 16.7 控制窗体的大小和位置

分析：设置窗体的大小很简单。第14和15行设置了窗体的大小。正如您看到的，可以设置属性 Width和Height，也可以使用Size对象同时设置窗体的高度和宽度。

设置窗体的位置稍微复杂些。第17行创建了一个Point对象，它包含您希望窗体在屏幕上的位置。第19行将该对象赋给DesktopLocation属性。为在使用Point类时不必使用全限定名，需要将名称空间System.Drawing包含进来，如第5行所示。

第18行设置了另一个属性，如果删除第18行，将得不到您想要的结果。必须将StartPosition属性设置为FormStartPosition枚举中的一个值，来设置窗体的起始位置，表16.1列出了该枚举可能的取值。您应该注意到了FormStartPosition枚举的其他取值，如果要将窗体放置在屏幕中央，则可以将第17~19行的代码替换为下面的一行代码：

```
myForm.StartPosition = FormStartPosition.CenterScreen;
```

上述代码行将窗体放在屏幕的中央，而不管屏幕的分辨率如何。

16.3.3 窗体的颜色和背景

要设置窗体的背景色，需要将BackColor属性设置为相应的颜色值。可以从结构Color中取得颜色值，该结构位于名称空间System.Drawing中。表16.2列出了一些常用的颜色。

要设置背景颜色，只需将表16.2的值赋给BackColor属性即可：

```
myForm.BackColor = Color.HotPink;
```

除了可以设置窗体的颜色外，还可以设置窗体的背景图案，方法是将BackgroundImage属性设置为一幅图像。程序清单16.4将背景设置为一幅图像，图16.8是其输出。这里，将图像文件作为一个参数传递给程序。

警告：请注意，出于简洁的目的，该程序清单中没有包含异常处理。如果用户传递一个不存在的文件名，则程序将引发异常。

表 16.2 常用的颜色值

AliceBlue	AntiqueWhite	Aqua	Aquamarine	Azure	Beige
Bisque	Black	BlanchedAlmond	Blue	BlueViolet	Brown

续表

BurlyWood	CadetBlue	Chartreuse	Chocolate	Coral	CorianderBlue
Cornsilk	Crimson	Cyan	DarkBlue	DarkCyan	DarkGoldenrod
DarkGray	DarkGreen	DarkKhaki	DarkMagenta	DarkOliveGreen	DarkOrange
DarkOrchid	DarkRed	DarkSalmon	DarkSeaGreen	DarkSlateBlue	DarkSlateGray
DarkTurquoise	DarkViolet	DeepPink	DeepSkyBlue	DimGray	DodgerBlue
Firebrick	FloralWhite	ForestGreen	Fuchsia	Gainsboro	GhostWhite
Gold	Goldenrod	Gray	Green	GreenYellow	Honeydew
HotPink	IndianRed	Indigo	Ivory	Khaki	Lavender
LavenderBlush	LawnGreen	LemonChiffon	LightBlue	LightCoral	LightCyan
LightGoldenrodYellow	LightGray	LightGreen	LightPink	LightSalmon	LightSeaGreen
LightSkyBlue	LightSlateGray	LightSteelBlue	LightYellow	Lime	LimeGreen
Linen	Magenta	Maroon	MediumAquaMarine	MediumBlue	MediumOrchid
MediumPurple	MediumSeaGreen	MediumSlateBlue	MediumSpringGreen	MediumTurquoise	MediumVioletRed
MidnightBlue	MintCream	MistyRose	Moccasin	NavajoWhite	Navy
OldLace	Olive	OliveDmb	Orange	OrangeRed	Orchid
PaleGoldenrod	PaleGreen	PaleTurquoise	PaleVioletRed	PapayaWhip	PeachPuff
Peru	Pink	Plum	PowderBlue	Purple	Red
RosyBrown	RoyalBlue	SaddleBrown	Salmon	SandyBrown	SeaGreen
SeaShell	Sienna	Silver	SkyBlue	SlateBlue	SlateGray
Snow	SpringGreen	SteelBlue	Tan	Teal	Thistle
Tomato	Transparent	Turquoise	Violet	Wheat	White
WhiteSmokeYellow	YellowGreen				

程序清单 16.4 form4.cs: 使用背景图像

```

1: // form4.cs - Form Backgrounds
2: //-----
3:
4: using System.Windows.Forms;
5: using System.Drawing;
6:
7: public class frmApp : Form
8: {
9:     public static void Main( string[] args )

```

```
10:    {  
11:        frmApp myForm = new frmApp();  
12:        myForm.BackColor = Color.HotPink;  
13:        myForm.Text = "Form4 - Backgrounds";  
14:  
15:        if (args.Length >= 1)  
16:        {  
17:            myForm.BackgroundImage = Image.FromFile(args[0]);  
18:  
19:            Size tmpSize = new Size();  
20:            tmpSize.Width = myForm.BackgroundImage.Width;  
21:            tmpSize.Height = myForm.BackgroundImage.Height;  
22:            myForm.ClientSize = tmpSize;  
23:  
24:            myForm.Text = "Form4 - " + args[0];  
25:        }  
26:  
27:        Application.Run(myForm);  
28:    }  
29: }
```

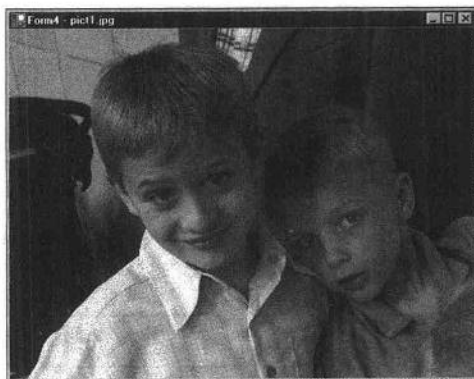


图 16.8 使用背景图像

分析：该程序将一个图像作为窗体背景，图像是通过命令行提供的。如果没有在命令行提供图像，窗体的背景色将被设置为粉红色（hot pink）。作者运行该程序时，提供的是外甥的照片，使用的命令如下：

```
form4 pict1.jpg
```

pict1.jpg 位于 form4.exe 所在的目录中。如果位于其他的目录中，则需要输入完整的路径。您可以提供其他的图像，只要路径有效即可。如果输入的文件名无效，将出现异常。

来看一下程序清单。创建显示图像的应用程序非常容易，.NET 框架类将负责替您完成所有困难的工作。第 12 行将背景颜色设置为粉红色，这是通过将 BackColor 属性设置为 Color 结构中的相应颜色值实现的。

第 15 行检查用户是否提供了命令行参数。如果没有，则跳过第 17~24 行，并将窗体显示为粉红色；如果提供了，则程序假设（您的程序不应该这样做）传递的参数是一个有效的图形文件，并

将该文件赋给窗体的 `BackgroundImage`。此时需要使用 `Image` 类将该文件转换为实际的背景图像，更准确地说，`Image` 类包含一个名为 `FromFile` 的静态方法，它接受文件名参数，并返回一个 `Image` 对象。这就是该程序清单的所有内容。

注意：如果要将某个特定的图像用作背景，可以删除程序清单中的 `if` 语句，并将第 17 行的 `arg[0]` 替换为要用作背景的文件名。

`BackgroundImage` 属性存储一个 `Image` 对象，因此可以使用 `Image` 类的方法来设置该属性的值。`Image` 类包含属性 `Width` 和 `Height`，它们的值等于该类中包含的图像的宽度和长度。第 20 和 21 行将这些值赋给一个临时的 `Size` 变量，然后在第 22 行将窗体客户区域的大小设置为该 `Size` 变量的值。这样，窗体客户区域的大小与图像相同，因此显示的窗体将总是显示整个图像。如果不这样做，窗体将可能只显示图像的一部分或显示多个平铺的图像。

16.3.4 边框

控制边框不仅影响窗体的外观，还将决定窗体的大小是否可以改变。要修改边框，可以将 `Form` 类的 `BorderStyle` 属性设置为枚举 `FormBorderStyle` 的值之一。表 16.3 列出了 `BorderStyle` 属性的可能取值。程序清单 16.5 包含一个边框被修改的窗体，该程序清单的运行结果如图 16.9 所示。

表 16.3 枚举 `FormBorderStyle` 的值

值	描 述
<code>Fixed3D</code>	固定的三维边框
<code>FixedDialog</code>	固定的厚边框
<code>FixedSingle</code>	固定的单线边框
<code>FixedToolWindow</code>	大小不可改变的工具窗口边框
<code>None</code>	没有边框
<code>Sizeable</code>	大小可变
<code>SizeableToolWindow</code>	大小可变的工具窗口边框

程序清单 16.5 border.cs: 改变窗体的边框

```

1: // border.cs - Form Borders
2: //-----
3:
4: using System.Windows.Forms;
5: using System.Drawing;
6:
7: public class frmApp : Form
8: {
9:     public static void Main( string[] args )
10:    {
11:        frmApp myForm = new frmApp();
12:        myForm.BackColor = Color.SteelBlue;
13:        myForm.Text = "Borders";

```

```
14:
15:     myForm.FormBorderStyle = FormBorderStyle.Fixed3D;
16:
17:     Application.Run(myForm);
18: }
19: }
```

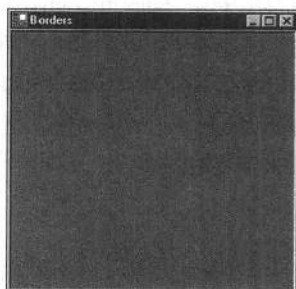


图 16.9 改变窗体的边框

分析：正如您看到的，边框的大小是固定的，用户无法在运行阶段改变窗体的大小。

要将窗体设置为大小可变的，应设置另一个属性 `SizeGripStyle`，该属性决定窗体是否包含一个大小调整的指示器。图 16.10 将大小调整的指示器调起来了。可以将窗体设置为该指示器自动被显示、总是隐藏或总是显示，这是通过使用 `SizeGripStyle` 枚举的三个值（`Auto`、`Hide`、`Show`）来实现的。图 16.10 中的指示器是使用下面的代码行来显示的：

```
MyForm.SizeGripStyle = SizeGripStyle.Show;
```

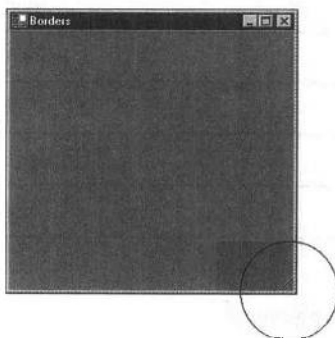


图 16.10 大小调整指示器

警告：不要被使用相互冲突的属性迷惑。例如，如果您使用大小固定的边框，同时将大小调整指示器设置为总是显示，由于前者意味着窗体的大小不可调整，因此大小调整指示器将不会显示，而不管您如何设置它。

16.4 将控件加入到窗体中

至此，介绍了如何设置窗体的外观，但不包含控件的窗体将毫无用处。控件使 Windows 应用程序变得有用。

控件可以是按钮、列表框、文本框、图像，乃至纯文本。最简单的加入控件的方式是使用图形开发工具，如 Microsoft Visual C#。图形工具让您能够将控件拖放到窗体上，并加入显示控件所需要的所有基本代码。

虽然不一定非得使用图形开发工具，但了解工具能为您做哪些工作也很重要。表 16.4 列出了 .NET 框架提供的一些标准控件，当然您可以创建并使用其他的控件。

表 16.4 基类库中的一些标准控件

Button	CheckBox	CheckedListBox	ComboBox
ContainerControl	DataGrid	DateTimePicker	DomainUpDown
Form	GroupBox	HScrollBar	ImageList
Label	LinkLabel	ListBox	ListView
MonthCalendar	NumericUpDown	Panel	PictureBox
PrintReviewControl	ProgressBar	PropertyGrid	RadioButton
RichTextBox	ScrollableControl	Splitter	StatusBar
StatusBarPanel	TabControl	TabPage	TabStrip
TextBox	Timer	ToolBar	ToolBarButton
ToolTip	TrackBar	TreeView	VScrollBar
UserControl			

表 16.4 列出的控件是在名称空间 System.Windows.Forms 中定义的，接下来的几节将介绍其中的一些控件，但您需要知道的是，这里介绍的只是各个控件的非常少的一部分内容。与表 16.4 列出的控件相关的属性、方法和事件成百上千，要详细介绍各个控件，需要一篇篇幅比本书还长的书。这里仅介绍如何使用其中的一些非常重要的控件，使用其他控件的步骤与此类似。另外，这里只介绍为数不多的几个属性，有关全部的属性，可以在 C# 编译器或开发工具提供的帮助文档中找到。

16.4.1 使用标签来显示文本

可以使用 Label（标签）控件在屏幕上显示简单的文本。Label 以及其他内置控件都位于名称空间 System.Windows.Forms 中。

要将控件加入到窗体中，必须首先创建该控件，然后通过控件属性和方法定制它。按需要修改好后，便可以将控件加入到窗体中。

新术语：标签是一种用于将信息显示给用户，但不允许用户直接修改的控件。创建控件的方式与其他对象一样：

```
Label myLabel = new Label();
```

标签被创建后，便可以将其加入到窗体中，不过此时标签的内容为空。程序清单 16.6 演示了如何使用标签的一些属性，包括使用 Text 属性设置标签的文本值。图 16.11 是该程序清单的运行结果。要将控件加入到窗体中，可以使用窗体的 Controls 属性的 Add 方法。例如，要将 myLabel 控件

加入到前面使用的 myForm 窗体中，可以使用下面的代码：

```
myForm.Controls.Add(myLabel);
```

要加入其他的控件，只要将 myLabel 替换为相应的控件名即可。

程序清单 16.6 Control1.cs: 使用 Label 控件

```
1: // controll1.cs - Working with controls
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmApp : Form
9: {
10:     public static void Main( string[] args )
11:     {
12:         frmApp myForm = new frmApp();
13:
14:         myForm.Text = Environment.CommandLine;
15:         myForm.StartPosition = FormStartPosition.CenterScreen;
16:
17:         // Create the controls...
18:         Label myDateLabel = new Label();
19:         Label myLabel = new Label();
20:
21:         myLabel.Text = "This program was executed at:";
22:         myLabel.AutoSize = true;
23:         myLabel.Left = 50;
24:         myLabel.Top = 20;
25:
26:         DateTime currDate = DateTime.Now;;
27:         myDateLabel.Text = currDate.ToString();
28:
29:         myDateLabel.AutoSize = true;
30:         myDateLabel.Left = 50 + myLabel.PreferredWidth + 10;
31:         myDateLabel.Top = 20;
32:
33:         myForm.Width = myLabel.PreferredWidth + myDateLabel.PreferredWidth + 110;
34:         myForm.Height = myLabel.PreferredHeight+ 100;
35:
36:         // Add the control to the form...
37:         myForm.Controls.Add(myDateLabel);
38:         myForm.Controls.Add(myLabel);
39:
40:         Application.Run(myForm);
41:     }
42: }
```

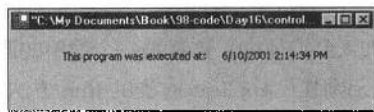


图 16.11 使用 Label 控件

分析：该程序创建两个标签控件，并在窗体中显示它们。该程序清单将这些标签放在窗体的大概中央位置，而不是随便放置。

来看程序清单。该程序首先创建了一个新的窗体（第 12 行），该窗体的标题被设置为 Environment 类中的 CommandLine 成员的值，这将是程序名和完整的路径。第 15 行将窗体的 StartPosition 属性设置为屏幕中央。到目前位置，还没有设置窗体的大小，稍后将这样做。

第 18 和 19 行创建了两个标签，第一个标签 myDateLabel 将用于放置当前的日期和时间；第二个标签用于放置描述性文本。前面介绍过，标签是一种用来为用户显示信息的控件，但不允许用户直接修改其值，因此在这里使用标签是合适的。

第 21 ~ 24 行设置了标签 myLabel 的属性。第 21 行将要显示的文本赋给了该标签的 Text 属性；第 22 行将属性 AutoSize 设置为 true。您可以控制标签的大小，也可以让它自己决定最佳的大小。将 AutoSize 属性设置 true，让标签能够调整自己的大小。第 23 和 24 行设置了属性 Left 和 Top 的值，这些值用于决定控件在窗体中的位置。这里，myLabel 控件放置在离窗体客户区域最左边 50 个像素，离最上边 20 个像素。

接下来的两行（第 26 和 27 行）迂回地将当前日期和时间赋给另一个标签 myDateLabel 的 Text 属性，正如您看到的，创建了一个 DateTime 对象，并将其值设置为 Now，然后将其转换为一个字符串，并赋给 myDateLabel 的 Text 属性。

第 29 行将 myDateLabel 的 AutoSize 的属性也设置为 true，这样该标签的大小将自动调整为最合适。第 30 和 31 行设置了 myDateLabel 的位置。其中 Top 的值容易理解——该标签的垂直位置与另一个标签相同，但 Left 的值稍微复杂些。MyDateLabel 将被放置在另一个标签的右边。要将其放在另一个标签的右边，需要将其 Left 值设置为另一个标签的 Width 和 Left 值之和，即 50 加上另一个标签的宽度。由于标签的大小是自动调整的，因此其宽度为最合适的宽度。标签最合适的宽度可以从其 PreferredWidth 属性取得。因此，要将 myDateLabel 放在 myLabel 的右边，应将 myLabel 的最合适宽度及其偏移量（Left）相加。为增加一些空余，再加了 10 个像素。图 16.12 可帮助您理解第 30 行代码的含义。

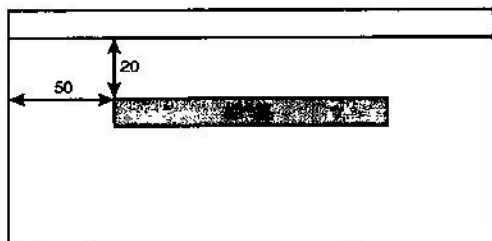


图 16.12 标签的位置

第 33 和 34 行设置了窗体的宽度和高度。正如您看到的，宽度被设置为使得标签大概位于窗体中央的值。这是通过补偿偏移量并使用两个标签的宽度来完成的；高度被设置为确保文本周围有一定的空间。

第 37 和 38 行将这些标签加入到窗体中，这只是一个简单的调用，为每个控件调用了 Controls 属性的 Add 方法。然后，第 40 行调用了 Application 类的 Run 方法，以显示窗体。最终的结果是，窗体中显示了文本。

其他控件的使用步骤在很大程度上与此相同，这包括创建控件、设置控件的属性以及将其加入到窗体中。

16.4.2 建议使用控件的方法

前一节介绍的使用控件的步骤是合适的。Microsoft Visual Studio.NET 是最常用的 Windows 应用程序开发工具，因而对于 C# 应用程序来说，最常用的工具是 Microsoft Visual C#。在使用控件方面，这种开发工具提供了一种独特的结构。虽然这不是必须的，但这种结构确实对代码进行了组织，让图形开发工具能更好地理解代码。由于学会这种方法很容易，因此值得考虑。程序清单 16.7 以这种稍微不同的结构对程序清单 16.6 进行了重新组织，这种结构与 Microsoft Visual C# 生成的类似。

程序清单 16.7 为集成开发环境结构化代码

```
1: // cntrl1b.cs - Working with controls
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmApp : Form
9: {
10:     public frmApp()
11:     {
12:         InitializeComponent();
13:     }
14:
15:     private void InitializeComponent()
16:     {
17:         this.Text = Environment.CommandLine;
18:         this.StartPosition = FormStartPosition.CenterScreen;
19:
20:         // Create the controls...
21:         Label myDateLabel = new Label();
22:         Label myLabel = new Label();
23:
24:         myLabel.Text = "This program was executed at:";
25:         myLabel.AutoSize = true;
26:         myLabel.Left = 50;
27:         myLabel.Top = 20;
28:
29:         DateTime currDate = new DateTime();
30:         currDate = DateTime.Now;
```

```

31:     myDateLabel.Text = currDate.ToString();
32:
33:     myDateLabel.AutoSize = true;
34:     myDateLabel.Left = 50 + myLabel.PreferredWidth + 10;
35:     myDateLabel.Top = 20;
36:
37:     this.Width = myLabel.PreferredWidth + myDateLabel.PreferredWidth + 110;
38:     this.Height = myLabel.PreferredHeight + 100;
39:
40:     // Add the control to the form...
41:     this.Controls.Add(myDateLabel);
42:     this.Controls.Add(myLabel);
43: }
44:
45: public static void Main( string[] args )
46: {
47:     Application.Run( new frmApp() );
48: }
49: }

```

分析：该程序清单的运行结果与前一个程序清单相同，如图16.12所示。该程序清单演示了一种不同的编写代码的结构。这里提供并分析该程序清单，是为了让您在使用诸如Visual C#这样的开发工具，并发现其使用的结构与前面不同时，不会感到惊讶。

来看看该程序清单，其中的代码被划分成几个方法，而不是全部放在 Main 方法中。另外，这里没有专门声明一个窗体实例，而是在调用 Application.Run 方法的同时创建这种实例。

该应用程序被执行时，第 45 ~ 48 行的 Main 方法将首先被执行。该方法只有一行代码，它创建一个 frmApp 实例，并将其传递给 Application.Run 方法。这行代码将引发一系列其他的活动。首先，frmApp 的构造函数被调用，以创建一个新的 frmApp。构造函数位于第 10~13 行，它调用另一个方法 InitializeComponent，这导致第 17 ~ 43 行的代码被执行。这些代码除了很少一部分外，其余的与前一个程序清单完全相同。这里使用的不是窗体名，而是关键字 this。由于这是位于窗体实例内部，因此 this 指的是当前窗体。前一个清单中引用 myForm 实例的地方，这里使用的都是 this。当窗体中对象被初始化后，控制权回到构造函数，而此时构造函数也执行完毕，因此控制权回到 Main 方法，Main 方法将新初始化的 frmApp 对象传递给 Application.Run 方法。该方法将显示窗体，并负责 Windows 循环，直到程序终止。

这种结构的优点是，将所有组件和窗体的初始化代码放在一个与其他编程逻辑分开的方法中。对于大型程序，这样做的益处将更大。

16.4.3 使用按钮

按钮是 Windows 应用程序中最常用的控件之一。按钮可以使用 Button 类来创建。按钮不同于标签，您很可能希望当用户单击按钮时执行某种操作。

介绍如何创建按钮事件之前，有必要先花些时间来介绍如何创建和设置按钮。和标签一样，按钮的属性、数据成员和方法也非常多，这里无法一一列出，完整的列表请参阅帮助文档。表 16.5 列出了其中的一些属性。

表 16.5 Button 的一些属性

属 性	描 述
BackColor	返回或设置按钮的背景色
BackgroundImage	返回或设置显示在按钮背景上的图像
Bottom	返回按钮的最下边与其所在容器的最上边之间的距离
Enabled	返回一个指示按钮是否可用的值
Height	返回或设置按钮的高度
Image	返回或设置按钮上的图像
Left	返回或设置按钮左边框的位置
Right	返回或设置按钮右边框的位置
Text	返回或设置按钮上的文本
TextAlign	返回或设置按钮上文本的对齐方式
Top	返回或设置按钮上边框的位置
Visible	返回或设置一个指示按钮是否可见的值
Width	返回或设置按钮的宽度

注意：仔细查看表16.5列出的属性，您将发现它们与Label相应的属性类似。这种类似性是有道理的，所有的控件都是从一种更通用的类Control派生出来的，这个类让所有的控件都可以使用相同的方法或名称完成类似的任务。例如，不管是按钮、文本还是其他控件，其Top属性指的都是其上边框的位置。

16.4.3.1 按钮事件

前面介绍过，按钮不同于标签，您通常使用按钮来导致某种操作。您希望在用户单击按钮时执行某种操作，为导致这种操作发生，您使用事件。

创建按钮后，可以将一个或更多的事件与之关联起来，关联的方式与第14天介绍的相同。首先，需要创建处理事件的方法，该方法在事件发生时将被调用。正如第14天介绍的，这种方法必须接受两个参数——引发事件的对象和一个 System.EventArgs 变量。这种方法还必须是保护的，返回类型为 void。其格式如下：

```
protected void methodName( Object sender, System.EventArgs args)
```

在 Windows 应用程序中，通常基于导致事件的控件以及发生的事件给事件命名。例如，如果按钮 ABC 被单击，则事件处理程序的名称为 ABC_Click。

要激活事件，需要将它与合适的代表关联起来。一个名为 System.EventHandler 的代表对象负责看管所有的 Windows 事件。通过将事件处理程序与该代表对象关联起来，则在合适的时候，该事件处理程序将被调用。关联的格式如下：

```
ControlName.Event += new System.EventHandler( this.methodName);
```

其中, `ControlName.Event` 是控件名和该控件的事件名; `this` 是当前窗体; `methodName` 是处理该事件的方法。

程序清单 16.8 在程序清单 16.7 基础上进行了修改, 其运行结果如图 16.13 所示。从图中可知, 窗体中仍然显示了日期和时间, 同时包含一个按钮。当该按钮被单击时, 将引发一个更新日期和时间的事件。另外, 为增加趣味性, 该程序清单中还包含另外四个事件处理程序。它们分别在鼠标进入和离开这两个控件时被执行。

程序清单 16.8 button1.cs: 使用按钮和事件

```

1: // button1.cs - Working with buttons and events
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmApp : Form
9: {
10:     private Label myDateLabel;
11:     private Button btnUpdate;
12:
13:     public frmApp()
14:     {
15:         InitializeComponent();
16:     }
17:
18:     private void InitializeComponent()
19:     {
20:         this.Text = Environment.CommandLine;
21:         this.StartPosition = FormStartPosition.CenterScreen;
22:         this.FormBorderStyle = FormBorderStyle.Fixed3D;
23:
24:         myDateLabel = new Label();    // Create label
25:
26:         DateTime currDate = new DateTime();
27:         currDate = DateTime.Now;
28:         myDateLabel.Text = currDate.ToString();
29:
30:         myDateLabel.AutoSize = true;
31:         myDateLabel.Location = new Point( 50, 20);
32:         myDateLabel.BackColor = this.BackColor;
33:
34:         this.Controls.Add(myDateLabel); // Add label to form
35:
36:         // Set width of form based on Label's width
37:         this.Width = (myDateLabel.PreferredWidth + 100);
38:
39:         btnUpdate = new Button();    // Create a button
40:

```

```
41:     btnUpdate.Text = "Update";
42:     btnUpdate.BackColor = Color.LightGray;
43:     btnUpdate.Location = new Point(((this.Width/2) - (btnUpdate.Width / 2)),
44:                                     (this.Height - 75));
45:
46:     this.Controls.Add(btnUpdate); // Add button to form
47:
48:     // Add a click event handler using the default event handler
49:     btnUpdate.Click += new System.EventHandler(this.btnUpdate_Click);
50:     btnUpdate.MouseEnter += new System.EventHandler(this.btnUpdate_MouseEnter);
51:     btnUpdate.MouseLeave += new System.EventHandler(this.btnUpdate_MouseLeave);
52:
53:     myDateLabel.MouseEnter += new System.EventHandler(this.myDataLabel_MouseEnter);
54:     myDateLabel.MouseLeave += new System.EventHandler(this.myDataLabel_MouseLeave);
55: }
56:
57: protected void btnUpdate_Click( object sender, System.EventArgs e)
58: {
59:     DateTime currDate =DateTime.Now ;
60:     this.myDateLabel.Text = currDate.ToString();
61: }
62:
63:
64: protected void btnUpdate_MouseEnter( object sender, System.EventArgs e)
65: {
66:     this.BackColor = Color.HotPink;
67: }
68:
69: protected void btnUpdate_MouseLeave( object sender, System.EventArgs e)
70: {
71:     this.BackColor = Color.Blue;
72: }
73:
74: protected void myDataLabel_MouseEnter( object sender, System.EventArgs e)
75: {
76:     this.BackColor = Color.Yellow;
77: }
78:
79: protected void myDataLabel_MouseLeave( object sender, System.EventArgs e)
80: {
81:     this.BackColor = Color.Green;
82: }
83:
84:
85: public static void Main( string[] args )
86: {
87:     Application.Run( new frmApp() );
88: }
89: }
```

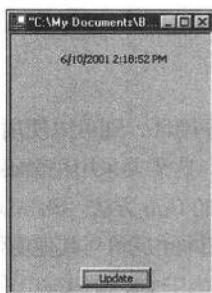



图 16.13 使用按钮和事件

分析：该程序清单采用的是Windows设计工具的格式，虽然这里并没有使用这样的设计工具。这是一种不错的格式化代码的方式，因此这里采用了这种格式。

您将发现，这里有一个与前一个程序清单不同的地方。第 10 和 11 行将按钮和标签声明为窗体的成员，而不是方法的成员，这样该窗体内的所有方法都可以使用这两个变量。它们是私有的，因此只能在这个类中使用。

Main 方法和构造函数与前一个程序清单相同，而 InitializeComponent 做了重大修改，不过大部分修改都很容易理解。首先，第 31 行使用一个 Point 对象来设置 myDateLabel 的位置，而不是使用 Top 和 Left 属性。创建的 Point 对象的值为(50, 20)，并被立刻赋给标签的 Location 属性。

提示：您将发现，与执行多次赋值操作相比，创建一个对象，并立刻给它赋值将更容易理解。这两种方式都管用，哪种方式最得心应手或最容易理解就使用哪一种。

第 39 行创建了一个名为 btnUpdate 的按钮，然后通过给它的几个属性赋值，对其进行定制。不要被第 43 ~ 44 行的计算所迷惑，这与第 31 行类似，只是未使用字面值，使用的是计算公式。另外需要记住的是，this 指的是窗体，因此 this.Width 指的是窗体的宽度。

第 46 行将按钮加入到窗体中，这与将其他按钮加入到窗体中完全相同。

第 49 ~ 54 行是该程序清单中有趣的部分，它们为事件指定处理程序。在这些赋值运算的左边，是控件及其事件，事件被指定为由传递给 System.EventHandler 的方法来处理。例如，在第 49 行，btnUpdate_Click 方法被指定来处理 btnUpdate 按钮的 Click 事件。第 50 和 51 行，指定了负责处理 btnUpdate 的 MouseEnter 和 MouseLeave 事件的方法。第 53 和 54 行指定了负责处理 myDateLabel 的 MouseEnter 和 MouseLeave 事件的方法。是的，标签控件也可以有事件！几乎所有的控件都有事件。

注意：与各类控件相关的事件非常多，本书无法一一列出。要知道有哪些事件，可参阅帮助文档。

要使事件起作用，必须创建与之相关联的方法。第 57 ~ 82 行是几个非常简单的方法，它们正是第 49 ~ 54 行中被关联到事件的方法。

16.4.3.2 创建 OK 按钮

OK 按钮是一种常用的、许多窗体都有的按钮。这种按钮在用户完成其操作后被单击。通常，这会导致窗体被关闭。

如果您创建窗体，并使用了 Application 类的 Run 方法，则可以为按钮创建一个终止 Run 方法的事件处理程序，该事件处理程序可以非常简单，如下所示：

```
protected void btnOK_Click( object sender, System.EventArgs e)
{
    // Final code logic before closing form
```

```

    Application.Exit(); // Ends the Application.Run message loop.
}

```

如果不想结束应用程序或应用程序循环，则可以使用窗体的 `Close` 方法，该方法关闭窗口。

还有另一种实现 OK 逻辑的方式，这需要采用稍微不同的方式。首先应该使用 `Form` 对象的 `ShowDialog` 方法，而不是 `Application` 类的 `Run` 方法。`ShowDialog` 方法显示一个对话框，并等待对话结束。对话框是一个窗体，创建这种窗体的逻辑与其他窗体相同。

通常，如果用户按 `Enter` 键，则相当于 OK 按钮被单击。您可以使用窗体的 `AcceptButton` 属性将按钮与 `Enter` 键关联起来，方法是将该属性设置为按 `Enter` 键将被激活的按钮。

16.4.4 使用文本框

另一个常用的控件是文本框，该控件用于获取用户输入的文本。使用文本框控件及其事件可以从用户那里获取信息，然后使用这些信息。程序清单 16.9 演示了如何使用文本框控件，其运行结果如图 16.14 所示。

程序清单 16.9 `text1.cs`: 使用文本框控件

```

1: // text1.cs - Working with text controls
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmGetName : Form
9: {
10:     private Button btnOK;
11:
12:     private Label lblFirst;
13:     private Label lblMiddle;
14:     private Label lblLast;
15:     private Label lblFullName;
16:     private Label lblInstructions;
17:
18:     private TextBox txtFirst;
19:     private TextBox txtMiddle;
20:     private TextBox txtLast;
21:
22:     public frmGetName()
23:     {
24:         InitializeComponent();
25:     }
26:
27:     private void InitializeComponent()
28:     {
29:         this.FormBorderStyle = FormBorderStyle.Fixed3D;
30:         this.Text = "Get User Name";
31:         this.StartPosition = FormStartPosition.CenterScreen;
32:

```

```
33:         // Instantiate the controls...
34:         lblInstructions = new Label();
35:         lblFirst      = new Label();
36:         lblMiddle     = new Label();
37:         lblLast       = new Label();
38:         lblFullName   = new Label();
39:
40:         txtFirst      = new TextBox();
41:         txtMiddle     = new TextBox();
42:         txtLast       = new TextBox();
43:
44:         btnOK = new Button();
45:
46:         // Set properties
47:
48:         lblFirst.AutoSize = true;
49:         lblFirst.Text      = "First Name:";
50:         lblFirst.Location = new Point( 20, 20);
51:
52:         lblMiddle.AutoSize = true;
53:         lblMiddle.Text     = "Middle Name:";
54:         lblMiddle.Location = new Point( 20, 50);
55:
56:         lblLast.AutoSize = true;
57:         lblLast.Text      = "Last Name:";
58:         lblLast.Location = new Point( 20, 80);
59:
60:         lblFullName.AutoSize = true;
61:         lblFullName.Location = new Point( 20, 110 );
62:
63:         txtFirst.Width = 100;
64:         txtFirst.Location = new Point(140, 20);
65:
66:         txtMiddle.Width = 100;
67:         txtMiddle.Location = new Point(140, 50);
68:
69:         txtLast.Width = 100;
70:         txtLast.Location = new Point(140, 80);
71:
72:         lblInstructions.Width = 250;
73:         lblInstructions.Height = 60;
74:         lblInstructions.Text = "Enter your first, middle, and last name."
75:             + "\nYou will see your name appear as you type."
76:             + "\nFor fun, edit your name after entering it.";
77:         lblInstructions.TextAlign = ContentAlignment.MiddleCenter;
78:         lblInstructions.Location =
79:             new Point(((this.Width/2) - (lblInstructions.Width / 2 )), 140);
80:
81:         this.Controls.Add(lblFirst);    // Add label to form
82:         this.Controls.Add(lblMiddle);
```

```

83:     this.Controls.Add(lblLast);
84:     this.Controls.Add(lblFullName);
85:     this.Controls.Add(txtFirst);
86:     this.Controls.Add(txtMiddle);
87:     this.Controls.Add(txtLast);
88:     this.Controls.Add(lblInstructions);
89:
90:     btnOK.Text = "Done";
91:     btnOK.BackColor = Color.LightGray;
92:     btnOK.Location = new Point(((this.Width/2) - (btnOK.Width / 2)),
93:                               (this.Height - 75));
94:
95:     this.Controls.Add(btnOK); // Add button to form
96:
97:     // Event handlers
98:     btnOK.Click += new System.EventHandler(this.btnOK_Click);
99:     txtFirst.TextChanged += new
System.EventHandler(this.txtChanged_Event);
100:    txtMiddle.TextChanged += new
System.EventHandler(this.txtChanged_Event);
101:    txtLast.TextChanged += new
System.EventHandler(this.txtChanged_Event);
102: }
103:
104: protected void btnOK_Click( object sender, System.EventArgs e)
105: {
106:     Application.Exit();
107: }
108:
109: protected void txtChanged_Event( object sender, System.EventArgs e)
110: {
111:     lblFullName.Text = txtFirst.Text + " " + txtMiddle.Text + " " + txtLast.Text;
112: }
113:
114: public static void Main( string[] args )
115: {
116:     Application.Run( new frmGetName() );
117: }
118: }

```

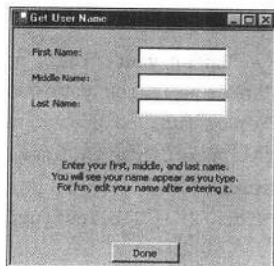


图 16.14 使用文本框控件

分析：从结果可知，您创建的应用程序开始变得很有用。该程序清单中的文本框控件让用户能够输入其姓名，然后，这些姓名被组合起来，并被显示到屏幕上。

虽然该程序清单很长，但大部分代码是重复的，这是由于其中包含的三个用于接受姓、名和中名的文本框控件非常类似。第 10 ~ 20 行，在 `frmGetName` 类中声明了多个控件，这些控件在 `InitializeComponent` 方法中被实例化和赋值（第 34 ~ 44 行）。第 48 ~ 58 行给指示姓、名和中名文本框的三个标签赋值。首先将它们的 `AutoSize` 属性设置为 `true`，以便能够足以容纳信息；然后设置了 `Text` 属性；最后设置了它们在窗体上的位置，它们都是离窗体左边框 20 个像素，但垂直位置各不相同。

第 60 ~ 61 行声明了一个用于显示全名的标签，此时其 `Text` 属性没有被赋值，这将在事件发生时进行。

第 63 ~ 70 行指定文本框的位置和宽度，正如您看到的，指定方式与前面介绍的其他控件相同。

第 72 ~ 79 行加入了另一个用于显示操作说明的标签，不要被这里的代码所迷惑。第 74 行将该控件的 `Text` 属性设置为三行文本，实际上，这只不过是一个非常长的字符串，为方便阅读，而被拆分而已。加号将三部分组合成一个字符串，然后将其赋给 `lblInstructions.Text` 属性。第 77 行使用了一个以前没有介绍过的属性——`TextAlign`，它设置控件中文本的对齐方式。该属性被设置为 `ContentAlignment` 枚举中的一个值，这里是 `MiddleCenter`。`ContentAlignment` 枚举中的其他值还有 `BottomCenter`、`BottomLeft`、`BottomRight`、`MiddleLeft`、`MiddleRight`、`TopCenter`、`TopLeft` 和 `TopRight` 等。

警告：虽然不同的控件可能包含名称相同的属性，但这些属性接受的值可能不同。例如，标签控件的 `TextAlign` 属性接受的是枚举 `ContentAlignment` 的值；而文本框控件的 `TextAlign` 属性接受的却是枚举 `HorizontalAlignment` 的值。

第 98 ~ 101 行加入了事件处理程序。第 98 行加入的是 `btnOK` 按钮的 `Click` 事件的处理程序，它位于第 104 ~ 107 行。该方法结束应用程序循环，从而结束程序。

第 99 ~ 101 行加入了文本框的 `TextChanged` 事件的处理程序。每当这三个文本框中某一个的文本发生变化，`txt_Changed_Event` 都将被调用。正如您看到的，同一个方法可以用作多个事件处理程序。该方法将三个文本框中的内容组合起来，并将结果赋给 `lblFullName.Text` 属性。

16.4.5 使用其他控件

程序清单 16.9 包含构建基本应用程序所需的東西，您还可以使用许多其他的控件。控件的基本用法在很大程度上与今天各个程序清单介绍的相同——创建控件、修改属性、为需要对其做出反应的事件创建处理程序，最后将控件放到窗体上。一些控件（如列表框）在设置初始值方面稍微复杂些，但使用的整个过程相同。

正如前面指出的，介绍所有的控件及其功能需要一本专门的非常厚的书。要详细了解这些控件，可以从帮助文档开始。虽然过深地介绍基于 Windows 编程技术超出了本书的范围，但由于这种技术非常流行，因此有必要在介绍 Web 表单和服务之前，在明天的课程中介绍其他的一些 Windows 主题。

16.5 总 结

今天介绍了许多有趣的内容。正如您知道的，使用名称空间 `System.Windows.Forms` 中定义的类、方法、属性和事件，只需很少的代码便可以创建出基于 Windows 的应用程序。今天介绍了如何创建和定制窗体、如何将基本控件加入到窗体中以及使用事件给窗体提供功能。虽然介绍的控件不多，但使用其他控件的方式在很大程度上与今天介绍的控件相同。

明天将扩展今天介绍的知识，而第 18 天将介绍 Windows 窗体和 Web 表单之间的差别。

16.6 问与答

问：从哪里可以了解到更多关于 Windows 窗体的知识？

答：可以从 .NET SDK 自带的文档中了解到更多关于 Windows 窗体的知识，这包括 Windows 窗体快速入门（Quick Start）。

问：我注意到，控件表中包含了 Form，这是为什么？

答：窗体也是一种控件，窗体具备大部分控件功能。

问：为什么不列出今天介绍的控件的所有属性、事件和方法？

答：.NET 框架类中包含 40 多个控件。另外，这些控件包含的方法、事件和属性超过 100 个。要介绍所有这些东西，即使其中每一项只占一行，也需要 80 多页的篇幅。

16.7 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

16.7.1 小测验

1. 大部分 Windows 控件都位于哪个名称空间中？
2. 哪个方法用于显示窗体？
3. 将控件加入到窗体中包括哪三步？
4. 将文件 xyz.cs 编译为 Windows 程序时，需要使用的命令行命令是什么？
5. 如果要在编译文件 xyz.cs 时包含组合体 myAssemb.dll，应该使用的命令行命令是什么？
6. Form 类的 Show() 有何功能？使用这种方法存在什么样的问题？
7. 下面哪种东西会导致 Application.Run 结束？
 - a. 方法；
 - b. 事件；
 - c. 到达程序的最后一行；
 - d. 该方法永远不会结束；
 - e. 以上答案都不对。
8. 哪些颜色属于窗体？要使用这些颜色，需要包含哪个名称空间？
9. 哪个属性用于设置标签的文本值？
10. 文本框和标签之间有何差别？

16.7.2 练习

1. 尽您的能力编写一个最短的 Windows 应用程序？
2. 创建一个程序，将一个 200X200 像素的窗体放在屏幕中央。
3. 创建一个窗体，该窗体包含一个可用来输入数字的文本框，当用户单击按钮后，在标签中显示一条消息，指出该数字是否位于 0 ~ 100 之间。
4. 下面的程序有问题。在编辑器中输入该程序，然后编译它。哪一行导致错误消息？

```
1: using System.Windows.Forms;
2:
3: public class frmHello : Form
4: {
5:     public static void Main( string[] args )
6:     {
7:         frmHello frmHelloApp = new frmHello();
8:         frmHelloApp.Show();
9:     }
10: }
11: }
```

第 17 天课程

创建 Windows 应用程序

昨天介绍了如何创建 Windows 窗体以及在其中添加控件,今天将介绍其他几种控件以及改进窗体的几种方式,包括以下内容:

- 单选按钮组;
- 容器;
- 在列表框控件中添加选项;
- 通过添加菜单改进应用程序;
- MessageBox 类;
- 如何使用已有的对话框。

注意:今天将继续介绍Windows窗体功能,但这只是为您更全面地学习这方面的知识打下一定的基础。全面介绍控件和Windows窗体功能将需要1000页的篇幅,但您将发现大部分功能与这两天介绍的类似。

17.1 使用单选按钮

昨天介绍了如何创建一些基本控件,您被告知,大多数控件有相同的创建和实现步骤:

1. 实例化控件对象;
2. 设置属性值;
3. 将控件对象加入到窗体中。

单选按钮的创建方式与此相同。单选按钮通常成组地使用,选中其中的一个按钮将取消对组中其他按钮的选中。因此,在用户可以从有限的选项中选择一个时,这种控件很有用;当需要显示所有的选项时——如选择性别(男或女)或选择婚姻状态(已婚或未婚),这种控件也很方便。

17.1.1 将单选按钮分组

单选按钮不同于其他控件的地方在使用方式。通常您将一系列单选按钮作为一组,并希望其中的一个单选按钮被选中时,其他的单选按钮将不被选中。程序清单 17.1 演示了一个包含两组单选按钮的窗体,该程序清单的主窗体如图 17.1 所示。

程序清单 17.1 radio1.cs: 使用单选按钮并将其分组

```
1: // radio1.cs - Using Radio Buttons
```

21 天学通 C#


```
2: //      - Not quite right...
3: //-----
4:
5: using System.Drawing;
6: using System.Windows.Forms;
7:
8: public class Form1 : Form
9: {
10:     private RadioButton rdMale;
11:     private RadioButton rdFemale;
12:     private RadioButton rdYouth;
13:     private RadioButton rdAdult;
14:     private Button btnOK;
15:     private Label lblText1;
16:     private Label lblText2;
17:
18:     public Form1()
19:     {
20:         InitializeComponent();
21:     }
22:
23:     private void InitializeComponent()
24:     {
25:         this.rdMale = new System.Windows.Forms.RadioButton();
26:         this.rdFemale = new System.Windows.Forms.RadioButton();
27:         this.lblText1 = new System.Windows.Forms.Label();
28:         this.rdYouth = new System.Windows.Forms.RadioButton();
29:         this.rdAdult = new System.Windows.Forms.RadioButton();
30:         this.lblText2 = new System.Windows.Forms.Label();
31:         this.btnOK = new System.Windows.Forms.Button();
32:
33:         // Form1
34:         this.ClientSize = new System.Drawing.Size(350, 225);
35:         this.Text = "Radio Buttons 1";
36:
37:         // rdMale
38:         this.rdMale.Location = new System.Drawing.Point(50, 65);
39:         this.rdMale.Size = new Size(90, 15);
40:         this.rdMale.TabIndex = 0;
41:         this.rdMale.Text = "Male";
42:
43:         // rdFemale
44:         this.rdFemale.Location = new System.Drawing.Point(50, 90);
45:         this.rdFemale.Size = new System.Drawing.Size(90, 15);
46:         this.rdFemale.TabIndex = 1;
47:         this.rdFemale.Text = "Female";
48:
49:         // lblText1
50:         this.lblText1.Location = new System.Drawing.Point(50, 40);
51:         this.lblText1.Size = new System.Drawing.Size(90, 15);
```

```
52:     this.lblText1.TabIndex = 2;
53:     this.lblText1.Text     = "Sex";
54:
55:     // rdYouth
56:     this.rdYouth.Location = new System.Drawing.Point(220, 65);
57:     this.rdYouth.Size     = new System.Drawing.Size(90, 15);
58:     this.rdYouth.TabIndex = 3;
59:     this.rdYouth.Text     = "Over 21";
60:
61:
62:     // rdAdult
63:     this.rdAdult.Location = new System.Drawing.Point(220, 90);
64:     this.rdAdult.Size     = new System.Drawing.Size(90, 15);
65:     this.rdAdult.TabIndex = 4;
66:     this.rdAdult.Text     = "Under 21";
67:
68:     // lblText2
69:     this.lblText2.Location = new System.Drawing.Point(220, 40);
70:     this.lblText2.Size     = new System.Drawing.Size(90, 15);
71:     this.lblText2.TabIndex = 5;
72:     this.lblText2.Text     = "Age Group";
73:
74:     // btnOK
75:     this.btnOK.Location = new System.Drawing.Point(130, 160);
76:     this.btnOK.Size     = new System.Drawing.Size(70, 30);
77:     this.btnOK.TabIndex = 6;
78:     this.btnOK.Text     = "OK";
79:     this.btnOK.Click += new System.EventHandler(this.btnOK_Click);
80:
81:     this.Controls.Add(rdMale);
82:     this.Controls.Add(rdFemale);
83:     this.Controls.Add(lblText1);
84:     this.Controls.Add(rdYouth);
85:     this.Controls.Add(rdAdult);
86:     this.Controls.Add(lblText2);
87:     this.Controls.Add(btnOK);
88: }
89:
90: private void btnOK_Click(object sender, System.EventArgs e)
91: {
92:     Application.Exit();
93: }
94:
95: static void Main()
96: {
97:     Application.Run(new Form1());
98: }
99: }
100:
```

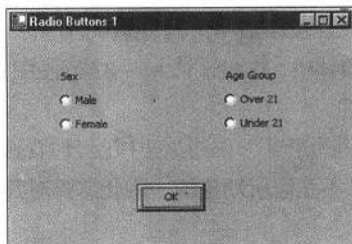


图 17.1 使用单选按钮

分析：该程序清单创建了一个包含四个单选按钮和一个命令按钮的窗体，该窗体中还包含两个标签控件，用于给用户提供描述性信息。运行该程序，您将发现单选按钮的运行方式与您期望的相同——选中其中的一个单选按钮，将取消对其他按钮的选中，这是单选按钮的标准运行方式。但存在的问题是，选中性别类中的两个按钮之一，将取消对年龄类中按钮的选中。如果能够将这两类按钮分开将更佳。介绍如何解决这种问题之前，有必要花几分钟的时间解释一下该程序清单。

第 5 和 6 行使用 `using` 语句将名称空间 `System` 中的名称空间 `Drawing` 和 `Windows.Forms` 包含进来。包含 `Drawing` 名称空间可以简化 `Point` 类的名称；而包含名称空间 `Windows.Forms` 可以简化窗体和控件的名称。

从第 8 行开始是一个类，该行定义了一个从 `Form` 类派生而来的新窗体类 `Form1`。

第 10~16 行为 `Form1` 类声明了几个私有成员，包括四个单选按钮（第 10~13 行）、一个 OK 按钮（第 14 行）和两个标签（第 15 和 16 行）。后面实例化了这些控件，并定义了它们的值。

第 18~21 行是窗体的标准构造函数，您对它应该很熟悉，因为其格式与昨天的程序清单中使用的相同。它调用 `InitializeComponent` 方法，该方法完成应用程序窗体的所有建立工作。

`InitializeComponent` 方法首先实例化每个控件（第 25~31 行）。注意到这里使用了构造函数的显式名称。由于前面的 `using` 语句，这里可以只使用构造函数的名称即可。例如，对于第 25 行，可以修改为如下所示：

```
this.rdMail = new RadioButton();
```

这里之所以使用全限定名称，是为了让读者知道使用了哪个名称空间。整个程序清单都是这样做的。

从第 34 行开始设置窗体的值。第 34 行设置了应用程序窗体的大小，通过将一个 `Size` 对象赋给当前窗体（`this`）的 `ClientSize` 属性，来设置该属性的值。第 35 行设置了该窗体的 `Text` 属性。记住，`Text` 属性用于设置窗体的标题。

第 25 行实例化第一个单选按钮，该按钮的值是从第 38 行开始设置的。首先通过将一个 `Point` 对象赋给该单选按钮的 `Location` 属性来设置其位置。`Point` 对象使用 `x` 和 `y` 指定窗体中的位置，这里是离窗体左上角向右 50 个像素，向下 65 个像素。第 39 行设置了该控件的大小，这是给按钮和文本提供的空间。第 40 行设置了 `TabIndex` 属性。最后，第 41 行设置了按钮包含的文本，这里为 `Male`。接下来的代码行以相同的方式设置了单选按钮 `rdFemal`、`rdYouth` 和 `rdAdult`。

注意：大多数控件都有 `TabIndex` 属性，该属性设置用户按下 `Tab` 键时，控件被选中的顺序。首先被选中的控件的索引为 0，第二个为 1，第三个为 2 等等。设置 `TabIndex` 可以控制用户在窗体的控件间导航的顺序。

第 49~53 行和第 69~72 行分别建立了窗体中的两个标签。第 75~79 行建立了 OK 按钮，并

具有这样一个事件处理程序，即在该按钮被单击时，结束程序。

最后，将所有的控件加入到窗体中，这是在第 81 ~ 87 行使用 `Controls.Add` 完成的。至此，窗体被初始化，并可以显示所有的控件。

这里不需要提供操纵单选按钮的逻辑，单选按钮中已经包含选中和取消选中所需的代码。前面指出过，该程序清单的运行情况并不是您所期望的，因此需要对其进行修改，以使两组单选按钮独立地运行。

17.1.2 使用容器

对于程序清单 17.1 中的问题，解决的办法是使用容器。容器让您能够将控件分组，您实际上已经使用过容器，这就是主窗体。您也可以创建自己的容器，并将其放在窗体容器和其他容器中。通过将程序清单 17.1 中的两组单选按钮放在各自的容器中，可以将它们分开。

也可以使用另一种控件——分组框，将控件分开。分组框的工作方式与容器相同，但还包含其他的功能，如显示文本标签。

程序清单 17.2 删除了标签控件，而是使用分组框。另外，该程序清单也没有使用显式名称。程序清单 17.1 之所以使用显式名称，是为了让您知道各种类和类型的存储位置。图 17.2 显示了单选按钮被放置在分组框中的情况。

程序清单 17.2 radio2.cs: 将单选按钮分组

```
1: // radio2.cs - Using Radio Buttons
2: //           - Using Groups
3: //-----
4:
5: using System.Drawing;
6: using System.Windows.Forms;
7:
8: public class Form1 : Form
9: {
10:     private GroupBox gboxAge;
11:     private GroupBox gboxSex;
12:
13:     private RadioButton rdMale;
14:     private RadioButton rdFemale;
15:     private RadioButton rdYouth;
16:     private RadioButton rdAdult;
17:     private Button btnOK;
18:
19:     public Form1()
20:     {
21:         InitializeComponent();
22:     }
23:
24:     private void InitializeComponent()
25:     {
26:         this.gboxAge = new GroupBox();
27:         this.gboxSex = new GroupBox();
28:         this.rdMale = new RadioButton();
```

```
29:     this.rdFemale = new RadioButton();
30:     this.rdYouth = new RadioButton();
31:     this.rdAdult = new RadioButton();
32:     this.btnOK = new Button();
33:
34:     // Form1
35:     this.ClientSize = new Size(350, 200);
36:     this.Text = "Grouping Radio Buttons";
37:
38:     // groupBox
39:     this.groupBox.Location = new Point(15, 30);
40:     this.groupBox.Size = new Size(125, 100);
41:     this.groupBox.TabStop = false;
42:     this.groupBox.Text = "Sex";
43:
44:     // rdMale
45:     this.rdMale.Location = new Point(35, 35);
46:     this.rdMale.Size = new Size(70, 15);
47:     this.rdMale.TabIndex = 0;
48:     this.rdMale.Text = "Male";
49:
50:     // rdFemale
51:     this.rdFemale.Location = new Point(35, 60);
52:     this.rdFemale.Size = new Size(70, 15);
53:     this.rdFemale.TabIndex = 1;
54:     this.rdFemale.Text = "Female";
55:
56:     // groupBox
57:     this.groupBox.Location = new Point(200, 30);
58:     this.groupBox.Size = new Size(125, 100);
59:     this.groupBox.TabStop = false;
60:     this.groupBox.Text = "Age Group";
61:
62:     // rdYouth
63:     this.rdYouth.Location = new Point(35, 35);
64:     this.rdYouth.Size = new Size(70, 15);
65:     this.rdYouth.TabIndex = 3;
66:     this.rdYouth.Text = "Over 21";
67:
68:     // rdAdult
69:     this.rdAdult.Location = new Point(35, 60);
70:     this.rdAdult.Size = new Size(70, 15);
71:     this.rdAdult.TabIndex = 4;
72:     this.rdAdult.Text = "Under 21";
73:
74:     // btnOK
75:     this.btnOK.Location = new Point(130, 160);
76:     this.btnOK.Size = new Size(70, 30);
77:     this.btnOK.TabIndex = 6;
78:     this.btnOK.Text = "OK";
```

```

79:     this.btnOK.Click += new System.EventHandler(this.btnOK_Click);
80:
81:     this.Controls.Add(gboxSex);
82:     this.Controls.Add(gboxAge);
83:
84:     this.gboxSex.Controls.Add(rdMale);
85:     this.gboxSex.Controls.Add(rdFemale);
86:     this.gboxAge.Controls.Add(rdYouth);
87:     this.gboxAge.Controls.Add(rdAdult);
88:
89:     this.Controls.Add(btnOK);
90: }
91:
92: private void btnOK_Click(object sender, System.EventArgs e)
93: {
94:     Application.Exit();
95: }
96:
97: static void Main()
98: {
99:     Application.Run(new Form1());
100: }
101: }
102:

```

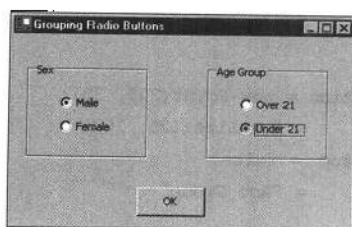


图 17.2 将单选按钮分组

分析：该程序清单的重点是分组框的用法。如果使用容器，而不是分组框，则代码将与此类似，只是不能设置容器的Text和其他属性，而是需要使用其他控件（如标签）来定制容器的外观。

第 10 和 11 行声明了两个将用于窗体的分组框成员。这里删除了标签控件，因为文本描述可以包含在分组框中。第 26 和 27 行调用 GroupBox 的构造函数实例化这两个分组框。

第 39~42 行设置第一个分组框控件 gboxSex 的特征。和其他控件一样，也可以设置 Location、Size、TabIndex 和 Text 等属性，当然也可以设置其他属性。第 57~60 行设置分组框 gboxAge 的属性。

提示：有关组合框的属性和成员的完整列表，请参阅在线文档。和其他控件一样，该控件的属性和成员太多，这里无法一一列出。

该程序清单与前一个的最大差别是第 81~87 行。第 81 和 82 行将两个分组框（gboxSex 和 gboxAge）加入到窗体中。第 84~87 行不是将单选按钮加入到窗体中，而是加入到分组框中。分组框位于窗体中，因此单选按钮将出现在窗体的分组框中。

将单选按钮加入到分组框中后，它们便位于不同的容器中了。运行该程序清单时，选择 Sex 中

的选项将不会影响 Age。

注意：可以使得对 Sex 选项的选择影响到 Age 选项，方法是，加入这样的一个事件，即包含对其他控件的属性进行操纵的代码。

17.2 使用列表框

另一个您想在窗体上使用的常见控件是列表框，它让您能够在一个小框中列出很多的选项。可以将列表框设置为在选项间滚动，还可以设置成允许用户选择一个或多个选项。由于设置列表框属性的工作量大些，因此有必要简要地介绍其基本功能。

设置列表框的方式与前面介绍的其他控件相同，首先将列表框控件指定为窗体的一部分：

```
private ListBox myListBox;
```

然后实例化它：

```
myListBox = new ListBox( );
```

当然可以在一行中完成这两项工作：

```
private ListBox myListBox = new ListBox( );
```

创建列表框后，便可以将其加入到窗体中。添加的方式与其他控件相同：

```
this.Controls.Add(myListBox);
```

列表框的不同之处在于，您希望它包含一系列选项。

17.2.1 在列表框中添加选项

在列表框中添加选项包括几个步骤。首先，需要告诉列表框，您要更新它。这是通过调用列表框的 `BeginUpdate` 方法实现的。对于名为 `myListBox` 的列表框，代码如下：

```
myListBox.BeginUpdate( );
```

调用上述方法后，便可以调用列表框的 `Items` 成员的 `Add` 方法，将选项加入到列表框中。这比想象的要容易。要将 `My First Item` 加入到 `myListBox` 中，使用的代码如下：

```
myListBox.Items.Add("My First Item");
```

可以以同样的方式加入其他的选项：

```
myListBox.Items.Add("My Second Item");
```

```
myListBox.Items.Add("My Third Item");
```

添加完选项后，需要告诉列表框。这是通过调用 `EndUpdate` 方法实现的：

```
myListBox.EndUpdate( );
```

仅此而已。这就是在列表框中添加选项需要做的所有工作。程序清单 17.3 使用了两个列表框，从图 17.3 的输出可以知道，这两个列表框的外观不同。

程序清单 17.3 list1.cs: 使用列表框

```
1: // list1.cs - Working with list box controls
```

```
2: //-----
3:
4: using System.Windows.Forms;
5: using System.Drawing;
6:
7: public class frmGetName : Form
8: {
9:     private Button btnOK;
10:    private Label lblFullName;
11:    private TextBox txtFullName;
12:    private ListBox lboxSex;
13:    private Label lblSex;
14:    private ListBox lboxAge;
15:
16:    public frmGetName()
17:    {
18:        InitializeComponent();
19:    }
20:
21:    private void InitializeComponent()
22:    {
23:        this.FormBorderStyle = FormBorderStyle.Fixed3D;
24:        this.Text = "Get User Info";
25:        this.StartPosition = FormStartPosition.CenterScreen;
26:
27:        // Instantiate the controls...
28:        lblFullName = new Label();
29:        txtFullName = new TextBox();
30:        btnOK = new Button();
31:        lblSex = new Label();
32:        lboxSex = new ListBox();
33:        lboxAge = new ListBox();
34:
35:        // Set properties
36:        lblFullName.Location = new Point(20, 40);
37:        lblFullName.AutoSize = true;
38:        lblFullName.Text = "Name:";
39:
40:        txtFullName.Width = 170;
41:        txtFullName.Location = new Point(80, 40);
42:
43:
44:        btnOK.Text = "Done";
45:        btnOK.Location = new Point(((this.Width/2) - (btnOK.Width / 2)),
46:                                   (this.Height - 75));
47:
48:        lblSex.Location = new Point(20, 70);
49:        lblSex.AutoSize = true;
50:        lblSex.Text = "Sex:";
51:
```



```

52:         // Set up ListBox
53:         listBoxSex.Location = new Point(80, 70);
54:         listBoxSex.Size = new Size(100, 20);
55:         listBoxSex.SelectionMode = SelectionMode.One;
56:
57:         listBoxSex.BeginUpdate();
58:         listBoxSex.Items.Add(" ");
59:         listBoxSex.Items.Add(" Boy ");
60:         listBoxSex.Items.Add(" Girl ");
61:         listBoxSex.Items.Add(" Man ");
62:         listBoxSex.Items.Add(" Lady ");
63:         listBoxSex.EndUpdate();
64:
65:         // Set up ListBox
66:         listBoxAge.Location = new Point(80, 100);
67:         listBoxAge.Size = new Size(100, 60);
68:         listBoxAge.SelectionMode = SelectionMode.One;
69:         listBoxAge.BeginUpdate();
70:         listBoxAge.Items.Add(" ");
71:         listBoxAge.Items.Add(" Under 21 ");
72:         listBoxAge.Items.Add(" 21 ");
73:         listBoxAge.Items.Add(" Over 21 ");
74:         listBoxAge.EndUpdate();
75:         listBoxAge.SelectedIndex = 0;
76:
77:         this.Controls.Add(btnOK); // Add button to form
78:         this.Controls.Add(lblFullName);
79:         this.Controls.Add(txtFullName);
80:         this.Controls.Add(listBoxSex);
81:         this.Controls.Add(lblSex);
82:         this.Controls.Add(listBoxAge);
83:
84:         // Event handlers
85:         btnOK.Click += new System.EventHandler(this.btnOK_Click);
86:     }
87:
88:     protected void btnOK_Click( object sender, System.EventArgs e)
89:     {
90:         Application.Exit();
91:     }
92:
93:     public static void Main( string[] args )
94:     {
95:         Application.Run( new frmGetName() );
96:     }
97: }

```

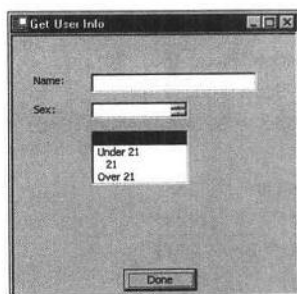


图 17.3 使用列表框

分析：该程序清单使用列表框来显示关于年龄和性别的选项，还包含一个文本框，让用户输入其姓名。从图17.3可知，这两个列表框的默认选项都为空白。这两个列表框的第一个选项都是空白值。

第9~14行定义了该应用程序的窗体。第12和14行声明了列表框。第28~33行实例化了程序使用的所有控件。两个列表框是分别第32和33行声明的。

接下来设置了两个列表框的细节。其中 `lboxSex` 是在第53~63行设置的。第53和54行设置了其位置和大小；第55行设置了选择模式，表17.1列出了列表框的各种选择模式。在第55行，`lboxSex` 的选择模式被设置为 `SelectionMode.One`，即只能同时选中一个选项。

表 17.1 列表框的选择模式

模 式	描 述
<code>SelectionMode.One</code>	只能同时选中一个选项
<code>SelectionMode.MultiExtended</code>	可以选中多个选项。可以使用 Shift、Ctrl 和方向键来选中多个选项
<code>SelectionMode.MultiSimple</code>	可以选中多个选项
<code>SelectionMode.None</code>	不能选中任何选项

第57~63行给 `lboxSex` 列表框加入了各种选项，添加的方式与前面介绍的相同。首先调用 `BeginUpdate` 方法，然后使用 `Items.Add` 方法加入各个选项，最后调用 `EndUpdate` 方法结束添加操作。

第66~75行以类似的方式设置列表框 `lboxAge`，但有两个明显的差别。首先，第75行设置了 `lboxAge` 的 `SelectedIndex` 属性，该属性决定了列表框的哪个选项被默认选中。其次是控件的大小。`lboxSex` 被设置为长100，宽20（第54行）；而 `lboxAge` 被设置为长100，宽60（第67行）。从显示的窗体中可以看到这样设置的结果（图17.3）。由于其高度较小，`lboxSex` 只能同时显示一个选项；由于无法同时显示所有的选项，因此自动添加了垂直滚动条。`lboxAge` 足够大，因此不需要垂直滚动条。

列表框的其他方面与其他控件类似。可以创建在被选择的选项发生变化以及用户离开控件时发生的事件，还可以在窗体中添加确保用户必须做出选择的逻辑，等等。

17.3 在窗体中加入菜单

控件是使窗体能够实现功能的方式之一，另一种方式是添加菜单。大多数 Windows 应用程序都包括某种类型的菜单，通常至少包括菜单 File 和 Help。这些菜单被选择后，通常将显示一组子菜单，供用户选择。您也可以在窗体中加入菜单。

17.3.1 创建一个基本菜单

程序清单 17.4 包含一个昨天介绍的显示当前日期和时间的窗体（见图 17.4）。该窗体使用菜单项，而不是按钮来显示当前的日期和时间。虽然这里使用菜单项不是非常好，但说明了许多重要的东西。首先，在窗体中添加一个菜单；然后讨论了如何将菜单项和事件关联起来；最后您将知道如何给菜单项事件编写代码。

程序清单 17.4 menu1.cs: 基本菜单

```

1: // menu1.cs - menus
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmApp : Form
9: {
10:     private Label myDateLabel;
11:     private MainMenu myMainMenu;
12:
13:     public frmApp()
14:     {
15:         InitializeComponent();
16:     }
17:
18:     private void InitializeComponent()
19:     {
20:         this.Text = "STY Menus";
21:         this.StartPosition = FormStartPosition.CenterScreen;
22:         this.FormBorderStyle = FormBorderStyle.Fixed3D;
23:
24:         myDateLabel = new Label();    // Create label
25:
26:         DateTime currDate = new DateTime();
27:         currDate = DateTime.Now;
28:         myDateLabel.Text = currDate.ToString();
29:
30:         myDateLabel.AutoSize = true;
31:         myDateLabel.Location = new Point( 50, 70);
32:         myDateLabel.BackColor = this.BackColor;
33:
34:         this.Controls.Add(myDateLabel); // Add label to form

```

```
35:
36:     // Set width of form based on Label's width
37:     this.Width = (myDateLabel.PreferredWidth + 100);
38:
39:     myMainMenu = new MainMenu();
40:
41:     MenuItem menuItemFile = myMainMenu.MenuItems.Add("File");
42:     menuItemFile.MenuItems.Add(new MenuItem("Update Date",
43:         new
EventHandler(this.MenuUpdate_Selection)));
44:     menuItemFile.MenuItems.Add(new MenuItem("Exit",
45:         new
EventHandler(this.FileExit_Selection)));
46:     this.Menu = myMainMenu;
47: }
48:
49: protected void MenuUpdate_Selection( object sender, System.EventArgs e )
50: {
51:     DateTime currDate = new DateTime();
52:     currDate = DateTime.Now;
53:     this.myDateLabel.Text = currDate.ToString();
54: }
55: protected void FileExit_Selection( object sender, System.EventArgs e )
56: {
57:     this.Close();
58: }
59:
60: public static void Main( string[] args )
61: {
62:     Application.Run( new frmApp() );
63: }
64: }
```

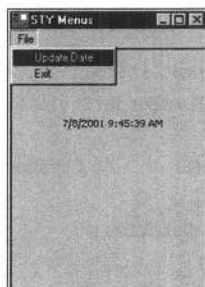


图 17.4 选择窗体中的基本菜单

分析: 从图17.4可知, 该程序清单创建了一个简单的菜单, 它包含两个选项: Update Date和Exit。选择Update Date将更新窗体中显示的日期和时间; 选择Exit将调用this.Close()来关闭当前窗体, 从而结束程序。

窗体中的基本菜单被称为主菜单, 包括 File 以及其他选项, 它们出现在窗体的顶部。第 11 行

为窗体 `frmApp` 声明了一个名为 `myMainMenu` 的 `MainMenu` 数据成员。

第 39 行实例化数据成员 `myMainMenu`, 第 41 ~ 44 行设置其属性, 而第 46 行将其加入到窗体中

第 41 ~ 46 行完成了大量的工作。事实上, 仅第 41 行便完成了许多工作。该行声明了一个名为 `menuItemFile` 的数据成员。另外, 下面的语句将一个新的菜单项(第一个菜单项)加入到 `myMainMenu` 菜单中:

```
myMainMenu.MenuItems.Add("File");
```

该菜单项名为 `File`, 并被赋给数据成员 `menuItemFile`。

概括地说, 要加入菜单项, 可以调用菜单或菜单项的 `MenuItems.Add` 方法。如果调用的是主菜单的 `MenuItems.Add` 方法, 则菜单项将被加入到主菜单中; 如果调用的是菜单项的 `MenuItems.Add` 方法, 则得到一个子菜单, 其中包含加入的菜单项。

第 42 行调用了值为 `File` 的菜单项 `menuItemFile` 的 `MenuItems.Add` 方法, 因此子菜单项(这里为 `Update Date`)将被加入到 `File` 中。第 44 行加入了子菜单项 `Exit`。

仔细查看该程序清单, 您将发现第 41 对 `MenuItems.Add` 的调用不同于第 42 和 44 行。显然, 该方法被重载了。如果只传递一个字符串参数, 则该参数被认为是要显示的文本选项。在第 42 和 44 行的调用中, 传递了两个参数, 其中第一个是新建的 `MenuItem`, 它将被直接赋给菜单, 而不是中间变量。

第二个参数是一个事件处理程序, 它在该菜单项被选择时被调用。第 43 和 45 行传递的事件处理程序分别是在第 49 和 55 行创建的。

第 49 行是菜单项 `Update Date` 的事件处理程序, 其名称与第 42 行传递给 `MenuItems.Add` 方法的名称相同。别忘了, 建立事件处理程序时, 将需要两个参数: 发送者和 `EventArgs`。当菜单项 `Update Date` 被选择时, 第 49 ~ 54 行的事件处理程序将被调用, 它更新窗体上显示的日期和时间。

当 `Exit` 被选择时, 第 55 ~ 58 行的事件处理程序被调用, 它通过关闭窗体来退出应用程序。

17.3.2 创建多个菜单

本节将在主菜单中加入另一个菜单项, 并设置菜单项的快捷键。程序清单 17.5 是一个主菜单(见图 17.5)中包含菜单项 `File` 和 `Help` 的程序, 这些菜单项都有子菜单项。

程序清单 17.5 menu2.cs: 包含多个菜单项的主菜单

```
1: // menu2.cs - menus
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmApp : Form
9: {
10:     private Label myDateLabel;
11:     private MainMenu myMainMenu;
12:
13:     public frmApp()
14:     {
15:         InitializeComponent();
```

```
16:     }
17:
18:     private void InitializeComponent()
19:     {
20:         this.Text = "STY Menus";
21:         this.StartPosition = FormStartPosition.CenterScreen;
22:         this.FormBorderStyle = FormBorderStyle.Fixed3D;
23:
24:         myDateLabel = new Label();    // Create label
25:
26:         DateTime currDate = new DateTime();
27:         currDate = DateTime.Now;
28:         myDateLabel.Text = currDate.ToString();
29:
30:         myDateLabel.AutoSize = true;
31:         myDateLabel.Location = new Point( 50, 70);
32:         myDateLabel.BackColor = this.BackColor;
33:
34:         this.Controls.Add(myDateLabel); // Add label to form
35:
36:         // Set width of form based on Label's width
37:         this.Width = (myDateLabel.PreferredWidth + 100);
38:
39:         CreateMyMenu();
40:     }
41:
42:     protected void MenuUpdate_Selection( object sender, System.EventArgs e)
43:     {
44:         DateTime currDate = new DateTime();
45:         currDate = DateTime.Now;
46:         this.myDateLabel.Text = currDate.ToString();
47:     }
48:
49:     protected void FileExit_Selection( object sender, System.EventArgs e)
50:     {
51:         this.Close();
52:     }
53:
54:     protected void FileAbout_Selection( object sender, System.EventArgs e)
55:     {
56:         // display an about form
57:     }
58:
59:     public void CreateMyMenu()
60:     {
61:         myMainMenu = new MainMenu();
62:
63:         MenuItem menuItemFile = myMainMenu.MenuItems.Add("&File");
64:         menuItemFile.MenuItems.Add(new MenuItem("Update &Date",
65:         new
```

```

EventHandler(this.MenuUpdate_Selection),
66:         Shortcut.CtrlD));
67:     menuItemFile.MenuItems.Add(new MenuItem("E&xit",
68:         new EventHandler(this.FileExit_Selection),
69:         Shortcut.CtrlX));
70:
71:     MenuItem menuItemHelp = myMainMenu.MenuItems.Add("&Help");
72:     menuItemHelp.MenuItems.Add(new MenuItem("&About",
73:         new
EventHandler(this.FileAbout_Selection)));
74:
75:     this.Menu = myMainMenu;
76: }
77:
78: public static void Main( string[] args )
79: {
80:     Application.Run( new frmApp() );
81: }
82: }

```

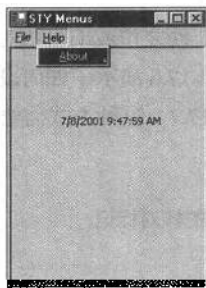


图 17.5 包含多个菜单项的主菜单

分析：该程序清单与 17.4 类似，主要区别之一是将创建菜单的代码放在一个单独的方法中，以便更好地组织代码。该方法名为 `CreateMenu`，位于第 59~76 行。`InitializeComponent` 方法在初始化窗体时调用 `CreateMenu` 方法，如第 39 行所示。

第 42~57 行包含了选择各个菜单项时将激活的事件对应的事件处理程序，前两个（`MenuUpdate_Selection` 和 `FileExit_Selection`）与程序清单 17.4 中的类似，第三个（`FileAbout_Selection`）将与 `Help` 菜单中的 `About` 选项关联起来。第 56 行没有包含任何代码，但可以将任何代码放在这里。在 `About` 菜单项被选择时，通常是显示一个描述性对话框。本章后面将介绍这样的对话框。

该程序清单的重点是第 59~76 行的菜单，这些菜单的创建方式与前一个程序清单相同。第 61 行实例化主菜单。第 63 行将菜单项 `File` 加入到 `myMainMenu` 中，该方法调用中有一个不同的地方，即在 `File` 的前面包含了一个 `&`，这指定 `&` 后面的字符将带下划线。对于 `MainMenu` 选项，这还指定使用 `Alt` 和 `&` 后面的字符，可以选择该菜单项。就这里而言，按下 `Alt+F` 将选择菜单选项 `File`。

第 64 和 67 行将菜单选项加入到 `File` 菜单中，这与前一个程序清单类似，但调用的是 `MenuItems.Add` 方法的另一个版本。这里包含第三个参数，它指定在菜单项中还应该显示其快捷键。查看 `File` 菜单的两个子菜单选项可以知道，它们后面都包含额外的文本，这表明它们都有快捷键，

如图 17.6 所示。

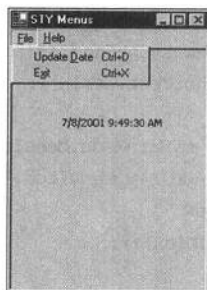


图 17.6 菜单项中的快捷键指示器

注意：幸运的是，数据类型 `Shortcut` 已经定义了快捷键。通常可以使用 `Shortcut.Ctrl*` 来表示快捷键，其中 `*` 可以是任何字母。

在主菜单中加入第二个菜单项的方式与第一个相同。第 71 行加入了菜单项 `Help`。由于该菜单项是一个子菜单项，因此需要将其赋给一个 `MenuItem` 变量，这里是 `menuItemHelp`，然后调用 `menuItemHelp` 的 `MenuItems.Add` 方法加入其菜单项。

17.3.3 设置菜单的核对标记

使用菜单时，另一种常见的特性是，使之带或不带核对标记。带核对标记说明该菜单项在起作用，否则，说明它没有起作用。程序清单 17.6 演示了如何设置菜单项的核对标记，并提供了另一种声明和定义菜单的方式，运行结果见图 17.7。这种方式需要更多的代码，但有些人认为它更容易阅读和理解。

程序清单 17.6 menu3.cs: 设置菜单的核对标记

```
1: // menu3.cs - menus
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmApp : Form
9: {
10:     private Label myDateLabel;
11:     private MainMenu myMainMenu;
12:
13:     private MenuItem menuItemFile;
14:     private MenuItem menuItemUD;
15:     private MenuItem menuItemActive;
16:     private MenuItem menuItemExit;
17:     private MenuItem menuItemHelp;
18:     private MenuItem menuItemAbout;
19:
20:     public frmApp()
21:     {
```



```
22:     InitializeComponent();
23: }
24:
25: private void InitializeComponent()
26: {
27:     this.Text = "STY Menu";
28:     this.StartPosition = FormStartPosition.CenterScreen;
29:     this.FormBorderStyle = FormBorderStyle.Sizable;
30:
31:     myDateLabel = new Label();    // Create label
32:
33:     DateTime currDate = new DateTime();
34:     currDate = DateTime.Now;
35:     myDateLabel.Text = currDate.ToString();
36:
37:     myDateLabel.AutoSize = true;
38:     myDateLabel.Location = new Point( 50, 70);
39:     myDateLabel.BackColor = this.BackColor;
40:
41:     this.Controls.Add(myDateLabel); // Add label to form
42:
43:     // Set width of form based on Label's width
44:     this.Width = (myDateLabel.PreferredWidth + 100);
45:
46:     CreateMyMenu();
47: }
48:
49: protected void MenuUpdate_Selection( object sender, System.EventArgs e)
50: {
51:     if( menuitemActive.Checked == true)
52:     {
53:         DateTime currDate = new DateTime();
54:         currDate = DateTime.Now;
55:         this.myDateLabel.Text = currDate.ToString();
56:     }
57:     else
58:     {
59:         this.myDateLabel.Text = "*** " + this.myDateLabel.Text + " ***";
60:     }
61: }
62:
63: protected void FileExit_Selection( object sender, System.EventArgs e)
64: {
65:     Application.Exit();
66: }
67:
68: protected void FileAbout_Selection( object sender, System.EventArgs e)
69: {
70:     // display an about form
71: }
```

```
72:
73:     protected void ActiveMenu_Selection( object sender, System.EventArgs e)
74:     {
75:         MenuItem tmp;
76:         tmp = (MenuItem) sender;
77:
78:         if ( tmp.Checked == true )
79:             tmp.Checked = false;
80:         else
81:             tmp.Checked = true;
82:     }
83:
84:     public void CreateMyMenu()
85:     {
86:         myMainMenu = new MainMenu();
87:
88:         // FILE MENU
89:         menuItemFile = myMainMenu.MenuItems.Add("&File");
90:
91:         menuItemUD = new MenuItem();
92:         menuItemUD.Text = "Update &Date";
93:         menuItemUD.Shortcut = Shortcut.CtrlD;
94:         menuItemUD.Click += new EventHandler(this.MenuUpdate_Selection);
95:         menuItemFile.MenuItems.Add( menuItemUD );
96:
97:         menuItemExit = new MenuItem();
98:         menuItemExit.Text = "E&xit";
99:         menuItemExit.Shortcut = Shortcut.CtrlX;
100:        menuItemExit.ShowShortcut = false;
101:        menuItemExit.Click += new EventHandler(this.FileExit_Selection);
102:        menuItemFile.MenuItems.Add( menuItemExit );
103:
104:        // HELP MENU
105:        menuItemHelp = myMainMenu.MenuItems.Add("&Help");
106:
107:        menuItemActive = new MenuItem();
108:        menuItemActive.Text = "Active";
109:        menuItemActive.Click += new EventHandler(this.ActiveMenu_Selection);
110:        menuItemActive.Checked = true;
111:        menuItemHelp.MenuItems.Add( menuItemActive );
112:
113:        menuItemAbout = new MenuItem();
114:        menuItemAbout.Text = "&About";
115:        menuItemAbout.Shortcut = Shortcut.CtrlA;
116:        menuItemAbout.ShowShortcut = false;
117:        menuItemAbout.Click += new EventHandler(this.FileAbout_Selection);
118:        menuItemHelp.MenuItems.Add( menuItemAbout );
119:
120:        this.Menu = myMainMenu;
121:    }
```

```

122:
123:     public static void Main( string[] args )
124:     {
125:         Application.Run( new frmApp() );
126:     }
127: }

```

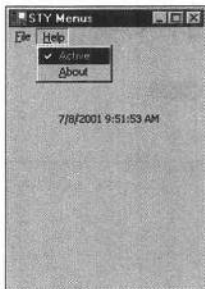


图 17.7 设置菜单的核对标记

分析：该程序清单比前两个更长，这是由于采用了另一种创建菜单的方式。

该程序清单的第 11 行声明了一个 MainMenu，然后为每一个菜单项声明一个 MenuItem 变量，而不是只声明一个 Mainmenu 和各个顶级菜单项。第 13 ~ 18 行为每个菜单项都声明了一个 MenuItem 变量，用于存储这些菜单项。

和前一个程序清单一样，创建菜单的代码被放在一个单独的方法中，该方法是从第 84 行开始的。第 86 行实例化 MainMenu 对象 myMainMenu。

第 89 行实例化第一个菜单项 File，方式与以前介绍的相同。而实例化子菜单项的方式则不同。第 91 ~ 95 行创建了菜单项 Update Date。首先将菜单项变量 menuItemUD 实例化为一个菜单项，然后设置该菜单项的各个属性值（第 92 和 93 行）。第 94 行将一个事件处理程序与该菜单项的 Click 事件关联起来，每当该菜单项被选择时，该事件处理程序都将被调用。最后，第 95 行将菜单项加入到父菜单项中，这里是将 menuItemUD 加入到 File 菜单 menuItemFile 中。

其他菜单项的加入方式与此相同，只是有时候设置的属性不同而已。

该程序清单还使用了一种以前没有介绍过的新特性。Help 菜单中的选项 Active 可以带或不带核对标记。如果带核对标记，则 Data Update 菜单项被选择时，将更新日期和时间；如果不带核对标记，则显示的日期和时间将被位于星号之间。这是通过设置菜单项的 Checked 属性（如第 110 行所示），并在菜单项的事件处理程序 ActiveMenu_Selection 中添加一些代码来实现的。

事件处理程序 ActiveMenu_Selection 位于第 73 ~ 83 行，这里并没有什么新东西，但有必要对其进行解释。和其他事件处理程序一样，该方法的第一个参数为一个对象。由于这种事件是由选择菜单引发的，因此您知道该对象实际上是一个 MenuItem。第 75 和 76 行创建了一个临时的 MenuItem 变量，并将第一个参数强制转换为 MenuItem，然后赋给该变量。这样，便可以使用临时变量 tmp 来访问 MenuItem 的所有属性和方法。

您还知道，该事件是由选择菜单项 Active 引发的。因此，第 78 行检查菜单项 Active 的 Checked 属性，看它是否为 true，即是否带核对标记。如果是，则第 79 行将 Checked 属性设置为 false，以取消核对标记；如果不是，则将 Checked 属性设置为 true。因此，最终结果在 Active 菜单是否带核对标记之间切换。

该程序清单也对事件处理程序 `MenuUpdate` 做了修改，如果菜单项 `Active` 带核对标记，则该事件处理程序显示当前日期和时间；否则，重新显示当前日期和时间，并将其放在星号之间。该事件处理程序的关键并不在于其显示的内容，而在于带核对标记的菜单选项可用于确定应该执行什么样的操作。

17.3.4 创建弹出式菜单

除了前面介绍的标准菜单外，您还可以创建弹出式菜单。程序清单 17.7 包含一个弹出式菜单，该菜单在用户单击鼠标右键时出现，如图 17.8 所示。

程序清单 17.7 `popup.cs`：使用弹出式菜单

```

1: // popup.cs - popup menus
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmApp : Form
9: {
10:     private ContextMenu myPopUp;
11:
12:     public frmApp()
13:     {
14:         InitializeComponent();
15:     }
16:
17:     private void InitializeComponent()
18:     {
19:         this.Text = "STY Pop-up Menu";
20:         this.StartPosition = FormStartPosition.CenterScreen;
21:
22:         CreatePopUp();
23:     }
24:
25:     protected void PopUp_Selection( object sender, System.EventArgs e)
26:     {
27:         // Determine menu item and do logic...
28:         this.Text = ((MenuItem) sender).Text;
29:     }
30:
31:     private void CreatePopUp()
32:     {
33:         myPopUp = new ContextMenu();
34:
35:         myPopUp.MenuItems.Add("First Item",
36:                               new EventHandler(this.PopUp_Selection));
37:
38:         myPopUp.MenuItems.Add("Second Item",

```

```

39:             new EventHandler(this.PopUp_Selection));
40:
41:         myPopUp.MenuItems.Add("-");
42:
43:         myPopUp.MenuItems.Add("Third Item",
44:             new EventHandler(this.PopUp_Selection));
45:
46:         this.ContextMenu = myPopUp;
47:
48:     }
49:
50:     public static void Main( string[] args )
51:     {
52:         Application.Run( new frmApp() );
53:     }
54: }

```

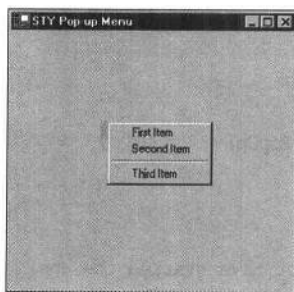


图 17.8 一个自定义的弹出式菜单

分析：所谓创建弹出式菜单，不是创建一个MainMenu对象，而是创建一个ContextMenu对象。这是在第33行完成的，这里将第10行声明的变量myPopUp实例化为一个ContextMenu对象。

给 ContextMenu 添加菜单项的方式与 MainMenu 相同——使用 MenuItems.Add 方法。这里，加入了四个菜单项。第 1、2、4 个菜单项的添加方式与前面介绍的相同，但第 41 行添加的第 3 个菜单项很特别。从第 41 行可知，加入到菜单中的好像只是一个短划线，但实际上将创建一条跨越菜单的线段，从图 17.8 的结果中可以知道这一点。加入 ContextMenu 对象 myPopUp 的所有选项后，该菜单被作为一个 ContextMenu 加入到当前窗体中。

这三个菜单项使用的是同一个事件处理程序——PopUp_Selection。该事件处理程序是在第 25 ~ 29 行定义的，其真正的功能是在第 28 行实现的。第 28 行将一个新的值赋给当前窗体的 Text 属性。记住，Text 属性是窗体的标题。所赋的值是调用该事件处理程序的对象的 Text 属性，即被选择的菜单项中的文本。第 28 行将发送对象强制转换为 MenuItem，然后使用该 MenuItem 的文本值。这是一种实现前一个程序清单中第 75 和 76 行的功能的简捷方式。

17.4 显示弹出式对话框和窗体

您已经知道如何在窗体中显示控件和菜单，还知道如何创建事件处理程序，以便对窗体上发生

的事件做出反应。到目前为止的范例中，还未介绍如何显示另一个对话框或窗体。

接下来的几节将介绍三个与显示新窗体或对话框相关的主题。首先，将使用 `MessageBox` 类的基本功能；然后讨论 Microsoft Windows 中的一些对话框；最后，将介绍如何创建自己的对话框窗体，这与使用自定义对话框有天壤之别。

17.4.1 MessageBox 类

编写 Windows 应用程序时，经常使用的一个类是消息框。基类库 (BCL) 中也定义了一个对话框类，它让您能够在弹出式对话框中显示消息。程序清单 17.8 使用 `MessageBox` 类来显示弹出式消息，结果如图 17.9 ~ 17.12 所示。

程序清单 17.8 msgbox.cs: 使用 `MessageBox` 类

```
1: // msgbox.cs - Using the MessageBox class
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmApp : Form
9: {
10:     private ContextMenu myPopUp;
11:
12:     public frmApp()
13:     {
14:         MessageBox.Show( "You have started the application.", "Status");
15:         InitializeComponent();
16:         CreatePopUp();
17:         MessageBox.Show( "Form has been initialized.", "Status");
18:     }
19:
20:     private void InitializeComponent()
21:     {
22:         this.Text = "STY C# Pop-up Menu";
23:         this.StartPosition = FormStartPosition.CenterScreen;
24:     }
25:
26:     protected void PopUp_Selection( object sender, System.EventArgs e)
27:     {
28:         // Determine menu item and do logic...
29:         MessageBox.Show( ((MenuItem) sender).Text, this.Text + " Msg Box");
30:     }
31:
32:     private void CreatePopUp()
33:     {
34:         myPopUp = new ContextMenu();
35:
36:         myPopUp.MenuItems.Add("First Item",
37:                               new EventHandler(this.PopUp_Selection));
```

```
38:     myPopUp.MenuItems.Add("Second Item",
39:         new EventHandler(this.PopUp_Selection));
40:     myPopUp.MenuItems.Add("-");
41:     myPopUp.MenuItems.Add("Third Item",
42:         new EventHandler(this.PopUp_Selection));
43:
44:     this.ContextMenu = myPopUp;
45: }
46:
47: public static void Main( string[] args )
48: {
49:     Application.Run( new frmApp() );
50:     MessageBox.Show( "You are done with the application", "Status");
51: }
52: }
53: }
```



图 17.9 程序运行时显示的消息框



图 17.10 初始化窗体后显示的消息框

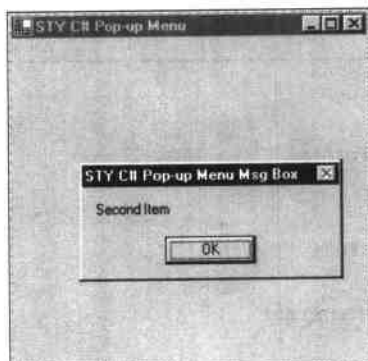


图 17.11 选择弹出式菜单项后显示的消息框



图 17.12 程序结束时显示的消息框

分析：该程序清单在程序运行的不同阶段使用MessageBox对象来显示消息。

MessageBox 的基本用途让您能够在包含一个 OK 按钮的对话框中显示文本字符串。您也可以指

定对话框的标题。程序清单 17.8 使用了多个消息框。第一个是在构造函数内调用的（第 14 行），这里在对 `MessageBox` 的 `Show` 方法的基本调用中，传递了两个参数。第一个参数是将显示在对话框中的消息，第二个参数是对话框的标题。对比第 14 行的代码和图 17.9 中的对话框可以知道，确实是这样的。第 14 行立刻显示对话框。

第 15 和 16 行初始化应用程序的主窗体，并建立弹出式菜单。然后，显示另一个消息框，指出初始化已经完成。这是显示的第二个消息框。

用户单击第二个消息框中的 OK 按钮后，构造函数（`frmApp()`）的代码便执行完毕，将出现应用程序的主窗体。该窗体是空的，它包含一个单击鼠标右键时将弹出的菜单。当用户选择弹出式菜单中的选项后，第 26 ~ 30 行的事件处理程序 `PopUp_Selection` 将被调用。无论被选择的是哪个菜单项，该事件处理程序都将显示一个消息框。

图 17.12 是最后一个消息框，它在用户退出程序时出现。该对话框是由第 50 行调用 `MessageBox.Show` 显示的，因此它的主窗体被关闭后才出现。

17.4.2 Microsoft Windows 中已有的对话框

除了 `MessageBox` 类外，还定义了大量更为复杂的对话框。下面是一些非常有用的对话框：

- 颜色选择对话框（`ColorDialog`）；
- 打印预览对话框（`PrintPreviewDialog`）；
- 字体对话框（`FontDialog`）；
- 打开文件对话框（`OpenFileDialog`）；
- 保存文件对话框（`SaveFileDialog`）。

这些对话框位于 BCL 中。程序清单 17.9 表明，在程序中加入这些对话框的基本特性非常容易，该程序清单的运行结果如图 17.13 ~ 17.15 所示。

程序清单 17.9 `canned.cs`：使用一些对话框

```
1: // canned.cs - using existing dialogs
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmApp : Form
9: {
10:     private MainMenu myMainMenu;
11:
12:     public frmApp()
13:     {
14:         InitializeComponent();
15:     }
16:
17:     private void InitializeComponent()
18:     {
19:         this.Text = "Canned Dialogs";
20:         this.StartPosition = FormStartPosition.CenterScreen;
21:         this.FormBorderStyle = FormBorderStyle.Sizable;
```



```
22:     this.Width = 400;
23:
24:     myMainMenu = new MainMenu();
25:
26:     MenuItem menuItemFile = myMainMenu.MenuItems.Add("&File");
27:     menuItemFile.MenuItems.Add(new MenuItem("Colors Dialog",
28:         new EventHandler(this.Menu_Selection)));
29:     menuItemFile.MenuItems.Add(new MenuItem("Fonts Dialog",
30:         new EventHandler(this.Menu_Selection)));
31:     menuItemFile.MenuItems.Add(new MenuItem("Print Preview Dialog",
32:         new EventHandler(this.Menu_Selection)));
33:     menuItemFile.MenuItems.Add("-");
34:     menuItemFile.MenuItems.Add(new MenuItem("Exit",
35:         new EventHandler(this.Menu_Selection)));
36:     this.Menu = myMainMenu;
37: }
38:
39: protected void Menu_Selection( object sender, System.EventArgs e )
40: {
41:     switch ((MenuItem) sender).Text )
42:     {
43:         case "Exit":
44:             Application.Exit();
45:             break;
46:
47:         case "Colors Dialog":
48:             ColorDialog myColorDialog = new ColorDialog();
49:             myColorDialog.ShowDialog();
50:             break;
51:
52:         case "Fonts Dialog":
53:             FontDialog myFontDialog = new FontDialog();
54:             myFontDialog.ShowDialog();
55:             break;
56:
57:         case "Print Preview Dialog":
58:             PrintPreviewDialog myPrintDialog =
59:                 new PrintPreviewDialog();
60:             myPrintDialog.ShowDialog();
61:             break;
62:
63:         default:
64:             MessageBox.Show("DEFAULT", "PopUp");
65:             break;
66:     }
67: }
68:
69: public static void Main( string[] args )
70: {
71:     Application.Run( new frmApp() );
}
```

```
72:     }  
73: }
```

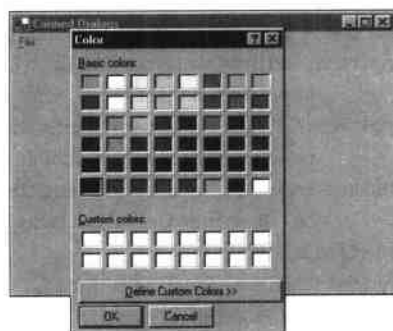


图 17.13 显示基本颜色对话框

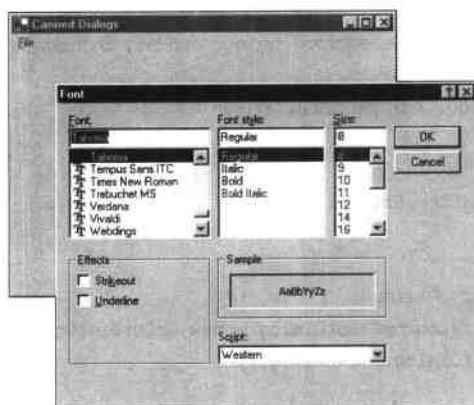


图 17.14 显示基本字体对话框

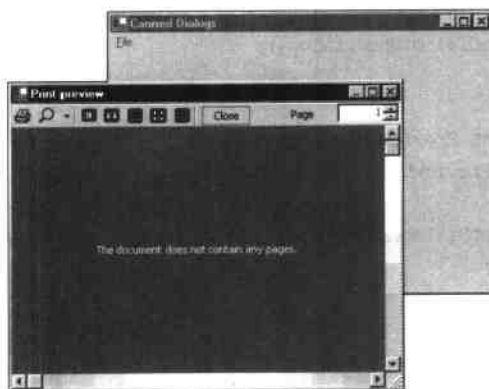


图 17.15 显示打印预览对话框

分析：该程序清单使用前面的程序清单中介绍的功能来显示对话框。在窗体中加入了一个File菜单，它包含用于显示三种对话框的菜单项。其中每个对话框的显示方式都相同——实例化一个对

对话框对象，然后调用该对象的 ShowDialog 方法来显示对话框。选择不同的菜单项将显示相应的对话框。

注意：更多关于如何使用和定制这些对话框特性的信息，请参阅在线帮助文档。

17.4.3 弹出自己的对话框

您也可以创建自己的对话框，这与创建基本窗体的方式相同——定义一个窗体对象，加入控件，然后显示它。

显示窗体的方式有两种。可以显示一个窗体，用于必须对其进行响应，应用程序才能继续运行。这是使用 ShowDialog 方法实现的。另一种方式是，显示一个窗体，并允许应用程序继续运行或允许继续显示其他窗体。这是通过 Show 方法实现的。程序清单 17.10 说明了 Show 和 ShowDialog 之间的差别，如图 17.16 所示。

程序清单 17.10 myform.cs: 使用 Show 和 ShowDialog

```
1: // myform.cs - displaying subforms
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class frmApp : Form
9: {
10:     private MainMenu myMainMenu;
11:
12:     public frmApp()
13:     {
14:         InitializeComponent();
15:     }
16:
17:     private void InitializeComponent()
18:     {
19:         this.Text = "Canned Dialogs";
20:         this.StartPosition = FormStartPosition.CenterScreen;
21:         this.FormBorderStyle = FormBorderStyle.Sizable;
22:         this.Width = 400;
23:
24:         myMainMenu = new MainMenu();
25:
26:         MenuItem menuItemFile = myMainMenu.MenuItems.Add("&File");
27:         menuItemFile.MenuItems.Add(new MenuItem("My Form",
28:             new EventHandler(this.Menu_Selection)));
29:         menuItemFile.MenuItems.Add(new MenuItem("My Other Form",
30:             new EventHandler(this.Menu_Selection)));
31:         menuItemFile.MenuItems.Add("-");
32:         menuItemFile.MenuItems.Add(new MenuItem("Exit",
33:             new EventHandler(this.Menu_Selection)));
34:         this.Menu = myMainMenu;
```

```
35:     |
36:
37:     protected void Menu_Selection( object sender, System.EventArgs e )
38:     {
39:         switch (((MenuItem) sender).Text )
40:         {
41:             case "Exit":
42:                 Application.Exit();
43:                 break;
44:
45:             case "My Form":
46:                 subForm aForm = new subForm();
47:                 aForm.Text = "A Show form";
48:                 aForm.Show();
49:                 break;
50:
51:             case "My Other Form":
52:                 subForm bForm = new subForm();
53:                 bForm.Text = "A ShowDialog form";
54:                 bForm.ShowDialog();
55:                 break;
56:
57:             default:
58:                 MessageBox.Show("DEFAULT", "PopUp");
59:                 break;
60:         }
61:     }
62:
63:     public static void Main( string[] args )
64:     {
65:         Application.Run( new frmApp() );
66:     }
67: }
68:
69:
70: public class subForm : Form
71: {
72:     private MainMenu mySubMainMenu;
73:
74:     public subForm()
75:     {
76:         InitializeComponent();
77:     }
78:
79:     private void InitializeComponent()
80:     {
81:         this.Text = "My sub-form";
82:         this.StartPosition = FormStartPosition.CenterScreen;
83:         this.FormBorderStyle = FormBorderStyle.FixedDialog;
84:         this.Width = 300;
```

```

85:     this.Height = 250;
86:
87:     mySubMainMenu = new MainMenu();
88:
89:     MenuItem menuItemFile = mySubMainMenu.MenuItems.Add("&File");
90:     menuItemFile.MenuItems.Add(new MenuItem("Close",
91:         new EventHandler(this.CloseMenu_Selection)));
92:     this.Menu = mySubMainMenu;
93: }
94:
95: protected void CloseMenu_Selection( object sender, System.EventArgs e )
96: {
97:     this.Close();
98: }
99: }

```

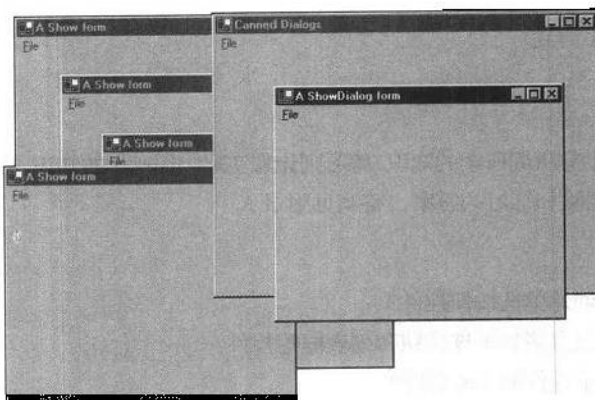


图 17.16 使用 ShowDialog 时, 只能同时显示一个窗体; 使用 Show 时, 可以同时显示多个窗体

分析: 运行该程序时, 您将发现, 使用 Show 方法, 可以同时显示多个窗体, 并可以在它们和应用程序主窗体之间改变焦点; 而使用 ShowDialog 方法显示窗体时, 在关闭该窗体之前, 您不能在程序中执行其他任何操作。

注意: 必须对其进行响应, 才会放弃焦点的窗体被称为是模态的 (Modal)。

17.5 总 结

今天的课程是本书中最长的, 但其中包含大量的代码。今天接着昨天进一步介绍了关于创建 Windows 应用程序的知识。虽然今天介绍的内容不属于 C# 语言的 ECMA 标准的一部分, 但适用于 Microsoft Windows 编程。支持 .NET 的其他平台很可能也包含类似 (如果不是相同的话) 的功能。

今天首先介绍了两种控件: 单选按钮和列表框。您学习了如何使用分组框将控件分组以及如何给应用程序添加主菜单和弹出式菜单。另外, 今天还介绍了如何创建包含多个窗体和对话框的应用程序。您不但学习了如何使用消息框, 还学习了如何创建自己的对话框或使用已有的对话框。

这两天关于 Windows 编程的内容并不是要做全面的介绍, 但确实为您开发 Windows 应用程序打下了基础。

17.6 问与答

问: 使用诸如 Visual Studio 或 Microsoft Visual C# 等工具时, 今天和昨天介绍的知识有多重要?

答: 图形 IDE 将为您完成大量的编码工作。例如, 使用 Visual Studio 时, 可以将控件拖放到窗体上, 并使用该工具中的对话框来设置属性。这使得创建对话框非常容易。昨天和今天介绍的知识有助于您理解这些工具为您生成的代码, 这样您便能更深入地理解程序及其是如何运行的。

问: 关于 Windows 编程, 还有大量的知识需要学习吗?

答: 昨天和今天的课程只是介绍了基于 Windows 编程中很少一部分知识, 但确实为您编写这种程序打下了坚实的基础。

问: 昨天和今天介绍的知识可以移植到其他平台中吗?

答: 由于介绍的所有方法都是基类库中的, 因此可以移植到新平台中。实际上, 这些类并不都是 C# 的 ECMA 标准的一部分, 因此不能保证这些窗口类和控件类都可以移植到其他平台, 但能够移植的可能性较大。

17.7 作业

下面的小测验帮助您巩固所学的知识, 练习则让您实际应用所学的知识。在阅读下一课时之前, 应尽可能理解这些小测验和练习的答案, 答案见附录 A。

17.7.1 小测验

1. 哪个类被用于创建单选按钮控件?
2. 哪个名称空间包含诸如单选按钮和列表框等控件?
3. 如何设置窗体中控件的 Tab 顺序?
4. 在列表框中加入选项包括哪几步?
5. MainMenu 和 ContextMenu 对象之间的区别何在?
6. 在简单的对话框中显示简单消息的简易方式是什么?
7. 基类库中包含哪些可用的对话框?
8. 如果要显示一个窗体, 并不允许应用程序中的其他窗体被显示或激活, 应该使用哪种方法?
9. 使用 Show 方法时, 可以同时显示多少窗体?

17.7.2 练习

1. 编写这样的代码: 将名为 btn1 和 btn2 的单选按钮控件加入到一个名为 grbox 的组合框中。
2. 要在菜单 MYMENU 中加入一条线段, 需要使用什么代码?
3. 创建一个使用 ColorDialog 对话框的应用程序。将应用程序主窗体的背景色设置为 ColorDialog 返回的颜色。返回的颜色被存储在 Color 属性中。提示: 创建一个 ColorDialog 变量, 调用该对话框时, 选择的颜色应该存储在 Color 属性中。

4. 下面的代码有问题吗? 如果有, 是什么问题?

```
1: using System;
2: using System.Windows.Forms;
3: using System.Drawing;
4:
```

```
5: public class frmApp : Form
6: {
7:     private Label myDateLabel;
8:     private MainMenu myMainMenu;
9:
10:    public frmApp()
11:    {
12:        this.Text = "STY Menu";
13:        this.FormBorderStyle = FormBorderStyle.Fixed3D;
14:
15:        myDateLabel = new Label();    // Create label
16:
17:        DateTime currDate = new DateTime();
18:        currDate = DateTime.Now;
19:        myDateLabel.Text = currDate.ToString();
20:        myDateLabel.AutoSize = true;
21:        myDateLabel.Location = new Point( 50, 70);
22:        myDateLabel.BackColor = this.BackColor;
23:        this.Controls.Add(myDateLabel); // Add label to form
24:        this.Width = (myDateLabel.PreferredWidth + 100);
25:
26:        CreateMyMenu();
27:    }
28:
29:    protected void MenuUpdate_Selection( object sender,
System.EventArgs e)
30:    {
31:        DateTime currDate;
32:        currDate = DateTime.Now;
33:        this.myDateLabel.Text = currDate.ToString();
34:    }
35:
36:    protected void FileExit_Selection( object sender, System.EventArgs
e)
37:    {
38:        this.Close();
39:    }
40:
41:    public void CreateMyMenu()
42:    {
43:        myMainMenu = new MainMenu();
44:
45:        MenuItem menuItemFile = myMainMenu.MenuItems.Add("&File");
46:        menuItemFile.MenuItems.Add(new MenuItem("Update &Date",
47:            new
EventHandler(this.MenuUpdate_Selection),
48:            Shortcut.CtrlH));
49:        menuItemFile.MenuItems.Add(new MenuItem("E&xit",
50:            new
EventHandler(this.FileExit_Selection),
```

```
51:             Shortcut.CtrlX));  
52:         this.Menu = myMainMenu;  
53:     }  
54:  
55:     public static void Main( string[] args )  
56:     {  
57:         Application.Run( new frmApp() );  
58:     }  
59: }
```

5. 选做题: 创建一个包含菜单的应用程序。用户选择该菜单时, 将显示一个对话框, 对话框中包含大量的控件, 其中一个是 OK 按钮。
6. 修改程序清单 17.6, 使之包含一个 About 对话框。

第 18 天课程

Web 开发

前两天介绍了如何创建使用 Windows 窗体的应用程序。如果您要创建用于 Internet 的应用程序,更具体地说是 Web 应用程序,则可能无法使用名称空间 `System.Windows.Forms` 中基于 Windows 的表单,而可能通过使用诸如 HTML 和 XML 等通用标准来充分发挥 Web 与不同系统交互的能力。今天介绍以下内容:

- 有关 Web 服务的基本知识;
- 使用 C# 创建一个简单的 Web 服务;
- 如何生成代理文件,以使用 Web 服务;
- 在客户程序中使用 Web 服务;
- Web 表单概述;
- Web 表单中使用的一些基本控件;
- 服务器控件和客户控件之间的区别;
- 创建一个基本的 Web 表单应用程序。

警告: 今天介绍的主题——Web 开发本身足够成一本书,为避免今天的课程的篇幅过大,我将做一些假设。如果您不符合这些假设,也不用着急——这里介绍的代码和概念将很有用处。

我所做的假设如下:

- 在介绍 Web 服务时,假设您能够访问 Web 服务器或 Web 服务提供者,它们可以托管您基于 .NET 运行环境编写的 Web 服务;
- 您熟悉基本的 Web 开发概念,包含使用 HTML 和基本的客户端脚本语言;
- 您使用的计算机安装了支持 ASP 和 ASP.NET 的 Web 服务器,如 IIS;
- 您的 Web 服务器包含标准的 `Inetpub/wwwroot` 目录,今天的课程将该目录作为 Web 服务器的根目录。如果您知道如何创建虚拟目录,则也可以使用虚拟目录。

18.1 创建 Web 应用程序

今天的课程将介绍两类 Web 应用程序: Web 服务和 Web 表单。这两类应用程序都有其各自的用途,我们首先介绍 Web 服务。

18.2 组件的概念

新术语：介绍 Web 服务的概念之前，有必要先介绍一下组件的概念。*组件*是一个软件，有定义良好的接口，内部细节被隐藏，并可以被发现。发现指的是不用查看内部的代码，便可以知道组件的功能。组件在很多方面与方法类似，它可以被调用（提供参数），并能够返回结果。

18.3 Web 服务

新术语：Web 中已经开始将方法作为组件。Web 组件可以被称为 Web 服务，*Web 服务*是一个实现功能或服务的组件，它也可以将信息返回给调用者。这种服务位于 Web 的某个地方，可以在 Web 的其他地方访问它。为使这种服务可以被调用，需要满足一些条件。首先，调用者必须知道如何调用服务；其次，调用必须跨越 Web 进行；最后，Web 服务必须知道如何做出响应。图 18.1 说明了调用 Web 服务的过程。



图 18.1 Web 服务

图中使用简单对象访问协议（Simple Object Access Protocol，或称 SOAP）在程序和 Web 服务之间进行通信。SOAP 是一种用于格式化有关方法调用和数据等信息的标准方式，这种格式化是基于 XML 标准的。使用 SOAP，程序和 Web 服务之间能够进行通信。

调用 Web 服务的程序可以是 C# 程序或使用其他任何编程语言编写的程序。另外，调用 Web 服务的程序还可以是浏览器或另一个 Web 服务（可以是使用 C# 或其他任何语言编写的）。由于使用的是 一种标准化协议（SOAP），因此调用 Web 服务的程序和 Web 服务之间能够进行交互。

注意：虽然对于创建 Web 服务而言，理解 SOAP 很有用，但不是必须的。

建立并使用 Web 服务包括三步：

1. 创建 Web 服务；
2. 创建使用 Web 服务的程序；
3. 创建一个帮助程序调用 Web 服务的文件，这种助手程序被称为 Web 代理。

接下来的几节将介绍如何创建上述各个部分。

18.3.1 创建简单的组件

创建 Web 服务之前，先来创建一个简单的类。然后，以这个类为基础，创建您的第一个 Web 服务。这个类将被编译为库中的一个例程。程序清单 18.1 列出了将被使用的简单程序。

程序清单 18.1 Calc.cs：一个基本组件

```
1: // Calc.cs
2: //-----
3:
4: using System;
```

```
5:
6: public class Calc
7: {
8:     public static int Add( int x, int y )
9:     {
10:         return x + y;
11:     }
12:     public static int Subtract( int x, int y )
13:     {
14:         return x - y;
15:     }
16: ;
```

分析: 欲使之成为一个可调用的外部类, 需要将该程序清单编译为一个库。为此, 需要将目标标记/t:中指定library, 如下所示:

```
csc /t:library Calc.cs
```

这样编译后的文件将为 Calc.dll, 而不是 Calc.exe。

从程序清单可知, 这个组件的 Calc 类中包含两个方法。第 8~11 行的 Add 方法将两个数相加, 第 12~15 行的 Subtract 方法返回两个数的差。程序清单 18.2 是一个可以使用这些方法的程序。

程序清单 18.2 main.cs: 使用方法 Add 和 Subtract

```
1: // main.cs
2: // Calling a component
3: //-----
4:
5: using System;
6:
7: public class myApp
8: {
9:     public static void Main()
10:    {
11:        Console.WriteLine("Using Calc component");
12:        Console.WriteLine("Calc.Add( 11, 33); = {0}",
13:                           Calc.Add(33, 11));
14:        Console.WriteLine("Calc.Subtract(33, 11); = {0}",
15:                           Calc.Subtract(33, 11));
16:
17:    }
18: }
```

该程序清单的输出如下:

```
Using Calc component
Calc.Add(11, 33); = 44
Calc.Subtract(33, 11); = 22
```

分析: 如果像通常那样编译该程序:

```
csc main.cs
```

将出现错误，指出某种类型或名称空间找不到。正如本书前面介绍的，您需要包含到要使用的组件的引用，这里是 Calc。请编译程序 main.cs，并包含到程序清单 18.1 创建的组件所在的文件的引用，方法是使用引用编译标记，如下所示：

```
csc /r:Calc.dll main.cs
```

/r 是引用标记，它命令编译器包含指定的文件——Calc.dll。这样，main.cs 便可以使用 Calc.dll 中的类和方法了。

来看一看程序清单 18.2 中的代码，其中没有任何不同于以前介绍的内容。Main 方法使用类 Calc.dll。由于您在编译命令中将 Calc.dll 包含了进来，因此 Calc 类及其方法 Add 和 Subtract 是可用的。

18.3.2 创建 Web 服务

Calc 类及其方法很不错，但程序清单 18.1 和 18.2 中的范例针对的是类位于本地计算机中的情况。Web 服务跨越 Web 使用组件。您希望 Calc 的方法可以像 Web 服务那样运行，以便任何基于 Web 的应用程序都能跨越 Web 来访问它们。这显然增加了使用类的复杂性。

要在 Calc 类的基础上创建 Web 服务，需要对其做一些修改。程序清单 18.3 显示了将 Calc 类作为 Web 服务的情况。

程序清单 18.3 WebCalc.asmx: 使用 Calc

```
1: <%@WebService Language="C#" Class="Calc"%>
2:
3: //-----
4: // WebCalc.asmx
5: //-----
6:
7: using System;
8: using System.Web.Services;
9:
10: public class Calc : WebService
11: {
12:     [WebMethod]
13:     public int Add( int x, int y )
14:     {
15:         return x + y;
16:     }
17:
18:     [WebMethod]
19:     public int Subtract( int x, int y )
20:     {
21:         return x - y;
22:     }
23: }
```

分析：该程序清单做了多处修改，以使之成为一个 Web 服务。从程序清单可知，这些修改都不是大刀阔斧的。

第一个不同的地方是文件名，Web 服务的文件扩展名总是为 .asmx，而不是 .cs。这种扩展名告诉运行环境和浏览器，这是一个 Web 服务。

代码中第一个不同的地方是第 1 行，其中包含许多奇怪的内容：

```
<%@WebService Language="C#" Class="Calc"%>
```

<%@和%>是针对 Web 服务器的指示器，通过这些指示器，Web 服务器将知道这是一个使用 C# 语言编写的 Web 服务，并知道其中的类名为 Calc。由于语言被指定为 C#，因此服务器阅读文件的其他内容时，将把它们看作是 C# 语言，而不是其他语言。

该服务将在首次调用时被编译，因此您无需对其进行编译。Web 服务器将根据命令 Language= 中指定的语言调用相应的编译器。

下一个不同的地方是第 8 行，它将名称空间 System.Web.Service 包含进来，这样程序清单的后面使用 WebMethod 和 WebService 时，便不用显式地指明名称空间。

第 10 行从 WebService 类派生出 Calc 类，这样 Calc 类将具备 .NET 框架中定义的 Web 服务的特征。

最后一个不同的地方是，指定可供每个访问服务的人使用的方法。这是通过在方法前加上 [WebMethod] 实现的，如第 12 和 18 行所示。

仅此而已。这样便创建了一个可以使用的 Web 服务了。

注意：如果使用的是 Visual Studio.NET，其中将包含创建 Web 服务工程的选项。这种工程提供了 Web 服务的基本结构。和其他 Visual Studio.NET 工程一样，其中还包含大量的代码。

警告：由于 Web 服务是跨越 Web 进行访问的，并可以从任何平台调用，因此应避免在 Web 服务中使用图形用户界面（GUI）。

接下来的几节将介绍如何创建代理以及调用 Web 服务。您可能急于想知道如何使用 Web 服务，下面马上介绍。

如果您使用的是诸如微软公司的 IIS 这样的 Web 服务器，则您的计算机中将有一个名为 Inetpub 的目录。该目录包含一个名为 wwwroot 的子目录，您可以将 Web 服务（WebCalc.asmx）复制到该目录下。

将 Web 服务复制到该目录下后，便可以使用浏览器调用该服务。要调用 WebCalc.asmx 服务，使用的地址如下：

```
http://localhost/WebCalc.asmx
```

这样将得到类似于图 18.2 所示的页面。如果 Web 服务有错，则得到的页面将不同，它将指出其中的错误。

该页面显示了大量关于 Web 服务的信息。最重要的是，它列出了您可以执行的操作：Add 和 Subtract，这是 Web 服务类中的两个方法。

单击其中的一个方法，将出现另一个页面（如图 18.3 所示），让您能够输入方法期望的参数。

Add 方法期望两个参数：x 和 y，这与程序清单 18.3 中的代码一致。如果输入 5 和 10，将得到如下所示的结果：

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://tempuri.org/">15</int>
```

这里的结果是 15，但为何还包含大量其他的内容。这些内容是将信息发回给调用程序所需的

SOAP 信息。

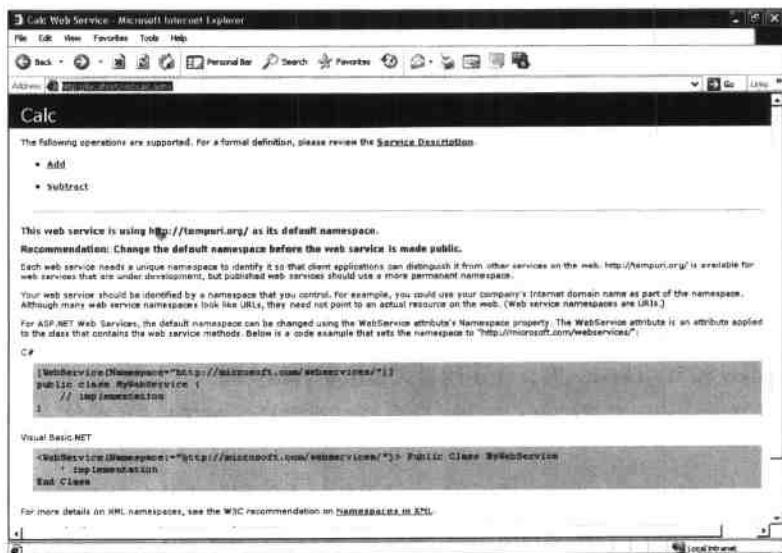


图 18.2 在 IE 中显示 Web 服务 WebCalc.asmx

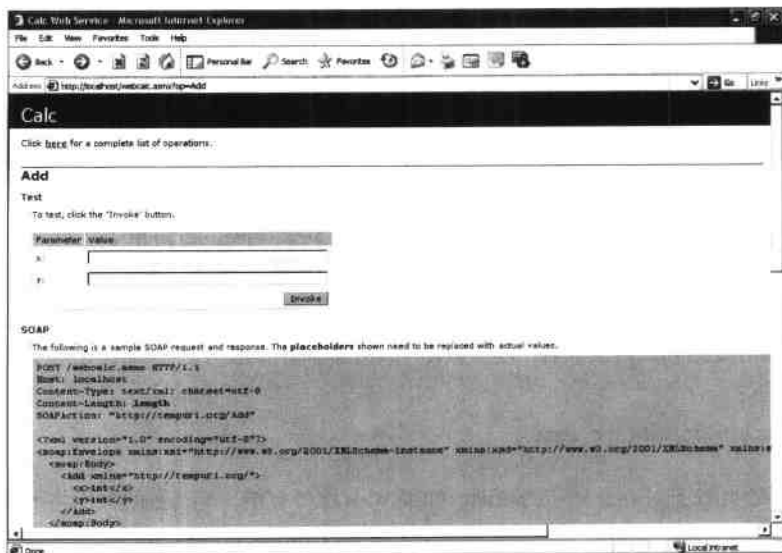


图 18.3 Web 服务中的 Add 方法

18.3.3 创建代理

前一节介绍了如何通过浏览器来使用本地机器上的 Web 服务，但更大的可能是，您想在另一个程序中使用该 Web 服务。为此，需要创建一个 Web 代理。

正如前面指出的，代理将帮助本地程序知道在 Web 的什么位置可以找到 Web 服务，同时它还包含用于同 Web 服务通信的详细信息（SOAP 信息）。

编写代码的工作量很大，但存在能够简化这种工作的工具。其中的一个是 `wsdl.exe`，微软公司在其 .NET 框架中提供了这种工具。它是一个命令行工具，可以使用下面的参数来运行：

```
wsdl webservice_file?wsdl /out:proxyfile
```

其中 `wsdl` 是您要执行的工具的名称, `webservice_file` 是 Web 服务文件的名称和位置。Web 服务名后面的 `?wsdl` 表示要生成一个使用 `wsdl` 标准的文件。对于 Web 服务 `WebCalc.asmx`, 当前其位置为 `localhost`, 如果该服务位于其他服务器上, 则位置将是访问该服务的 URL。

`/out:` 标记是可选的, 用于指定生成的代理的名称。如果省略该标记, 则代理的名称将与 Web 服务相同。我建议在代理名称中包含 `proxy`。下面的命令行为 `WebCalc.asmx` 创建一个代理文件, 将其命名为 `calcproxy.cs`, 并放置在目录 `inetpub\wwwroot` 中:

```
wsdl http://localhost/WebCalc.asmx?wsdl /out:c:\inetpub\wwwroot\calcproxy Cs
```

代理文件的扩展名为 `.cs`, 这意味着它是可编译的 C# 代码。程序清单 18.4 列出了使用 `wsdl` 根据前面创建的 `WebCalc.asmx` 文件生成的代码。

程序清单 18.4 `calcproxy.cs`: `wsdl` 生成的代码

```
//-----
// <autogenerated>
//   This code was generated by a tool.
//   Runtime Version: 1.0.2914.16
//
//   Changes to this file may cause incorrect behavior and will be lost
//   if the code is regenerated.
// </autogenerated>
//-----

//
// This source code was auto-generated by wsdl, Version=1.0.2914.16.
//
using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.Web.Services;

[System.Web.Services.WebServiceBindingAttribute(Name="CalcSoap",
    Namespace="http://tempuri.org/")]
public class Calc : System.Web.Services.Protocols.SoapHttpClientProtocol {

    [System.Diagnostics.DebuggerStepThroughAttribute()]
    public Calc() {
        this.Url = "http://localhost/WebCalc.asmx";
    }

    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.Web.Services.Protocols.SoapDocumentMethodAttribute(
        "http://tempuri.org/Add",
        Use=System.Web.Services.Description.SoapBindingUse.Literal,
        ParameterStyle=
            System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
```

```

public int Add(int x, int y) {
    object[] results = this.Invoke("Add", new object[] {
        x,
        y});
    return ((int)(results[0]));
}

[System.Diagnostics.DebuggerStepThroughAttribute()]
public System.IAsyncResult BeginAdd(int x, int y, System.AsyncCallback
    callback, object asyncState) {
    return this.BeginInvoke("Add", new object[] {
        x,
        y}, callback, asyncState);
}

[System.Diagnostics.DebuggerStepThroughAttribute()]
public int EndAdd(System.IAsyncResult asyncResult) {
    object[] results = this.EndInvoke(asyncResult);
    return ((int)(results[0]));
}

[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.Web.Services.Protocols.SoapDocumentMethodAttribute(
    "http://tempuri.org/Subtract",
    Use=System.Web.Services.Description.SoapBindingUse.Literal,
    ParameterStyle=
        System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
public int Subtract(int x, int y) {
    object[] results = this.Invoke("Subtract", new object[] {
        x,
        y});
    return ((int)(results[0]));
}

[System.Diagnostics.DebuggerStepThroughAttribute()]
public System.IAsyncResult BeginSubtract(int x, int y,
    System.AsyncCallback callback, object asyncState) {
    return this.BeginInvoke("Subtract", new object[] {
        x,
        y}, callback, asyncState);
}

[System.Diagnostics.DebuggerStepThroughAttribute()]
public int EndSubtract(System.IAsyncResult asyncResult) {
    object[] results = this.EndInvoke(asyncResult);
    return ((int)(results[0]));
}
}

```


解释该程序清单中的代码不在本书的范围之内。需要注意的重要的一点是，它负责为您处理 SOAP 信息。使用之前必须编译它。像前面所做的那样，需要将该程序清单编译为一个库。记住，这是使用目标标记 `/t:library` 实现的：

```
csc /t:library calcproxy.cs
```

编译得到的文件名为 `calcproxy.dll`，调用 Web 服务的程序将使用该文件。

18.3.4 调用 Web 服务

使用 Web 服务的最后一步是创建调用服务的程序。程序清单 18.5 是一个可使用 Web 服务 WebCalc 的简单程序。

程序清单 18.5 WebClient.cs: 使用 WebCalc 服务的客户程序

```
1: // WebClient.cs
2: // Calling a Web service
3: //-----
4:
5: using System;
6:
7: public class MyApp
8: {
9:     public static void Main()
10:    {
11:        Calc cSrv = new Calc();
12:
13:        Console.WriteLine("cSrv.Add( 11, 33); = {0}",
14:                           cSrv.Add(33, 11));
15:        Console.WriteLine("cSrv.Subtract(33, 11); = {0}",
16:                           cSrv.Subtract(33, 11));
17:    }
18: }
```

该程序清单的输出如下：

```
cSrv.Add(11, 33); = 44
cSrv.Subtract(33, 11); = 22
```

编译该程序清单时，需要包含对前面编译得到的代理文件的引用。方法与包含其他库相同，即使用 `/r` 标记：

```
csc /r:calcproxy.dll WebClient.cs
```

编译后，便得到了一个可使用 Web 代理（通过生成的 `calcproxy` 程序）来访问 Web 服务 WebCalc 的程序。从程序清单 18.5 可知，使用 Web 服务非常容易。第 11 行创建了一个名为 `cSrv` 的 `Calc` 对象，然后使用它来调用 Web 服务中的方法。实际上，这就像是在使用本地库一样。差别在于，您需要创建并使用 Web 代理文件，该文件负责连接到 Web 服务器上 `Calc` 类中的方法。

注意：可以将文件 `WebClac.asmx` 移到其他的 Web 服务器中。在这种情况下，您需要创建一个新的代理文件，并重新编译本地的程序。

18.4 创建常规 Web 应用程序

今天介绍的第二种 Web 应用程序是 Web 表单。Web 表单不同于 Web 服务，作为一种技术，它更接近于 Windows 窗体和您熟悉的标准程序。Web 表单是构筑动态网站的基本部件。

Web 表单应用程序并不被创建为扩展名为 .cs 的文件，而是被创建为 ASP.NET 的一部分。ASP.NET 指的是 .NET 框架中的活动服务器页面，因此 Web 表单应用程序的扩展名为 .aspx。

ASP.NET 应用程序，即 Web 表单，是终端用户在浏览器中查看的应用程序。这些应用程序可以使用任何通用的标记语言，如 HTML。另外，还可以在任何标准浏览器中查看它们。最重要的是，它们可以使用能在 Web 服务器上执行的编程代码。

您应该熟悉 HTML 以及 Web 页面是如何被显示的，您还应该知道基本浏览器是如何工作的。浏览器（客户）发送 Web 页面请求，然后 Web 服务器处理这种请求，并通过 Internet 将 Web 页面的 HTML 代码发回给发出请求的浏览器（客户）。然后，浏览器便可以处理并显示递送回来的 HTML 代码，如图 18.4 所示。

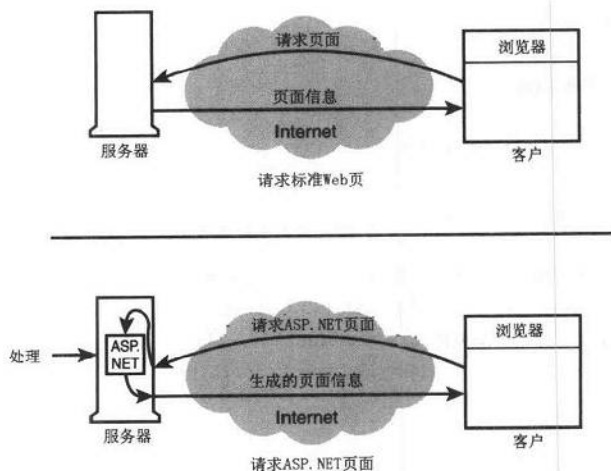


图 18.4 请求 Web 页面

对于基本 Web 页面，服务器收到的请求导致服务器将 HTML 代码发回给浏览器。微软公司提供的 ASP 可以干预这一过程，ASP 使得用户对发送给浏览器的内容有更大的控制权。

客户请求 ASP.NET 页面时，服务器将执行该页面。这种页面是在服务器，而不是浏览器上运行的。这意味着 ASP.NET 不必与浏览器兼容。这使得 ASP.NET 文件中可以包含真正的代码，如 C# 代码。这种处理方式的结果是，您可以在发送 HTML 之前，执行大量的操作，包括在发送 HTML 之前定制它。这种定制可能包括添加当前数据库中的信息，使 Web 页包含浏览器特定的特性，创建动态网页等。对于在服务器端进行的编程没有任何限制。

使用 ASP.NET，可以在 ASP.NET 应用程序中实现大量 .NET 平台的功能。这是因为 ASP.NET 文件是在服务器上执行的，而不是在客户机或客户的 Web 浏览器上执行的。对于这一点，我已重申过多次，它是 ASP.NET 功能强大的关键原因。只要服务器上运行了 .NET 运行环境和 Web 服务器，便可以创建几乎可供任何浏览器查看的网站。

ASP.NET 应用程序运行的结果将发送给客户的 Web 浏览器，因此应确保结果与大多数浏览器

兼容。Windows 窗体只与运行 .NET 运行环境的计算机兼容，因此对于创建 Web 页面而言，Windows 窗体不是一种好的解决方案。但不用担心，您可以使用标准的、老的 HTML 控件。另外，.NET 还提供了另外两类可在服务器上使用的控件。这些服务器端控件能够生成正确的标准 HTML 控件，还提供了新的、更好的信息显示方式（如果浏览器支持的话）。

18.4.1 Web 表单

Web 表单通常被分成两部分：可视化部分和代码部分。可视化部分通常定义了将创建的 Web 页面或 Web 表单的外观，即布局，包括控件或文本的位置。代码部分通常是将可视化部分组合起来的逻辑（代码），并提供了 Web 页面的功能。

18.4.2 创建一个基本的 ASP.NET 应用程序

创建一个简单的 ASP.NET 应用程序需要结合使用 HTML、ASP 脚本语言和控件。程序清单 18.6 便是一个简单的 ASP.NET 应用程序，该程序清单的运行结果如图 18.5 所示。

程序清单 18.6 firstasp.aspx: 一个简单的 ASP.NET 应用程序

```

1: <%@ Page Language="C#" %>
2:
3: <HTML>
4: <HEAD>
5:     <SCRIPT runat="server">
6:         protected void btnMyButton_Click(object Source, EventArgs e)
7:         {
8:             lblMyLabel.Text="The button was <b>clicked</b>!";
9:         }
10:    </SCRIPT>
11: </HEAD>
12:
13: <BODY>
14:     <H3>Simple Web Form Example</H3>
15:
16:     <FORM runat=server>
17:         <asp:Button id=btnMyButton
18:             runat="server"
19:             Text="My Button"
20:             onclick="btnMyButton_Click" />
21:         <br>
22:         <br>
23:         <asp:Label id=lblMyLabel
24:             runat=server />
25:     </FORM>
26: </BODY>
27: </HTML>

```

分析：输入该程序清单，并将其保存为 firstasp.aspx。您不用编译它，而是将它复制到 Web 服务器中，就像 Web 服务程序一样。如果将该程序放在目录 inetpub\wwwroot 下，则可以使用地址 localhost 来请求它。复制好该应用程序后，请在浏览器中使用下面的 URL 来执行 ASP.NET 应用程序：

<http://localhost/firstasp.aspx>

最初的运行结果如图 18.5 所示。正如您看到的，该页面显示的是标准 HTML。扩展名.aspx 表明这是一个 ASP.NET 页面（Web 表单）。Web 服务器知道，扩展名为.aspx 的文件应视为 ASP.NET 应用程序。



图 18.5 应用程序 firstasp.aspx 的运行结果

ASP.NET 应用程序可以使用多种不同的语言。程序清单 18.6 的第一行是一个标准的 ASP 编译指令，它指出该页面使用的是 C#。通过在页面的开始位置包含这样一行代码，便可以使用 C# 来编写 ASP.NET 页面。如果由于某种原因，需要使用其他的语言，则可以将“C#”改为要使用的语言。例如，要使用 Visual Basic，可以将“C#”改为“VB”。

该编译指令将被 Web 服务器截取，由于其中包含`<%`，因此 Web 服务器将把该文件作为 ASP.NET 页面进行处理。Web 服务器将标记`<%`和`%>`之间的内容看作是 ASP.NET 编译指令。

该程序清单中的其他代码类似于包含脚本元素的标准 Web 页面。第 5、17、18、20、23 和 24 行包含一些独特的东西。有两样东西值得注意。第一个是语句`runat=server`以及控件名前面的`asp:`，这些东西将在接下来的几节中介绍。

另一个独特的地方是，HTML 样式稍微有些变化。这里使用的是 XHTML 或 XML 格式，而不是常规的 HTML 格式。所有的控件标记都必须包括结束标记，这也是 XML 的一种规范。例如，`<html>`是开始标记，而`</html>`是结束标记。通常情况下，结束标记的名称与开始标记相同，只是前面加上了一个斜杠。对于有些标记，可以简写结束标记，方法是在开始标记命令的后面加上一个斜杠。程序清单 18.6 就是这样做的，即以`</>`结束。为清楚地说明这一点，下面列出了一个范例，它使用两种方法来开始和结束一个名为 XXX 的通用标记（注意，其中的空格无关紧要）：

```
< XXX attributes > text < /XXX >
< XXX attributes />
```

其中 attributes 和 text 都是可选的。

您应该单击该 Web 页上的按钮，这样做将导致事件处理程序`btnMyButton_Click`被执行。该事件处理程序将一个字符串赋给标签控件`lblMyLabel`的`Text`属性，如第 8 行所示。图 18.6 显示了单击按钮后的结果。

另外，您还应使用浏览器的 View Source 菜单项，查看页面的源文件。您看到的是前面列出的`firstasp.aspx`文件中的代码吗？不是！而是与下面类似的代码（单击按钮后）：

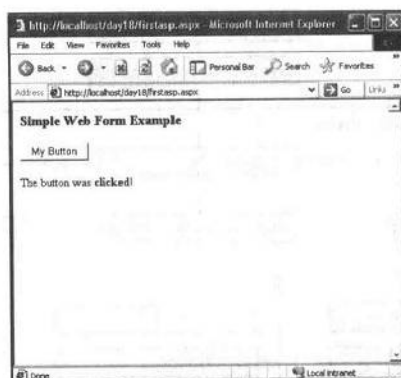


图 18.6 单击按钮后的结果

```
<HTML>
<HEAD>

</HEAD>

<BODY>

    <H3>Simple Web Form Example</H3>

    <form name="ctrl0" method="post" action="firstasp.aspx" id="ctrl0">

        <input type="hidden" name="__VIEWSTATE"
value="dDwi0DM3Nza0MzM7dDw7bLXpPDI+0z47bDx0PDtsPGk8Mz47PjtsPHQ8cDxwPGw8VGV4dDs+02w8VGhl
IGJldHRvbiB3YXNjXG9kXDI5bGlja2VkXDwvY1w+ITs+Pjs+Qzs+0z4+0z4+0z4=" />

        <input type="submit" name="btnMyButton" value="My Button"
id="btnMyButton" />

        <br>

        <br>

        <span id="lblMyLabel">The button was <b>clicked</b>!!</span>

    </form>

</BODY>

</HTML>
```

浏览器收到的不是 ASP.NET 文件，而是这种文件生成的结果，包括 HTML 代码以及对浏览器友好的控件。您可能注意到了，控件 `lblMyLabel` 被转换为一个 `HTML span` 标记，而不是一个控件，这是由 Web 服务器决定并生成的。

18.4.3 ASP.NET 控件

使用 ASP.NET 和 C# 创建 Web 页面时, 可以使用两套不同的控件: HTML 服务器控件和 Web 表单控件, 这两套控件都是在 Web 服务器而不是客户机上运行的。

18.4.3.1 HTML 服务器控件

如果您熟悉 HTML 表单控件, 将发现 HTML 服务器控件很眼熟。实际上, HTML 服务器控件与标准 HTML 控件极其类似——但它们又不是同一回事。图 18.7 说明了 .NET 框架中的 HTML 服务器控件。

续表

HTML 服务器控件	标准的 HTML 控件
HtmlInputText	<input type="text">和<input type="password">
HtmlSelect	<select>
HtmlTable	<table>
HtmlTableCell	<td>和<th>
HtmlTableRow	<tr>
HtmlTextArea	<textarea>

虽然图 18.7 列出控件的名称不同于 HTML 控件,但从中可以发现一个规律:标准 HTML 服务器控件都是按标准 HTML 控件命名的,只是前面加上了 Html。

当 aspx 文件首次被分析时,页面中所有的标准 HTML 控件都将保留不变,它们将被传递给浏览器。但如果控件的属性列表中包含 `runat=server`,则分析程序将按表 18.1 把它转换为对应的 HTML 服务器控件。通过转换为对应的 HTML 服务器控件,您便可以在服务器上操纵这些控件;如果不包含 `runat=server`,将无法在服务器上操纵这些控件,这些控件将被发送给浏览器。

程序清单 18.7 较长,它使用的是 HTML 服务器控件。该程序清单显示一个表单,让用户输入用户名和密码。在代码中,正确的用户名和密码分别被设置为 Brad 和 Swordfish。表单中包含两个输入文本框和两个按钮,这些控件的属性列表中都包含 `runat=server`,因此它们将作为 HTML 服务器控件在服务器上执行。该程序清单的运行结果如图 18.8 所示。

程序清单 18.7 htmcontrols.cs: 使用 HTML 服务器控件

```

1: <html>
2:   <script Language="C#" runat="server">
3:
4:       protected void SubmitBtn_Click(object source, EventArgs e)
5:       {
6:           if ((Name.Value == "Brad") &&
7:               (Password.Value == "Swordfish"))
8:           {
9:               Message.InnerHtml = "You Pass!";
10:          }
11:          else
12:          {
13:              Message.InnerHtml = "Incorrect user name or password.";
14:          }
15:      }
16:
17:      protected void ResetBtn_Click(object source, EventArgs e)
18:      {
19:          Name.Value = "";
20:          Password.Value = "";
21:      }

```

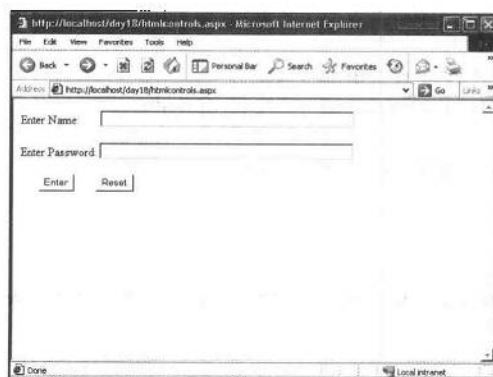
[illegible]

图 18.8 使用 HTML 服务器控件得到的结果

分析: 该程序清单的结构与前一个稍有不同, 它直接进入HTML, 而不是在开始使用ASP.NET 中的Page编译指令。第2行包含一组脚本代码, 这是一个标准的脚本标记吗? 实际上, 它包含编译指

令 `runat=server`，因此它实际上是在服务器上运行的 ASP 代码。这意味着当该表单在服务器上执行时，可以使用脚本功能。如果没有包含 `runat=server`，则这将是一个被发送给浏览器的标准脚本标记。

接下来的几行是用作脚本的 C# 代码。由于该脚本将在服务器上执行，因此可以使用 C# 这些代码检查用户名和密码是否有效，并根据结果设置消息字段。

表单是从第 24 行开始的。表单上的控件都是标准的，唯一特别的是，它们都包含属性 `runat="server"`，这将把控件和表单转换为 HTML 服务器控件。如果您了解标准 HTML，则应该能够理解其余的代码。

警告： 如果不了解标准 HTML，应在学习 Web 表单和 ASP.NET 之前学习它。

运行该 ASP.NET 页面，输入正确的用户名和密码后，将得到如图 18.9 所示的结果。

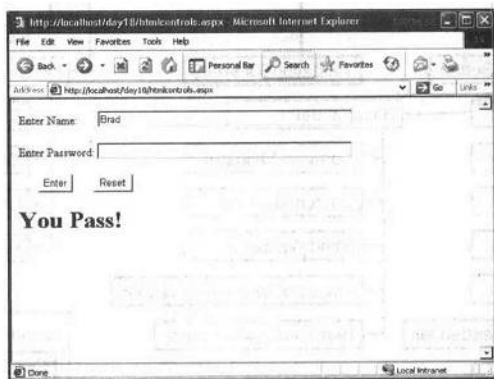


图 18.9 正确登录后，HTML 服务器控件程序的运行结果

可以像使用输入按钮 (`input button`) 一样，使用其他任何标准 HTML 控件。有关各个控件中可操纵的属性，可参阅在线文档。

提示： 注意这个表单中没有使用标签控件。生成 HTML 时，标准文本将作为发送给浏览器的 HTML 文件的一部分。这意味着您无需使用标签来显示信息，而可以使用标准 HTML。仅当需要修改显示的信息时，才应使用标签。

18.4.3.2 Web 服务器控件

除了 HTML 服务器控件外，还可以在 ASP.NET 应用程序中使用 Web 服务器控件。这些控件与第 16 和 17 天介绍的 Windows 表单控件极其类似。图 18.10 列出了一些常用的 Web 服务器控件。

您通常使用 Web 服务器控件来创建 Web 表单。从程序清单中，可以辨别出 Web 服务器控件，因为 Web 服务器控件除了编译指令 `runat=server` 外，前面还有 `asp:`。从程序清单 18.8 可以知道这一点，该程序清单列出了另一个简单的 Web 表单应用程序——这里使用的是 Web 服务器控件。该程序清单的运行结果如图 18.11 所示。

警告： 不要混淆 Web 表单应用程序的扩展名和 Web 服务应用程序的扩展名，前者为 `.aspx`，而后者为 `.asmx`。

程序清单 18.8 webform.aspx: 使用 Web 服务器控件

```
1: <%@ Page Language="C#" %>
2:
3: <HTML>
```

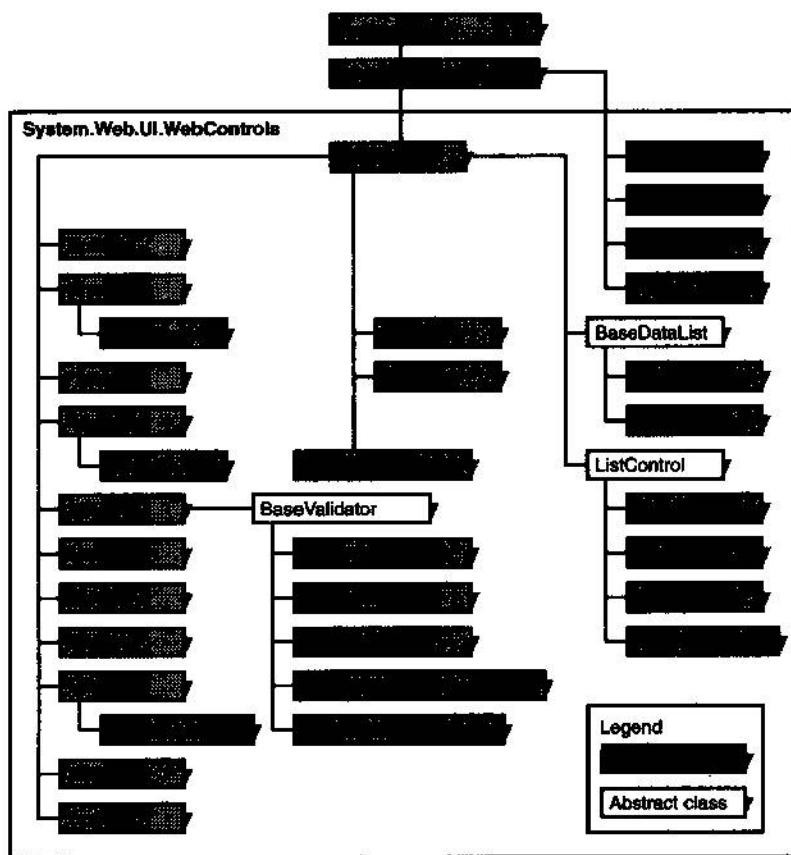


图 18.10 Web 服务器控件

```

4: <HEAD>
5:     <SCRIPT RUNAT="SERVER">
6:         protected void Button1_Click(object Source, EventArgs e)
7:         {
8:             DateTime currDate = new DateTime();
9:             currDate = DateTime.Now;
10:            myDateLabel.Text = currDate.ToString();
11:        }
12:    </SCRIPT>
13: </HEAD>
14: <BODY>
15:     <H3 align="center">Simple Web Server Controls Example</H3>
16:
17:     <FORM runat=server>
18:         <center><asp:Label id=myDateLabel runat="server" />
19:         <br><br>
20:         <asp:Button id=Button1 runat="server"
21:             Text="Update"
22:             onclick="Button1_Click" />
23:     </center>
24: </FORM>

```

```
25: </BODY>
26: </HTML>
```



图 18.11 webform.aspx 在浏览器中的显示结果

分析：该程序清单与 18.6 极其类似。页面的开始位置是 Page 编译指令，指出要使用的语言。该程序清单在用户单击 Update 按钮时显示日期和时间，这与本书前面使用 Windows 窗体创建的应用程序类似。第 8~10 行的代码也相同，这些代码取得日期值，并将其赋给标签控件。第 18 行的标签是一个 Web 服务器控件，这是因为它以 asp: 开始，并以 runat=server 结束。显然如果使用定时器，该程序清单将更佳，但这样便无法演示结合使用标签控件和 Web 服务器按钮控件的情况。

该程序清单在按钮被单击时显示日期和时间，如图 18.12 所示。这里显示的时间是服务器的还是浏览器的呢？正确的答案是服务器的，因为代码是在服务器上执行的！



图 18.12 Update 按钮被单击后，webform.aspx 的输出

这里也应该查看一下发送给浏览器的 HTML 代码。前面已经重申多次了，Web 服务器控件和 HTML 服务器控件是在服务器上执行的。来看看程序清单 18.8 发送给浏览器的源代码，方法是在浏览器中选择菜单项 View Source。显示的源代码与下面类似：

```
<HTML>
<HEAD>

</HEAD>
<BODY>
```

```

<H3 align="center">Simple Web Server Controls Example</H3>

    <form name="ctrl0" method="post" action="webform.aspx" id="ctrl0">
    <input type="hidden" name="__VIEWSTATE"
    value="dDwtMTA2MDQwMDUyMDt0PDtsPGk8Mj47PjtsPHQ8O2w8aTwxPjs+O2w8dDxwPHA8bDxUZXh0Oz4/bDw3
    LzkvMjAwMSA5OjQwOjAxIFBNOz4+Oz47Oz47Pj47Pj47Pg==" />

    <center><span id="myDateLabel">7/9/2001 9:40:01 PM</span>
    <br><br>
    <input type="submit" name="Button1" value="Update" id="Button1" />
    </center>
    </form>
</BODY>
</HTML>

```

这些代码完全不同于原程序清单中的代码。

18.5 总 结

今天的课程分两部分，第一部分介绍了如何创建和使用 Web 服务。Web 服务是位于 Web 上某个地方的一个代码片段，可以在程序中调用它。由于制定了相应的通信标准，因此调用和使用这种 Web 服务相对简单。

第二部分简要地介绍了基于 Web 的表单应用程序。可以在 ASP.NET 中使用 C#来创建以 Web 为中心的动态应用程序，这些信息足以引起您的兴趣。当前，市面上有大量专门介绍 ASP.NET 和 Web 表单编程的图书。

18.6 问与答

问：今天介绍了大量的内容，但都是浅尝辄止。为何不更全面、深入地介绍这方面的内容呢？

答：正如今天课程的开始指出的，Web 服务和 Web 表单都足以独自成一本书。另外，这些主题涉及到的是 C#的应用，而不是 C#语言的一部分。因此，许多关于 C#图书根本不涉及这些主题。作者认为，这些主题很重要，很多人都会感兴趣，因此有必要简要介绍一下与 Web 开发相关的 Web 技术。

问：Web 服务和使用 Web 服务的客户程序中的代码，与普通的代码并没有太大的差别，它们不能更复杂些吗？

答：今天介绍 Web 服务和客户时，使用的代码都非常简单。制定跨越 Web 的通信标准的工作量非常大。Web 服务的复杂性在于通信，而 wsdl 工具已经为您创建了这些复杂的代码。通过制定通信标准，消除了应用程序中大部分的复杂内容，这样应用程序可以将重点放在要实现的功能，而不是通信方面。

问：必须使用 wsdl.exe 来生成代理文件吗？

答：不。可以手工编写这些代码或使用诸如 Visual Studio 等能够帮助您生成一些所需代码的工具。

问：前面指出过，存在服务器控件和 HTML 控件，而 HTML 控件不同于浏览器中使用的标准

HTML 控件，那么哪些控件属于 HTML 控件呢？

答：微软公司开发了一条在 Web 服务器上运行的 HTML 控件，这些控件对应于原来在浏览器上运行的 HTML 控件。事实上，HTML 控件通常生成 HTML 浏览器控件。需要知道的重要的一点是，在服务器上运行的 HTML 控件将能够适应浏览器的处理能力。

18.7 作业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

18.7.1 小测验

1. 何为 Web 服务？
2. 帮助客户程序与 Web 服务进行通信的文件被称为什么？
3. 哪个程序用于创建与 Web 服务器通信的代码？
4. 如何区分 Web 服务和 ASP.NET 页面？
5. 如何执行 ASP.NET 页面？
6. 哪两类控件可用于 Web 表单？
7. 程序清单 18.6 使用的是 HTML 控件、HTML 服务器控件、还是 Web 服务器控件？
8. 标准 HTML 控件和 HTML 服务器控件的区别何在？
9. 与标准 HTML 表格标记对应的是哪种服务器控件？
10. 如何区分服务器端 HTML 控件和标准 HTML 控件？

18.7.2 练习

1. 用作 Web 服务的 C# 程序的第一行代码是什么？假设 Web 服务类名为 DBInfo，其第一个方法名为 GetData。
2. 要将方法用于 Web 服务中，需要对其做哪些修改？
3. 在 Web 服务 Calc 加入方法 Multiply 和 Divide。
4. 新建一个客户程序，使用练习 3 创建的方法 Multiply 和 Divide。
5. 选做题：创建一个使用 HTML 服务器控件的 Web 页面，然后重新编写该应用程序，以使用 ASP 服务器控件。
6. 选做题：在编译器自带的在线文档中查阅关于 HTML 服务器控件和 Web 服务器控件以及这些控件的属性、方法和事件等信息。

第 19 天课程

C#中的编译指令和调试技术

今天进入一个大多数程序员都想避开的领域，介绍调试方面的一些知识，以及如何使用编译指令。具体地说，今天介绍以下内容：

- 何为调试？
- 程序可能包含的主要错误类型。
- 如何让编译器忽略程序中的一些代码？
- 修改编译器报告的行号。
- 如何在代码中和编译时定义符号（symbol）。
- 在程序中定义可供 IDE 使用的区域。

19.1 何为调试

当编译或执行程序期间发生错误时，您需要确定问题所在。对于诸如本书使用的范例这样的小型程序中，检查程序清单并找出问题比较容易；但对于大型程序，找出错误要难得多。

新术语：查找并消除错误的过程被称为**调试**。错误通常指的是程序中的 bug。最初，导致计算机出现故障的原因之一是虫子——具体地说，是一只蛾子。管理员在计算机中找到这只虫子，并将其拿走。虽然这种故障是由一只真正的虫子引起的，但后来人们通常将计算机错误称为 bug。

提示：在电视节目《百万富翁》中，知道这只虫子是一只蛾子，是一个价值一百万美元的问题。这种无关紧要的事实有时候会变得非常重要。

19.2 错误类型

新术语：在本书的开始介绍过，错误类型很多，您必须捕获大部分错误后，才能运行程序。编译器将捕获这些错误，并以错误或警告的方式告诉您。其他错误难以找到，例如，您编写的程序可能在编译时没有任何错误，但运行结果与您期望的不同。这种错误被称为**逻辑错误**。另外，您编写的程序能够通过编译，但当最终用户输入不正确的信息，程序收到的数据不是其希望的，或出现其他数不胜数的情况之一时，程序将出错。

19.3 查找错误

语法错误通常在编译时便能发现。编译时，编译器将以错误和警告的方式指出这类问题，并提供错误的位置和描述。

导致运行错误的原因有几种。前面介绍过，防止这种错误导致程序崩溃的方法是，在程序中加入异常处理。例如，如果程序试图打开一个不存在的文件将引发异常。通过加入异常处理，可以捕获并处理运行阶段的异常错误。

另一种运行错误是由于用户输入不正确的信息导致的。例如，如果您使用一个整型变量来存储用户的年龄，从理论上说，用户可能输入 30000 或其他无效的数字。这不会导致异常或其他类型的错误，但对程序而言，这仍然是一个问题，因为这种数据是不正确的。这种问题很容易通过添加一些编程逻辑，对用户输入的值进行检查来解决。

有大量的运行错误难以发现，它们是逻辑错误，从语法上说，是正确的，也不会导致程序崩溃，却导致错误的结果。对于这类逻辑错误和一些较复杂的异常，要找到它们，需要花费更多的精力，而不仅仅是检查程序代码。这种错误需要进行认真地调试。

新术语：找出这种较为复杂的错误的方式之一是逐行执行程序中的代码，您可以手工进行，也可以使用自动化工具。这种自动化工具被称为**调试器**。您还可以利用C#提供的一些特性来查找这种错误，包括使用编译指令和一些内置类。

19.4 逐行检查代码——跟踪代码

逐行检查代码要求每次阅读一行代码。您首先阅读将执行的第一行代码，然后按执行次序逐行阅读。也可以阅读每一个类定义，确保其包含的逻辑是正确的。以手工方式进行时，这繁琐耗时，需要大量的时间，并容易出错；但优点是您将完全理解程序中的代码。

注意：许多公司将逐行检查代码作为开发过程的标准部分。通常，这要求与项目相关的人详细阅读其中的代码，您需要向其他参与者解释其中的代码。您可能认为，这样做没有太大的用处，但在此过程中您常常能够找到更好的方式来完成同样的任务。另外，您还能更深入地理解其中的代码。

19.5 预处理器编译指令

C#提供了大量的编译指令，您可以在代码中使用它们。这些编译指令可以决定编译器如何处理代码。如果您使用过 C 或 C++ 进行编程，则可能熟悉这类编译指令。但在 C# 中，编译指令更少。表 19.1 列出了 C# 中的编译指令，接下来的几节将介绍其中的一些重要的编译指令。

表 19.1 C#编译指令	
编译指令	描述
<code>#define</code>	定义一个符号
<code>#else</code>	开始 else 语句块
<code>#elif</code>	else 语句和 if 语句的组合

续表

编译指令	描 述
<code>#endregion</code>	指示结束区域
<code>#endif</code>	结束 <code>#if</code> 语句
<code>#if</code>	测试一个值
<code>#error</code>	编译时发送指定的错误消息
<code>#line</code>	指定源代码的行号，也可以包含将出现在输出中的文件名
<code>#region</code>	指示区域的开始，区域（region）是 IDE 中可以展开或折叠一段代码
<code>#undef</code>	取消对符号的定义
<code>#warning</code>	编译时，发送指定的错误消息

注意：在 C 和 C++ 中，这些编译指令被称为预处理器编译指令，因为在编译代码之前，编译器将预先处理程序，并对预处理器编译指令进行评估。这些编译指令仍然和名称预处理器一起使用，但编译器评估编译指令之前，并不一定需要进行预处理。

19.5.1 预处理声明

编译指令很容易识别，它们以 `#` 打头，位于代码行的开始位置；但不以分号结尾。

首先需要知道的编译指令是 `#define` 和 `#undef`。这些编译指令让您能够定义符号或取消对符号的定义，符号可用于确定程序中包含哪些代码。通过包含或排除代码，可以使得相同的代码以多种方式使用。

这些编译指令最常见的用途之一是用于调试。创建程序时，您常常希望它能生成一些在生产环境中不被显示的信息。在这种情况下，可以定义编译指令定义一个值，或取消对一个值的定义；而不用不断地添加和删除这些代码。

`#define` 和 `#undef` 的基本格式如下：

```
#define xxxx
#undef xxxx
```

其中 `xxxx` 是要定义或取消其定义的符号的名称。程序清单 19.1 使用了本书前面的一个程序清单中的代码，显示命令行提供的文件的内容。

警告：该程序清单没有包含异常处理代码，因此可能会出错。例如，如果您试图打开一个不存在的文件，将引发异常。

程序清单 19.1 reading.cs：使用编译指令 `#define`

```
1: // reading.cs - Read text from a file.
2: // Exception handling left out to keep listing short.
3: //-----
4:
5: #define DEBUG
6:
```



```

7: using System;
8: using System.IO;
9:
10: public class ReadingApp
11: {
12:     public static void Main(String[] args)
13:     {
14:         if( args.Length < 1 )
15:         {
16:             Console.WriteLine("Must include file name.");
17:         }
18:         else
19:         {
20:
21: #if DEBUG
22:
23:     Console.WriteLine("=====DEBUG INFO=====");
24:     for ( int x = 0; x < args.Length ; x++ )
25:     {
26:         Console.WriteLine("Arg[{0}] = {1}", x, args[x]);
27:     }
28:     Console.WriteLine("=====");
29:
30: #endif
31:
32:         string buffer;
33:
34:         StreamReader myFile = File.OpenText(args[0]);
35:
36:         while ( (buffer = myFile.ReadLine()) != null )
37:         {
38: #if DEBUG
39:             Console.Write( "{0:D3} - ", buffer.Length);
40: #endif
41:             Console.WriteLine(buffer);
42:         }
43:
44:         myFile.Close();
45:     }
46: }
47: }

```

该程序清单的输出如下:

```

=====DEBUG INFO=====
Arg[0] = reading.cs
=====
041 - // reading.cs - Read text from a file.
054 - // Exception handling left out to keep listing short.
054 - //-----

```

```

000 -
013 - #define DEBUG
000 -
013 - using System;
016 - using System.IO;
000 -
023 - public class ReadingApp
001 - {
041 -     public static void Main(String[] args)
004 -     {
027 -         if( args.Length < 1 )
007 -         {
055 -             Console.WriteLine("Must include file name.");
007 -         }
010 -         else
007 -         {
000 -
009 - #if DEBUG
000 -
064 -     Console.WriteLine("=====DEBUG INFO=====");
043 -     for ( int x = 0; x < args.Length ; x++ )
004 -     {
054 -         Console.WriteLine("Arg[{0}] = {1}", x, args[x]);
004 -     }
065 -     Console.WriteLine("=====");
000 -
006 - #endif
000 -
023 -         string buffer;
000 -
054 -         StreamReader myFile = File.OpenText(args[0]);
000 -
055 -         while ( (buffer = myFile.ReadLine()) != null )
010 -         {
010 - #if DEBUG
046 -             Console.Write( "{0:D3} - ", buffer.Length);
006 - #endif
039 -             Console.WriteLine(buffer);
010 -         }
000 -
024 -         myFile.Close();
007 -     }
004 - }
001 - }

```

分析: 该程序清单包含多个变异指令。如果定义了DEBUG, 则该程序清单将提供额外的输出。第5行定义了DEBUG, 因此编译并运行该程序时, 都将生成额外的输出。如果将第5行注释(删除)掉, 然后重新编译并运行该程序, 则不会显示额外的信息。

额外的信息是哪些呢? 第21行使用了另一个编译指令——`#if`。如果`#if`后面的值被定义了, 则

该语句的结果为 `true`；否则为 `false`。由于第 5 行定义了 `DEBUG`，所以程序将包含接下来的代码；否则将直接进入第 30 行的 `#endif` 语句。

第 22~29 行打印命令行参数，以便您知道输入的内容。将该程序作为产品发布时，可以删除对 `DEBUG` 的定义，这样这些信息将不会被显示，因为编译时将忽略生成这些信息的代码。

第 38 行包含另一个 `#if` 语句，它再次检查 `DEBUG`。这里在每行的开始位置打印一个值——被打印行的长度，这种信息可用于调试。同样，将该程序作为产品发布时，通过取消对 `DEBUG` 的定义，这些信息将不会被显示。

正如您看到的，定义一个值比较简单。使用编译指令的作用之一是可以避免修改代码，但需要修改 `DEBUG` 的定义情况，即修改程序清单 19.1 的第 5 行。对于这种情况，另一种方法是在编译时定义一个值。

19.5.1.1 在命令行定义值

请删除程序清单 19.1 中的第 5 行，然后重新编译并运行它，此时额外的调试信息将不会出现。要定义 `DEBUG`，可以在编译时使用 `/define` 标记，而无需重新在程序清单中加入第 5 行。这种编译选项的格式如下：

```
csc /define:DEBUG reading.cs
```

其中 `DEBUG` 是要在程序中定义的值，而 `reading.cs` 是程序名。如果使用 `/define` 开关编译程序清单 19.1，则 `DEBUG` 将再次被定义，而无需修改代码。不在命令行中使用 `/define` 标记，调试信息将不会出现。因此，无需修改任何代码，便可以控制调试信息是否出现。

提示：可以使用 `/define` 的简写格式 `/d`。

注意：如果您使用的是 IDE，请查阅与定义编译指令相关的文档。IDE 中应该提供对话框，让您输入要定义的符号。

19.5.1.2 #define 和 #undef 的位置

虽然前面没有演示，但您也可以使用 `#undef` 来取消对值的定义。从 `#undef` 所在位置到程序末尾，`#undef` 命令中的符号将不再是被定义的。

编译指令 `#undef` 和 `#define` 必须位于程序中真正的代码之前，它们可以位于注释和其他编译指令的后面，但不能位于声明或其他代码之后。

警告：不能将 `#define` 或 `#undef` 放在程序的中间。

19.5.2 条件处理（`#if`、`#elif`、`#else`、`#endif`）

正如您看到的，可以对定义的值使用 `if` 逻辑。C# 提供了完整的 `if` 逻辑，包括 `#if`、`#elif`、`#else` 和 `#endif`。这可以实现 `if`、`if...else` 和 `if...else if` 等逻辑结构。不管使用哪种格式，都必须以编译指令 `#endif` 结束。程序清单 19.1 中使用了 `#if`。 `if` 逻辑的一种常见用途是确定被编译的程序是开发版还是发行版：

```
#if DEBUG
// do some debug stuff
#elif PRODUCTION
// do final release stuff
#else
// display an error regarding the compile
#endif
```

上述程序清单生成的结果将随定义的值而异。

19.5.2.1 预处理表达式 (!, ==, !=, &&, ||)

编译指令的 if 逻辑可以包含多种运算符: !、==、!=、&& 和 ||。这些运算符的工作方式与在标准 if 语句中相同。! 检查是否不是某个值; == 检查是否相同; != 检查是否不等; && 用于检查是否多个条件皆为 true; || 用于检查是否至少一个条件为 true。

在程序中加入常用检查的格式如下:

```
#if DEBUG && PRODUCTION
//Produce an error and stop compiling
```

如果 DEBUG 和 PRODUCTION 都已定义, 则说明有问题。接下来的一节将介绍如何指出问题。

19.5.3 报告代码中的错误和警告 (#error、#warning)

由于编译指令是编译的一部分, 因此您希望能够指出错误和警告。前一节介绍过, 如果 DEBUG 和 PRODUCTION 都已定义, 则说明存在严重的问题, 需要引发错误。为此, 可以使用编译指令 #error。如果您希望仍然编译程序, 并创建可执行文件——如果其他代码一切正常, 则可以使用编译指令 #warning 来生成警告。程序清单 19.2 对 19.1 做了修改, 使用了编译指令 #error 和 #warning。

程序清单 19.2 reading2.cs: 使用编译指令 #error 和 #warning

```
1: // reading.cs - Read text from a file.
2: // Exception handling left out to keep listing short.
3: // Using the #error & #warning directives
4: //- --- -----
5:
6: #define DEBUG
7: #define BOOKCHECK
8:
9: #if DEBUG
10:    #warning Compiled listing in Debug Mode
11: #endif
12: #if BOOKCHECK
13:    #warning Compiled listing with Book Check on
14: #endif
15: #if DEBUG && PRODUCTION
16:    #error Compiled with both DEBUG and PRODUCTION!
17: #endif
18:
19: using System;
20: using System.IO;
21:
22: public class ReadingApp
23: {
24:     public static void Main(String[] args)
25:     {
26:         if( args.Length < 1 )
27:         {
28:             Console.WriteLine("Must include file name.");
29:         }
```

```

30:     else
31:     {
32:
33: #if DEBUG
34:
35:     Console.WriteLine("=====DEBUG INFO=====");
36:     for ( int x = 0; x < args.Length ; x++ )
37:     {
38:         Console.WriteLine("Arg[{0}] = {1}", x, args[x]);
39:     }
40:     Console.WriteLine("=====");
41:
42: #endif
43:
44:         string buffer;
45:
46:         StreamReader myFile = File.OpenText(args[0]);
47:
48:         while ( (buffer = myFile.ReadLine()) != null )
49:         {
50:
51: #if BOOKCHECK
52:
53:         if (buffer.Length > 72)
54:         {
55:             Console.WriteLine("*** Following line too wide to present in book ***");
56:         }
57:         Console.Write( "{0:D3} - ", buffer.Length);
58:
59: #endif
60:             Console.WriteLine(buffer);
61:         }
62:
63:         myFile.Close();
64:     }
65: }
66: }

```

编译该程序清单时，将出现以下两条警告：

```

reading2.cs(10,12): warning CS1030: #warning: 'Compiled listing in Debug Mode'
reading2.cs(13,12): warning CS1030: #warning: 'Compiled listing with Book Check on'

```

如果在命令行定义 **PRODUCTION**，则将出现以下警告和错误：

```

reading2.cs(10,12): warning CS1030: #warning: 'Compiled listing in Debug Mode'
reading2.cs(13,12): warning CS1030: #warning: 'Compiled listing with Book Check on'
reading2.cs(16,10): warning CS1029: #error: 'Compiled listing with both DEBUG and PRODUCTION!'

```

注意：PRODUCTION是在编译时在命令行使用/d:PRODUCTION定义的。

分析：该程序清单的开始几行使用了编译指令`#warning`和`#error`。警告让编译该程序清单的人知道当前使用的模式。这里包含两种模式：`DEBUG`和`BOOKCHECK`。第15行执行检查，确保程序清单编译时，没有同时定义`DEBUG`和`PRODUCTION`。

该程序清单中的大部分代码都简单易懂。加入的`BOOKCHECK`是针对本书的作者的，纸张对可显示的代码行宽度有一定的限制。该编译指令用于包含第52~58行的代码，这些代码检查代码行的长度是否合适。如果代码行包含的字符超过72个，则打印一条消息，然后显示该代码行及其长度；否则按正常情况打印该代码行。通过取消对`BOOKCHECK`的定义，可以删除这种逻辑。

19.5.4 修改行号

C#提供的另一种编译指令是`#line`，它让您能够修改代码的行号。这种影响可以在错误消息中看到。程序清单19.3使用了编译指令`#line`。

程序清单 19.3 lines.cs: 使用编译指令`#line`

```

1: // lines.cs -
2: //-----
3:
4: using System;
5:
6: public class linesApp
7: {
8:     #line 100
9:     public static void Main(String[] args)
10:    {
11:        #warning In Main...
12:        Console.WriteLine("In Main...");
13:        myMethod1();
14:        myMethod2();
15:        #warning Done with main
16:        Console.WriteLine("Done with Main");
17:    }
18:
19:    #line 200
20:    static void myMethod1()
21:    {
22:        Console.WriteLine("In Method 1");
23:        #warning In Method 1...
24:        int x; // not used. Will give warning.
25:    }
26:
27:    #line 300
28:    static void myMethod2()
29:    {
30:        Console.WriteLine("in Method 2");
31:        #warning In Method 2...
32:        int y; // not used. Will give warning.
33:    }
34: }
```

编译该程序清单，将显示如下所示的警告：

```
lines.cs(102,16): warning CS1030: #warning: "In Main..."
lines.cs(106,16): warning CS1030: #warning: "Done with main"
lines.cs(203,16): warning CS1030: #warning: "In Method 1..."
lines.cs(303,16): warning CS1030: #warning: "In Method 2..."
lines.cs(204,11): warning CS0168: The variable "x" is declared but never used
lines.cs(304,11): warning CS0168: The variable "y" is declared but never used
```

该程序清单的运行结果如下：

```
In Main...
In Method 1
In Method 2
Done with Main
```

分析：该程序清单并没有什么实用价值，但演示了编译指令`#line`。每种方法以不同的行号开始。Main方法从行号100开始；myMethod1从行号200开始；而myMethod2从行号300开始。这让您能够根据这些行号指出程序的什么位置有问题。

从编译器输出可以知道，警告中的行号是根据 `#line` 中的值进行编号的，而不是实际的行号。显然，该程序清单中没有 100 行代码！在编译指令`#warning` 以及编译器生成的错误和警告中，使用的是编译指令`#line` 的行号。

也可以让程序清单中的某些部分使用默认的行号：

```
#line default
```

上述代码使得以后的代码使用默认的行号。

提示：在ASP.NET中使用C#进行开发时，行号将很有用；除此之外，通常还是使用默认行号更佳。

19.5.5 区域简介

表 19.1 中还未被介绍的编译指令是`#region` 和 `#endregion`。这些编译指令用于将程序清单中的代码划分为区域。图形开发环境（如 Visual Studio.NET）使用区域来展开和折叠代码。编译指令`#region` 指示区域的开始，而 `#endregion` 指示区域的结束。

提示：一种非常好的使用编译指令来帮助调试的方式是，将要写入到控制台的消息包含进来。这些消息可以包含关于变量内容的信息、在程序中的位置或其他有帮助的信息。这些信息可以包含在 `#if` 语句块中，而通过取消对符号（如DEBUG）的定义或根本不定义符号，可以排除这种语句块。

注意：基类库包含一些可用于执行跟踪和调试工作的类，这些类使用符号DEBUG和TRACE来帮助显示关于程序中发生的情况的信息。介绍这些内容超出了本书的范围，但您可以查阅类库参考，从中找到关于名称空间System.Diagnostics的信息，该名称空间包含Trace和Debug类。

19.6 使用调试器

调试器的主要用途之一是，将逐行检查程序的过程自动化。调试器让您能够每次执行程序中的

一行代码。每执行一行代码，您便可以查看变量或其他数据成员的值；可以跳到程序的其他地方，甚至可以略过某些代码行，阻止它们执行。

注意：介绍如何使用调试器超出了本书的范围。Visual Studio和许多其他的C# IDE内置了调试器。另外，Microsoft .NET框架也自带了一个名为CORDBG的命令行调试器。

19.7 总 结

今天介绍了使用编译指令来告诉编译器如何处理。这包括通过定义符号或取消对符号的定义，来包含或排除代码；用于进行决策的`#if`、`#if !`、`#else`、`#endif`语句；如何修改编译器用于指出错误位置的行号；如何使用编译指令`#error`和`#warning`在编译时生成自己的错误和警告。

19.8 问与答

问：哪种定义符号的方式更好：在命令行中定义还是在程序中定义？

答：如果在程序中定义，则必须删除该定义或在程序中取消定义。使用命令行定义的方式，在定义和取消定义之间切换时，不影响代码。

问：取消未定义的符号将出现什么情况？

答：不出现任何情况。可以多次取消对符号的定义，而不会发生错误。

19.9 作 业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

19.9.1 小测验

1. 何为调试？
2. 预处理编译指令以分号结尾吗？
3. 是哪种虫子使得术语调试（debugging，即除掉虫子）与计算机永远地关联在一起？
4. 编译器能够发现哪种错误？
5. 逐行查看代码时，需要核对程序的哪些部分？
6. 哪种语言的编译指令最少：C、C++还是C#？
7. 用于在程序中定义符号和取消符号定义的是哪些编译指令？
8. 哪种标记用于在命令行定义符号？
9. 哪个编译指令让您能够修改编译器使用的行号？哪个值可用于将行号恢复为实际行号？
10. 基类库中包含哪些用于帮助调试和其他诊断的类？

19.9.2 练习

1. 编写定义用于预处理的符号的代码，该符号名为SYMBOL。
2. 编写使用行号从1000开始的代码。
3. 下面的程序能够通过编译吗？如果能，其功能是什么？


```
1: // fun.cs - Using Directives in a goofy way
```



```

2: //-----
3:
4: #define AAA
5: #define BBB
6: #define CCC
7: #define DDD
8: #undef CCC
9: using System;
10: #warning This listing is not practical...
11: #if DDD
12: public class
13: #endif
14: #if CCC
15: destroy();
16: #endif
17: #if BBB || EEE
18: myApp { public static void
19: #endif
20: #region
21: #if GGG
22: Main() {Console.WriteLine("Lions");
23: #elif AAA
24: Main() {Console.WriteLine(
25: #elif GGG
26: Console.ReadLine(
27: #else
28: Console.WriteLine(
29: #endif
30: "Hello"
31: #if AAA
32: + " Goofy "
33: #else
34: + " Strange "
35: #endif
36: #if CCC && DDD
37: + "Mom"
38: #else
39: + "World"
40: #endif
41: );}
42: #endregion
43: }
44:

```

4. 下面的程序有问题吗？如果有，是哪一行导致的错误消息？编译该程序时，应定义符号 MYLINES。

```

1: // bugbust.cs -
2: //-----
3:

```

```
4: using System;
5:
6: public class ReadingApp
7: {
8:     #if MYLINES
9:     #line 100
10:    #endif
11:    public static void Main(String[] args)
12:    {
13:        Console.WriteLine("In Main....");
14:        myMethod1();
15:        myMethod2();
16:        Console.WriteLine("Done with Main");
17:    }
18:
19:    #if MYLINES
20:    #line 200
21:    #endif
22:    static void myMethod1()
23:    {
24:        Console.WriteLine("In Method 1");
25:    }
26:
27:    #if MYLINES
28:    #line 300
29:    #endif
30:    static void myMethod2()
31:    {
32:        Console.WriteLine("in Method 2");
33:    }
34:    #undef MYLINES
35: }
```

5. 选做题：在今天创建的某个程序清单中加入编译指令，来打印关于程序中位置的消息，这些消息仅在符号 REPORT 被定义时，才被显示。

6. 选做题：在今天创建的某个程序清单中加入编译指令，使得符号 COMMENTS 被定义时，显示注释，而忽略程序中的其他内容（实际的代码）。

第 20 天课程

运算符重载

今天深入介绍使用类时可用的一些功能，包括讨论重载。前面已经介绍过方法重载，今天的课程包含以下内容：

- 再谈方法和构造函数重载；
- 运算符重载；
- 如何重载单目运算符、双目运算符、关系运算符和逻辑运算符？
- 重载逻辑运算符的独特之处；
- 指出哪些运算符可以重载，哪些不可以。

20.1 再谈函数重载

第 9 天介绍过，可以将方法重载多次。重载方法的关键在于，必须确保每次重载时，都有不同的特征标。特征标由方法的参数和返回类型决定。例如，下面都是不同的特征标：

```
int mymethod( int x, int y)
int mymetnod( int x)
int mymetnod( long x)
int mymethod( char x, long y, long z)
int mymethod( char x, long y, int z)
```

第 9 天还介绍过，可以像常规方法那样重载构造函数，今天不介绍重载方法，而是介绍如何重载类的运算符。

20.2 运算符重载

除了构造函数和存取器外，许多面向对象语言还允许重载运算符，C#也不例外。C#允许您重载许多数学运算符（如加和减）、关系运算符和逻辑运算符。

为何要重载这些运算符呢？没有理由要求您必须重载它们，但有时候重载将使得程序更容易理解，类更容易使用。

类 `String` 就是一个包含重载运算符的绝佳范例。正常情况下，不能对类执行加运算，但在 C# 中，可以使用加号运算符将两个字符串加起来。这种加法功能与您想象的相同——将两个字符串组

合起来。例如, "animal" + " " + "crackers" 的结果为字符串 "animal crackers"。

为实现这种功能, String 类和 string 数据类型重载了加法运算符。您将发现, 重载运算符也可使您的程序更好的工作。我们来看看程序清单 20.1。

注意: 该程序清单并不是一个使用运算符重载的好范例, 但它简单, 这样您可以将注意力集中在概念上, 而不是理解其中的代码。今天给出的其他一些程序清单将更实用。

程序清单 20.1 overl.cs: 一个有问题的程序

```

1: // overl.cs - A listing with a problem
2: //-----
3:
4: using System;
5:
6: public class AChar
7: {
8:     private char CH;
9:
10:    public AChar() { this.ch = ' '; }
11:    public AChar(char val) { this.ch = val; }
12:
13:    public char ch
14:    {
15:        get{ return this.CH; }
16:        set{ this.CH = value; }
17:    }
18: }
19:
20: public class myAppClass
21: {
22:
23:    public static void Main(String[] args)
24:    {
25:        AChar aaa = new AChar('a');
26:        AChar bbb = new AChar('b');
27:
28:        Console.WriteLine("Original value: {0}, {1}", aaa.ch, bbb.ch);
29:
30:        aaa = aaa + 3;
31:        bbb = bbb - 1;
32:
33:        Console.WriteLine("Final values: {0}, {1}", aaa.ch, bbb.ch);
34:    }
35: }
```

编译该程序清单时, 将出现如下所示的错误:

```

overl.cs(30,13): error CS0019: Operator '+' cannot be applied to operands of type 'Achar'
and 'int'
overl.cs(31,13): error CS0019: Operator '-' cannot be applied to operands of type 'Achar'
and 'int'
```

分析：该程序清单很容易理解。它创建了一个名为Achar的类。这个类不实用，但简单，使用它来说明重载很容易。第二个类将在今天后面的程序清单中使用。

Achar 存储一个字符，它有两个构造函数，如第 10 和 11 行所示。第一个构造函数在没有提供参数时被调用，它将被实例化的对象的字符值设置为空格；第二个构造函数接受一个字符，并将其赋给实例化的对象的私有字符变量。这个类使用一个存取器（如第 13 ~ 17 行所示）来设置字符值。

MyAppClass 使用了 Achar 类。第 25 和 26 行创建了两个 Achar 对象：aaa 和 bbb，前者包含 'a'，而后者包含 'b'。第 33 行打印这些值，第 30 行将 3 加入到 aaa 中。将 3 加入到 Achar 对象中将导致什么后果呢？它不是一个 char 对象，更不是一个数值对象，而是一个 Achar 对象。

该程序清单试图将 3 加入到对象中，而不是对象的成员中。结果将导致错误——正如您从编译器输出中看到的。

第 31 行将一个 Achar 对象减去 1，这也导致错误，因为不能像这样对对象执行加减操作。如果第 30 和 31 行能够运行，则第 33 行将打印对象的值。

可以对对象的数据成员（而不是对象本身）执行加减运算。将第 30 和 31 行修改为如下所示后，程序清单便能通过编译：

```
aaa.ch = (char) (aaa.ch + 1)
bbb.ch = (char) (bbb.ch - 1)
```

虽然这样做管用，但并不是一个根本的解决方案。其中的强制转换过多，代码也不是最简单的。另一种解决方案是，在类中添加一个方法，让最终用户可以对类对象执行加减或其他类型的运算。程序清单 20.2 提供了这种解决方案。

程序清单 20.2 over1b.cs: 数学运算符

```
1: // over1b.cs - Using methods for mathematic operations
2: //-----
3:
4: using System;
5:
6: public class AChar
7: {
8:     private char CH;
9:
10:    public AChar() { this.CH = ' '; }
11:    public AChar(char val) { this.CH = val; }
12:
13:    public char ch
14:    {
15:        get{ return this.CH; }
16:        set{ this.CH = value; }
17:    }
18:
19:    static public AChar Add ( AChar orig, int val )
20:    {
21:        AChar result = new AChar();
22:        result.ch = (char) (orig.ch + val);
```

```

23:     return result;
24: }
25: static public AChar Subtract ( AChar orig, int val )
26: {
27:     AChar result = new AChar();
28:     result.ch = (char) (orig.ch - val);
29:     return result;
30: }
31: }
32:
33: public class myAppClass
34: {
35:     public static void Main(String[] args)
36:     {
37:         AChar aaa = new AChar('a');
38:         AChar bbb = new AChar('b');
39:
40:         Console.WriteLine("Original value: {0}, {1}", aaa.ch, bbb.ch);
41:
42:         aaa = AChar.Add( aaa, 3 );
43:         bbb = AChar.Subtract( bbb, 1 );
44:
45:         Console.WriteLine("Final values: {0}, {1}", aaa.ch, bbb.ch);
46:     }
47: }

```

该程序清单的输出如下：

```

Original value: a, b
Final values: d, a

```

分析：该程序清单比前一个要好——它能够通过编译！它还使得能够对类对象执行数学运算，这是通过使用静态方法Add和Subtract实现的，这两个方法分别是在第19~24行和第25~30行声明的。

Add方法将原来的AChar字符值(ch)增加指定的值。在myAppClass类中，调用AChar.Add方法将aaa加3，因此原来的‘a’变成了‘d’。Add方法返回一个新的AChar对象，可用于覆盖原来的值。这样便可以将数字加入到类对象中，并使用得到的结果覆盖原来的值。Subtract方法的工作原理与此相同，只是将ch减去指定的值。

该程序清单相对简单。如果类中还包含其他的数据成员，则方法Add和Subtract将更为复杂，且更有价值。请看下面的几个例子：

1. Deposit类包含用于个人储蓄的数据成员——账号和储蓄金额。在这种情况下，Add方法只需操纵储蓄金额。
2. currency类包含一个枚举值和多个数字值，前者用于指示币种，后者用于存储不同币种（如美元）的金额。
3. salary类包含雇员姓名、雇员ID号、进入公司的日期以及雇员的薪水。

20.2.1 重载运算符

使用程序清单 20.2 中的方法来增减类的值，是一种完全可以接受的方式。但有时候，重载运

算符可使类更容易使用。前面的三个范例就属于这样的情况，字符串合并范例也是如此。

Achar 类可能不适合重载运算符。为什么？简单地说，如果您重载运算符，则对于使用这个类的每个人而言，被重载的运算符的功能都应该是显而易见的，请看下面的代码行，您认为这得到什么样的结果？其他人将认为是什么结果？

```
Salary = Salary + 1000;
MyChar = MyChar + 3;
MyChar = MyCahr + 'a';
Deposit = Deposit - 300;
```

第 1 和 4 条语句的结果显而易见，第 2 条语句好像比较明显，但结果是什么呢？而 MyChar + 'a' 则更晦涩。对于上述情况，都可以重载+运算符，使上述代码能够运行，但对于 MyChar，使用一个其名称的含义较为明显的方法可能更合适。

可被重载的运算符很多，包括基本的双目数学运算符以及大部分单目运算符、关系运算符和逻辑运算符。

20.2.2 重载基本的双目数学运算符

双目运算符指的是使用两个值的运算符，包括加(+)、减(-)、乘(*)、除(/)和求模(%)。对于这些运算符，都可以在类中进行重载。下面列出了所有可被重载的双目运算符：

- +;
- -;
- *;
- /;
- %;
- &;
- |;
- ^;
- <<;
- >>。

重载双目运算符格式与创建方法类似。重载运算符的通用格式如下：

```
public static return_type operator op(type x, type y)
{
    ...
    return return_type;
}
```

其中 return_type 是被重载的运算符返回的数据类型，对于前面范例中的 Achar 类，为 Achar。返回类型的前面是限定符 public 和 static。重载的运算符必须是公有的，以便能够使用它；也必须是静态的，以便能够通过类而不是各个对象来访问它。

接下来的 operator 用于指出这是一个重载运算符的方法。然后指出要重载的运算符(op)，如果要重载的是加法运算符，则 op 为+。最后，提供用于运算的参数。

这里要重载的是双目运算符，因此有两个参数。其中的一个参数的类型必须与运算符要被重载的类相同；而另一个参数可以是任何类型。重载运算符时，通常将这两个参数的类型设置为相同，

使两个参数的类型不同也是完全可行的。事实上，在重载运算符时，应为各种可能加入到原有类中的数据类型建立重载方法。

回过头来看一下 Achar 类，下面是重载加法运算符的方法头，这样便可以将整数值加入到 Achar 中：

```
. public static Achar operator+ (Achar x, int y)
```

虽然这里使用的参数名是 x 和 y，但您可以根据自己的喜好使用任何名称。之所以将第二个参数设置为整型，是因为前面的程序清单中将这样的值加入到 Achar 对象中。程序清单 20.3 再次使用了 Achar 类，不过这里重载了加法和减法运算符。

注意：您可能注意到了，前面介绍的格式中，operator 和 op 之间有一个空格，而刚才关于 Achar 类的范例中，并没有空格，运算符与 operator 连接在一起。这两种格式都有效。

程序清单 20.3 overlc.cs: 重载双目运算符

```
1: // overlc.cs - Overloading an operator
2: //-----
3:
4: using System;
5:
6: public class AChar
7: {
8:     private char CH;
9:
10:    public AChar() { this.CH = ' '; }
11:    public AChar(char val) { this.CH = val; }
12:
13:    public char ch
14:    {
15:        get{ return this.CH; }
16:        set{ this.CH = value; }
17:    }
18:
19:    static public AChar operator+ ( AChar orig, int val )
20:    {
21:        AChar result = new AChar();
22:        result.ch = (char)(orig.ch + val);
23:        return result;
24:    }
25:    static public AChar operator- ( AChar orig, int val )
26:    {
27:        AChar result = new AChar();
28:        result.ch = (char)(orig.ch - val);
29:        return result;
30:    }
31: }
32:
33: public class myAppClass
34: {
35:     public static void Main(String[] args)
```



```

36:    {
37:        AChar aaa = new AChar('a');
38:        AChar bbb = new AChar('b');
39:
40:        Console.WriteLine("Original value: {0}, {1}", aaa.ch, bbb.ch);
41:
42:        aaa = aaa + 25;
43:        bbb = bbb - 1;
44:
45:        Console.WriteLine("Final values: {0}, {1}", aaa.ch, bbb.ch);
46:    }
47: }

```

该程序清单输出如下:

```

Original value: a, b
Final values: z, a

```

分析: 第19~30行对AChar类的加法和减法运算符进行重载。第19行的重载格式与前面介绍的相同, 返回类型为AChar, 第一个参数的类型也是AChar。

这里将一个整型值与AChar对象相加, 您可以使用另一个AChar对象或者其他任何类型的值, 而不是整型值。事实上, 可以重载加法运算符多次, 每次将不同的数据类型值加入到AChar对象中, 只要重载后的方法有不同的特征标即可。

第21~23行实现了重载后的加法运算符的功能。第21行实例化一个新的AChar对象, 将其数据成员ch的值设置为加法运算的结果, 然后将这个新的AChar对象(result)返回。可以按您的喜好将该方法内的代码修改为任何代码, 但它们应该与加法运算符接受的值相关。

重载减法运算符的方式与此相同。今天的一个练习将要求您为减法运算符创建另一个重载方法, 该方法接受两个AChar参数, 并返回它们之间的字符数。

该程序清单只重载了加法运算符和减法运算符, 重载乘法、除法、求模和其他双目运算符的方式与此相同。

20.2.3 重载基本的单目数学运算符

单目运算符只使用操作数, 可以被重载的单目运算符如下:

- +;
- -;
- ++;
- --;
- !;
- ~;
- true;
- false。

重载单目运算符的方式与双目运算符类似, 不同的是只需要声明一个参数, 这个参数的类型与重载运算符的类相同。只需要传递一个参数, 因为单目运算符对一个值执行运算。程序清单 20.4 和 20.5 提供了两个重载单目运算符的例子。程序清单 20.4 重载了单目运算符+和-, 这里使用的是AChar

类，其中运算符+被重载为将 Achar 中的字符转换为大写，运算符-被重载为将 Achar 中的字符转换为小写。

程序清单 20.5 重载了递增运算符 (++) 和递减运算符 (--)，它们分别被重载为将字符转换为前一个字符和后一个字符。注意，对 'z' 执行递增运算或对 'A' 执行递减运算得到的将是非字母字符。但您可以添加这样的逻辑，即防止递增或递减运算跨越第一个字母或最后一个字母。

程序清单 20.4 over2.cs: 重载单目运算符+和-

```

1: // over2b.cs - Overloading
2: //-----
3:
4: using System;
5: using System.Text;
6:
7: public class AChar
8: {
9:     private char CH;
10:
11:     public AChar() { this.CH = ' '; }
12:     public AChar(char val) { this.CH = val; }
13:
14:     public char ch
15:     {
16:         get{ return this.CH; }
17:         set{ this.CH = value; }
18:     }
19:
20:     static public AChar operator+ ( AChar orig )
21:     {
22:         AChar result = new AChar();
23:         if( orig.ch >= 'a' && orig.ch <='z' )
24:             result.ch = (char) (orig.ch - 32 );
25:         else
26:             result.ch = orig.ch;
27:
28:         return result;
29:     }
30:     static public AChar operator- ( AChar orig )
31:     {
32:         AChar result = new AChar();
33:         if( orig.ch >= 'A' && orig.ch <='Z' )
34:             result.ch = (char) (orig.ch + 32 );
35:         else
36:             result.ch = orig.ch;
37:
38:         return result;
39:     }
40:
41: }
```

```
42:
43: public class myAppClass
44: {
45:     public static void Main(String[] args)
46:     {
47:         AChar aaa = new AChar('g');
48:         AChar bbb = new AChar('g');
49:         AChar ccc = new AChar('G');
50:         AChar ddd = new AChar('G');
51:
52:         Console.WriteLine("ORIGINAL:");
53:         Console.WriteLine("aaa value: {0}", aaa.ch);
54:         Console.WriteLine("bbb value: {0}", bbb.ch);
55:         Console.WriteLine("ccc value: {0}", ccc.ch);
56:         Console.WriteLine("ddd value: {0}", ddd.ch);
57:
58:         aaa = +aaa;
59:         bbb = -bbb;
60:         ccc = +ccc;
61:         ddd = -ddd;
62:
63:         Console.WriteLine("\n\nFINAL:");
64:         Console.WriteLine("aaa value: {0}", aaa.ch);
65:         Console.WriteLine("bbb value: {0}", bbb.ch);
66:         Console.WriteLine("ccc value: {0}", ccc.ch);
67:         Console.WriteLine("ddd value: {0}", ddd.ch);
68:     }
69: }
```

该程序清单的输出如下:

```
ORIGINAL:
aaa value: g
bbb value: g
ccc value: G
ddd value: G
```

```
FIANL:
aaa value: G
bbb value: G
ccc value: g
ddd value: g
```

分析: 从输出可以知道, 运算符+将小写字母转换为大写, 如果字母已经是大写的, 则不变; 运算符-的功能则相反, 即将大写字母转换为小写, 如果字母已经是小写的, 则不变。

第 20~39 行包含被重载的运算符方法, 这些方法是单目运算符重载方法, 因为它们只接受一个参数 (参见第 20 和 30 行)。这些重载运算符方法中的代码简单易懂, 它们分别检查原来的字符是否大写字母 (第 33 行) 和小写字母 (第 24 行), 如果是大写字母, 则将其加上 32, 转换为小写字母

母 (如果是小写字母, 将其减去 32, 转换为大写字母)。

注意: 字符是以数值的方式存储的。例如, 字母'A'被存储为65, 字母'a'被存储为97。大小写相同的所有字母被存储为的数值依次递增。

程序清单 20.4 重载的是单目运算符+和-, 而程序清单 20.5 重载的是运算符++和--。

程序清单 20.5 over2b.cs: 重载运算符++和--

```

1: // over2b.cs - Overloading
2: //-----
3:
4: using System;
5:
6: public class AChar
7: {
8:     private char CH;
9:
10:    public AChar() { this.CH = ' '; }
11:    public AChar(char val) { this.CH = val; }
12:
13:    public char ch
14:    {
15:        get{ return this.CH; }
16:        set{ this.CH = value; }
17:    }
18:
19:    static public AChar operator++ ( AChar orig )
20:    {
21:        AChar result = new AChar();
22:        result.ch = (char)(orig.ch + 1);
23:        return result;
24:    }
25:    static public AChar operator-- ( AChar orig )
26:    {
27:        AChar result = new AChar();
28:        result.ch = (char)(orig.ch - 1);
29:        return result;
30:    }
31:
32: }
33:
34: public class MyAppClass
35: {
36:     public static void Main(String[] args)
37:     {
38:         AChar aaa = new AChar('g');
39:         AChar bbb = new AChar('g');
40:
41:         Console.WriteLine("Original value: {0}, {1}", aaa.ch, bbb.ch);
42:
43:         aaa = ++aaa;

```

```

44:      bbb = --bbb;
45:
46:      Console.WriteLine("Current values: {0}, {1}", aaa.ch, bbb.ch);
47:
48:      aaa = ++aaa;
49:      bbb = --bbb;
50:
51:      Console.WriteLine("Final values: {0}, {1}", aaa.ch, bbb.ch);
52:
53:  }
54: }

```

该程序清单的输出如下:

```

Original value: g, g
Current values: h, f
Final values: i, e

```

分析: 该程序清单与前一个类似, 不过重载的是运算符`--`和`++`, 而不是`-`和`+`。被重载后, 这些运算符能够对类对象执行运算, 如第43行、44行、48行和49行所示。重载其他单目运算符的方式与此类似。

20.2.4 重载关系运算符

关系运算符也可以被重载, 可被重载的关系运算符如下:

- `<`;
- `<=`;
- `>`;
- `>=`。

下述逻辑运算符也可以被重载:

- `==`;
- `!=`。

重载上述运算符的不同之处在于声明, 它们的返回值不是类对象, 而是一个布尔值。这是有道理的, 因为这些运算符比较两个值, 并确定结果是否为 `true`。

程序清单 20.6 使用一个更真实的类 (`Salary`) 演示了如何重载一些关系运算符。该程序清单没有演示如何重载运算符`==`和`!=`, 因为这些运算符的重载方式稍微不同, 下一节将介绍如何重载它们:

程序清单 20.6 over3.cs: 重载关系运算符

```

1: // over3a.cs - Overloading Relational Operators
2: //-----
3:
4: using System;
5: using System.Text;
6:
7: public class Salary
8: {
9:     private int AMT;
10:

```

```
11: public Salary() { this.amount = 0; }
12: public Salary(int val) { this.amount = val; }
13:
14: public int amount
15: {
16:     get { return this.AMT; }
17:     set { this.AMT = value; }
18: }
19:
20: static public bool operator < ( Salary first, Salary second )
21: {
22:     bool retval;
23:
24:     if ( first.amount < second.amount )
25:         retval = true;
26:     else
27:         retval = false;
28:
29:     return retval;
30: }
31:
32: static public bool operator <= ( Salary first, Salary second )
33: {
34:     bool retval;
35:
36:     if ( first.amount <= second.amount )
37:         retval = true;
38:     else
39:         retval = false;
40:
41:     return retval;
42: }
43:
44: static public bool operator > ( Salary first, Salary second )
45: {
46:     bool retval;
47:
48:     if ( first.amount > second.amount )
49:         retval = true;
50:     else
51:         retval = false;
52:
53:     return retval;
54: }
55:
56: static public bool operator >= ( Salary first, Salary second )
57: {
58:     bool retval;
59:
60:     if ( first.amount >= second.amount )
```

```

61:         retval = true;
62:     else
63:         retval = false;
64:
65:     return retval;
66: }
67:
68: public override string ToString()
69: {
70:     return( this.amount.ToString() );
71: }
72: }
73:
74: public class myAppClass
75: {
76:     public static void Main(String[] args)
77:     {
78:         Salary mySalary = new Salary(24000);
79:         Salary yourSalary = new Salary(24000);
80:         Salary PresSalary = new Salary(200000);
81:
82:         Console.WriteLine("Original values: ");
83:         Console.WriteLine("    my salary: {0}", mySalary);
84:         Console.WriteLine("    your salary: {0}", yourSalary);
85:         Console.WriteLine(" a Pres' salary: {0}", PresSalary);
86:         Console.WriteLine("\n-----\n");
87:
88:         if ( mySalary < yourSalary )
89:             Console.WriteLine("My salary less than your salary");
90:         else if ( mySalary > yourSalary )
91:             Console.WriteLine("My salary is greater than your salary");
92:         else
93:             Console.WriteLine("Our Salaries are the same");
94:
95:         if ( mySalary >= PresSalary )
96:             Console.WriteLine("\nI make as much or more than a president.");
97:         else
98:             Console.WriteLine("\nI don't make as much as a president.");
99:     }
100: }

```

该程序清单的输出如下:

```

Original values:
    my salary: 24000
    your salary: 24000
 a Pres' salary: 200000
-----

```

```
Our Salaries are the same
```

```
I don't make as much as a president.
```

分析：该程序清单创建了一个名为Salary的类。Salary类包含雇员的薪水，您还可以使之包含最后一次加薪的日期和金额等，虽然这里没有这样做。不管在这个类中包含什么信息，它提供的基本信息都应该是薪水。

这个例子重载了多个关系运算符，其中每个运算符的重载方式都相同，因此这里只对其中的一个进行说明。

第 20 行重载运算符<，其返回值类型是 bool，因此返回的结果为 true 或 false。该方法接受两个 Salary 对象作为参数，在使用运算符<时，左边的值将作为第一个参数，右边的值将作为第二个参数。

```
first < second
```

使用这两个值，可以做出适合于这种类的决定。在这个例子中，第 24 行检查第一个 Salary 对象的 amount 值是否小于第二个 Salary 对象的 amount 值。如果是，则将返回值设置为 true；否则设置为 false。第 29 行返回被设置的值。

在 myAppClass 类中，使用被重载的关系运算符的方式与将关系运算符用于基本数据类型完全相同。可以对两个 Salary 对象进行比较，如第 88 行、90 行和 95 行所示。

对于该程序清单，需要解释的另一部分与运算符重载无关。第 68~71 行使用关键字 override 覆盖了方法 ToString()。ToString 方法是自动从基类 Object 那里继承而来的，前面关于继承的课程中介绍过，所有的类都是从 Object 派生而来的，因此包含 Object 中的所有方法，其中包括 ToString 方法。

在任何类中，都可以覆盖 ToString 方法，它总是返回类的字符串表示。对于可能包含大量成员的 Salary 类，可以返回大量可能的东西，但返回以字符串表示薪水值最为合理，而第 70 行正是这样做的。

更为重要的是，通过覆盖 ToString 方法，可以打印这个类，如第 83~85 行所示。当您这样显示类时，ToString 方法将自动被调用。

20.2.5 重载逻辑运算符

与关系运算符相比，重载逻辑运算符==和!=的工作量要大些。首先，您不能只重载其中的一个，要重载其中的一个，必须对两个都进行重载。另外，要重载这些运算符，还必须重载两个方法：Equals() 和 GetHashCode()。和 ToString 方法一样，这些方法也包含在 Object 中，您创建的类将自动继承这些方法。之所以需要重载这些方法，是因为逻辑运算符在幕后使用了它们。对两个属于同一类的对象进行比较时，应该定义一个 Equals 方法，以覆盖基类的 Equals 方法。该方法 的格式如下：

```
public override bool Equals( object val)
{
    //determine if classes are equal or not
    //return (either true or false)
}
```

该方法可用于检查两个对象是否相等，您可以在其中执行任何逻辑，包括检查单个值或多个值。例如，如果两个 Salary 对象的 amount 相等，这两个对象就相等吗？如果 Salary 类包含雇佣日期，则

两个 `amount` 相同但雇佣日期不同的 `Salary` 对象相等吗？您必须在 `Equals` 方法中编写逻辑来做出这样的决策。

如果要重载运算符 `==` 和 `!=`，也必须覆盖方法 `GetHashCode`，该方法返回一个用于标识类实例的整型值。通常，不用修改该方法。您可以覆盖该方法，并返回当前实例的散列码，如下所示：

```
public override int GetHashCode()
{
    return this.ToString().GetHashCode();
}
```

覆盖方法 `Equals` 和 `GetHashCode` 后，需要为运算符 `==` 和 `!=` 定义重载方法，其结构与关系运算符相同，不同的是您需要使用 `Equals` 方法，而不是重复编写比较代码。在程序清单 20.7 中，重载运算符时调用 `Equals` 方法，并返回与该方法的结果相反的值。

注意：`Equals` 方法实际上使用 `GetHashCode` 方法返回的值来确定两个对象是否相等。

程序清单 20.7 over4.cs: 重载运算符 `==` 和 `!=`

```
1: // over4.cs - Overloading
2: //-----
3:
4: using System;
5: using System.Text;
6:
7: public class Salary
8: {
9:     private int AMT;
10:
11:     public Salary() { this.amount = 0; }
12:     public Salary(int val) { this.amount = val; }
13:
14:     public int amount
15:     {
16:         get{ return this.AMT; }
17:         set{ this.AMT = value; }
18:     }
19:
20:     public override bool Equals(object val)
21:     {
22:         bool retval;
23:
24:         if( ((Salary)val).amount == this.amount )
25:             retval = true;
26:         else
27:             retval = false;
28:
29:         return retval;
30:     }
31:
32:     public override int GetHashCode()
33:     {
```

```
34:     return this.ToString().GetHashCode();
35: }
36:
37: static public bool operator == ( Salary first, Salary second )
38: {
39:     bool retval;
40:
41:     retval = first.Equals(second);
42:
43:     return retval;
44: }
45: static public bool operator != ( Salary first, Salary second )
46: {
47:     bool retval;
48:
49:     retval = !(first.Equals(second));
50:
51:     return retval;
52: }
53:
54: public override string ToString()
55: {
56:     return( this.amount.ToString() );
57: }
58:
59: }
60:
61: public class MyAppClass
62: {
63:     public static void Main(String[] args)
64:     {
65:         string tmpstring;
66:
67:         Salary mySalary  = new Salary(24000);
68:         Salary yourSalary = new Salary(24000);
69:         Salary PresSalary = new Salary(200000);
70:
71:         Console.WriteLine("Original values: {0}, {1}, {2}",
72:             mySalary, yourSalary, PresSalary);
73:
74:         if (mySalary == yourSalary)
75:             tmpstring = "equals";
76:         else
77:             tmpstring = "does not equal";
78:
79:         Console.WriteLine("\nMy salary {0} your salary", tmpstring);
80:
81:         if (mySalary == PresSalary)
82:             tmpstring = "equals";
83:         else
```

```

84:         tmpstring = "does not equal";
85:
86:         Console.WriteLine("\nMy salary {0} a president\'s salary",
87:                             tmpstring);
88:     }
89: }

```

该程序清单的输出如下:

Original values: 24000, 24000,200000

My salary equals your salary

My salary does not equal a president's salary

分析: 该程序清单中的大部分代码以前已经分析过了。覆盖方法 Equals 位于第 20~30 行, 而覆盖方法 GetHashCode 方法位于第 32~35 行。您可以尝试删除这两个方法之一, 然后重新编译该程序清单, 这将出错。

从第 37 行起, 是重载运算符 `==` 的方法。前面指出过, 应该使用 Equals 方法来比较两个类对象, 而 `==` 的重载方法正是这样做的, 它调用 Equals 方法, 并返回该方法返回的值。第 45~52 行的 `!=` 重载方法执行相同的操作, 只是使用运算符 `!`, 返回相反的值。

myAppClass 类中的 Main 方法像使用其他被重载的运算符那样使用运算符 `==` 和 `!=`。您可以对两个类对象进行比较, 并得到 true 或 false 结果。

警告: 重载逻辑运算符 `==` 和 `!=` 时, 必须同时重载它们, 而不能只重载其中的一个。

20.2.6 运算符重载情况小结

许多运算符都可被重载, 但仅当得到的功能对使用类的人来说显而易见时, 才应重载运算符。如果对此有疑问, 则应该使用常规方法。表 20.1 和 20.2 分别列出了可被重载和不能被重载的运算符。

表 20.1 可被重载的运算符

+	-	++	--	!	-	true	false		
+	-	*	/	%	&		^	<<	>>
<	<=	>	>=	=	!=				

表 20.2 不能被重载的运算符

=	.	?:		&&	new
is	sizeof	typeof	checked	unchecked	

另外, 也不能重载括号 `()` 和任何复合运算符 `+=`、`-=` 等。复合运算符将使用被重载的双目运算符

剩下的唯一一个运算符是方括号 `[]`。本书前面介绍过, 这种运算符通过使用索引器进行重载。

20.3 总 结

今天介绍了本书的最后一个 OOP 主题——如何重载运算符。很多人认为运算符重载很复杂，但实际上它非常简单。今天介绍了如何重载单目运算符、双目运算符、关系运算符和逻辑运算符，最后一节包含两个表，指出哪些运算符可以被重载，哪些不可以。

至此，几乎介绍了 C# 的所有基本知识，其中包括该语言中的所有基本结构及其用法。明天的课程将简要地介绍几个高级主题。虽然前面介绍了创建复杂的 C# 应用程序所需的所有部件，但有几个高级主题还是有必要介绍一下的。

20.4 问与答

问：哪种方式更好：使用诸如 `Add()` 等方法还是重载运算符？

答：这两种方式都可行。许多人在使用诸如 C++ 和 C# 等高级语言时，喜欢重载运算符。只要对两个类对象执行加法或其他运算时，期望的结果明确，就应该考虑重载运算符。这样，可使代码更容易理解。

问：为何诸如 `+=` 等复合运算符不能重载？

答：今天的课程中实际上回答了这个问题。复合运算符实际上等效于 `xxx = xxx op yyy`，因此 `x += 3` 等效于 `x = x + 3`。这意味着可以使用重载后的双目运算符。如果您重载加法运算符，实际上相当于重载了复合加法运算符 (`+=`)。

问：我希望为递增运算符和递减运算符的前缀和后缀版本提供不同的方法，该做什么？

答：C# 不支持这种做法。只能为递增运算符和递减运算符定义一个重载方法。

20.5 作 业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

20.5.1 小测验

1. 在同一个类中，可以对同一个运算符重载多少次？
2. 什么因素决定运算符可被重载的次数？
3. 要重载运算符 `==`，必须重载哪些方法？
4. 下面哪些是使用重载运算符的绝佳范例？
 - a. 将加法运算符重载为将两个字符串对象组合起来。
 - b. 将减法运算符重载为计算两个 `MapLocation` 对象之间的距离。
 - c. 将加法运算符重载为将值加 1，同时将 `++` 运算符重载为将值加 2。
5. 如何重载运算符 `!=`？
6. 如何重载运算符 `[]`？
7. 哪些关系运算符可以被重载？
8. 哪些单目运算符可以被重载？
9. 哪些双目运算符可以被重载？

10. 哪些运算符不能被重载?
11. 重载运算符时, 必须使用哪些限定符?

20.5.2 练习

1. 将加法运算符重载为将两个 XYZ 对象相加时, 对应的方法头是什么?
2. 修改程序清单 20.3, 添加一个减法重载方法。该方法接受两个 Achar 对象, 返回的结果为这两个 Achar 对象存储的字符值之间的数值型差。

3. 下面的代码片段有问题吗? 如果有, 是什么问题? 如果没有, 它有何功能?

```
static public int operator >= ( Salary first, Salary second )
{
    int retval;

    if ( first.amount <= second.amount )
        retval = 1;
    else
        retval = 0;

    return retval;
}
```

4. 修改程序清单 20.7, 添加两个分别将 Salary 对象与 int 值和 long 值进行比较的方法。

第 21 天课程

反射和属性

至此，您已经阅读了介绍 C# 的 20 天课程，您学习了数据类型、类、面向对象编程等方面的知识，还学习了 C# 编程的所有基础知识。您已经能够创建 C# 程序了，包括使用基类库 (BCL) 的程序。这些程序可以是基于 Windows 的、基于 Web 的或控制台中的。今天将介绍 C# 中的两个高级主题，这样您便完整地学习了 C# 的基础知识。今天的课程包含以下内容：

- 反射的概念；
- 使用反射来确定程序的内容；
- 如何使用预定义的属性 (attribute)？
- 创建自定义属性；
- 如何将自定义属性与代码关联起来？
- 编写在运行阶段对属性进行评估的代码。

21.1 反 射

有时候，应该静下心来思考一下生活，具体地说，您可以静下心来反省一下自己。这样做，常常能够发现自己未被认识的一面。

就像您可以反省一样，也可以让 C# 程序对其自身进行反射 (reflect)，可以使用这种反射来了解应用程序。例如，您可以让类对自身进行反射，指出其包含的方法或属性。您将发现，对程序、类或其他东西进行反射，让您能够更充分地利用它们。

获取关于某种类型的信息的关键是使用反射方法。例如，GetMembers 方法可用于获取类型包含的成员。通过给 GetMembers 方法传递一个 Type 对象，可以获取其成员列表。Type 是一种用于存储类型的类型。有了一个 Type 对象后，便可以使用 GetMembers 方法来获知其成员。

反射的第一步是获知类型，获知类 (或其他数据类型) 所属类型的方法是使用静态方法 Type.GetType。该方法的返回值是一个类型，可被赋给 Type 对象。GetType 方法几乎可以使用任何数据类型作为参数。例如，要取得名为 testclass 的类的类型，并将其赋给一个名为 MyTestObject 的 Type 对象，则代码如下：

```
Type MyTypeObject = Type.GetType(testclass);
```

这样 MyTypeObject 将包含 testclass 所属的类型，并可以使用它来获取 testclass 的成员，这是通

过使用 `GetMembers` 方法来实现的。`GetMembers` 方法返回一个由 `MemberInfo` 组成的数组。调用 `MyTypeObject`（它包含 `testclass` 所属的类型）的 `GetMembers` 方法的代码如下：

```
MemberInfo[] Mymemberarray = MyTypeObject.GetMembers();
```

上述代码创建了一个名为 `Mymemberarray` 的 `MemberInfo` 数组，并将调用 `MyTypeObject` 中存储的类型的 `GetMembers` 方法返回的值赋给该数组。

这样，`Mymemberarray` 将包含该类型中的成员，您可以遍历该数据，查看每一个成员。程序清单 21.1 将上面介绍的内容组合在一起，为增加趣味性，这里反射的是 `System.Reflection.PropertyInfo` 类。

`MemberInfo` 类位于名称空间 `Reflection` 中，为避免使用全限定名，需要将 `System.Reflection` 包含进来。

程序清单 21.1 `reflect.cs`: 使用反射

```
1: using System;
2: using System.Reflection;
3:
4: class Mymemberinfo
5: {
6:     public static int Main()
7:     {
8:         //Get the Type and MemberInfo.
9:         string testclass = "System.Reflection.PropertyInfo";
10:
11:         Console.WriteLine ("\nFollowing is the member info for class: {0}",
12:                             testclass);
13:
14:         Type MyType = Type.GetType(testclass);
15:
16:         MemberInfo[] Mymemberinfoarray = MyType.GetMembers();
17:
18:         //Get the MemberType method and display the elements
19:
20:         Console.WriteLine("\nThere are {0} members in {1}",
21:                             Mymemberinfoarray.GetLength(0),
22:                             MyType.FullName);
23:
24:         for ( int counter = 0;
25:             counter < Mymemberinfoarray.GetLength(0);
26:             counter++ )
27:         {
28:             Console.WriteLine( "{0}. {1} Member type - {2}",
29:                                 counter,
30:                                 Mymemberinfoarray[counter].Name,
31:                                 Mymemberinfoarray[counter].MemberType.ToString());
32:         }
33:         return 0;
34:     }
```

```
35: }
```

该程序清单的输出如下:

There are 36 members in System.Reflection.PropertyInfo

```
0. get_CanWrite Member type - Method
1. get_CanRead Member type - Method
2. get_Attributes Member type - Method
3. GetIndexParameters Member type - Method
4. GetSetMethod Member type - Method
5. GetGetMethod Member type - Method
6. GetAccessors Member type - Method
7. SetValue Member type - Method
8. SetValue Member type - Method
9. GetValue Member type - Method
10. GetValue Member type - Method
11. get_PropertyType Member type - Method
12. IsDefined Member type - Method
13. GetCustomAttributes Member type - Method
14. GetCustomAttributes Member type - Method
15. get_ReflectedType Member type - Method
16. get_DeclaringType Member type - Method
17. get_Name Member type - Method
18. get_MemberType Member type - Method
19. GetHashCode Member type - Method
20. Equals Member type - Method
21. ToString Member type - Method
22. GetAccessors Member type - Method
23. GetGetMethod Member type - Method
24. GetSetMethod Member type - Method
25. get_IsSpecialName Member type - Method
26. GetType Member type - Method
27. MemberType Member type - Property
28. PropertyType Member type - Property
29. Attributes Member type - Property
30. IsSpecialName Member type - Property
31. CanRead Member type - Property
32. CanWrite Member type - Property
33. Name Member type - Property
34. DeclaringType Member type - Property
35. ReflectedType Member type - Property
```

介绍代码前,我们先来看一下输出。从输出可以知道, `System.Reflection.PropertyInfo` 类包含 36 个成员。输出中行号为 0 的一行是第一个成员 `get_CanWrite`, 它是一个方法。其他成员分别是 `get_Attributes`、`GetIndexParameters` 等。请看输出中行号为 13 和 14 的内容, 它们看起来是一样的, 都是 `GetCustomAttributes`。出错了么? 不! 因为每个重载方法都是一个独立的成员。

分析: 首先需要注意的是, 第 12 行包含进来的名称空间 `System.Reflection`。由于程序清单中使用了反射成员, 因此这是必需的。

第 9 行将一个特定的类名赋给一个变量, 这使得很容易反射不同的类——只需修改存储在该字

字符串中的名称即可。

可对该程序清单进行改进的是，捕获指示要反射的类的命令行参数，这留给读者来完成！使用命令行参数来指定要反射的类，可避免每当反射不同的类时重新编译程序。这也说明了非常重要的一点，即反射可以在运行阶段进行。

第 14 行将类名传递给 `Type.Get` 方法，并将返回的类型赋给变量 `MyType`。然后第 16 行使用 `MyType` 对象来取得类型的成员，并将它们赋给一个名为 `Mymemberinfoarray` 的 `MemberInfo` 数组。第 24~32 行遍历该数组，并打印每个元素的 `Name` 和 `MemberType` 值。正如您从输出中看到的，`Name` 包含了成员的名称，而作为字符串显示时，`MemberType` 指出了成员的类型。

获取基本信息很容易。如果要获取更具体的信息，则工作量要大些。介绍如何获取具体信息之前，先来看另一个程序清单，它演示的是 `MemberInfo` 对象。程序清单 21.2 再次演示了如何使用反射。

程序清单 21.2 reflect2.cs: 再次使用反射

```

1: // reflect2.cs
2: //-----
3: using System;
4: using System.Reflection;
5:
6: namespace Reflect
7: {
8:
9:     class Mymemberinfo
10:    {
11:        int MYVALUE;
12:
13:        public void THIS_IS_A_METHOD()
14:        {
15:            //
16:        }
17:
18:        public int myValue    // property
19:        {
20:            set { MYVALUE = value; }
21:        }
22:
23:        public static int Main()
24:        {
25:            //The following is the class being checked
26:            string testclass = "Reflect.Mymemberinfo";
27:
28:            Console.WriteLine ("\nFollowing is the member info for class: {0}",
29:                               testclass );
30:
31:            Type MyType = Type.GetType(testclass);
32:
33:            MemberInfo[] Mymemberinfoarray = MyType.GetMembers();
34:
35:            //Get the MemberType method and display the elements

```

```

36:
37:     Console.WriteLine("\nThere are {0} members in {1}",
38:         MyMemberInfoarray.GetLength(0),
39:         MyType.FullName);
40:
41:     for ( int counter = 0;
42:         counter < MyMemberInfoarray.GetLength(0);
43:         counter++ )
44:     {
45:         Console.WriteLine( "{0}. {1} Member type - {2}",
46:             counter,
47:             MyMemberInfoarray[counter].Name,
48:             MyMemberInfoarray[counter].MemberType.ToString());
49:     }
50:     return 0;
51: }
52: }
53: }

```

该程序清单的输出如下：

Following is the member info for class: Reflect.Mymemberinfo

```

There are 9 members in Reflect.Mymemberinfo
0. GetHashCode Member type - Method
1. Equals Member type - Method
2. ToString Member type - Method
3. THIS_IS_A_METHOD Member type - Method
4. set_myValue Member type - Method
5. Main Member type - Method
6. GetType Member type - Method
7. .ctor Member type - Constructor-
8. myValue Member type - Property

```

分析：该程序清单和前一个一样，也使用反射，它们的基本原理是相同的，但该程序清单反射的是自身。更重要的是，该程序清单加入了另外几个成员（包括一个名为myValue的属性），帮助说明使用MemberInfo类型能够反射出哪些东西。

MemberInfo类型让您能够获取一般性信息，您还可以使用许多其他的类型来限制获取的信息。例如，您可以声明一个FieldInfo类型，并获取关于类型中字段（数据成员）的信息。通过使用更有针对性的方法，如FieldInfo，还可以获得更为具体的、关于每个项目的信息。例如，FieldInfo类型让您能够知道诸如字段的访问限定符的信息以及实现细节，还可以让您设置和获取值。表21.1列出了一些可能对反射有帮助的类。

表 21.1 用于发现具体信息的类型

反射类型	描 述
Assembly	适用于组合体

续表

反射类型	描 述
ConstructorInfo	适用于构造函数, 确定构造函数的名称、参数、访问限定符和实现细节等信息。
EventInfo	适用于事件, 确定名称、事件处理信息、自定义属性 (attribute) 等信息。
FieldInfo	适用于字段, 确定关于字段的名称、访问限定符和实现细节等信息。
MethodInfo	适用于方法, 确定方法的名称、返回类型、参数、访问限定符和实现细节等信息。
Module	适用于模块, 确定诸如类等信息。
ParameterInfo	适用于参数, 确定诸如参数的名称、数据类型、类型 (如输入或输出) 以及参数在方法的特征标中的位置等信息。
PropertyInfo	适用于属性 (property), 确定诸如名称、数据类型、声明类型等信息。

21.2 属性 (attribute)

事物随时间的流逝而变化, 就像您现在正在变换主题。过去的几年, 诸如 Basic 和 C 等编程语言也在变化, 这通常是加入当初不知道或没有考虑的新功能。如果语言不容易改编——在不破坏已有程序的情况下, 则容易落后。诸如 C 和 COBOL 语言并不是为面向对象编程设计的。除了事物在不断变化外, 您还可能希望程序能够与其他程序交互。大多数编程语言都无法与其他系统或语言交互。

21.2.1 何为属性

C#的设计者在该语言中包括了让它能够扩展自身的方式, 这种可扩展性是通过使用属性来实现的。

使用属性的重要原因之一是为了将额外的信息与 C#程序中的代码关联起来, 这样便可以在运行阶段获得这些信息。

实际上, 您在本书中已经使用过属性, 只是没有意识到而已。在第 18 天中, 您在 Web 服务要暴露的方法前加上了代码 [WebMethod], 这实际上是将一个属性与方法关联起来。以后执行该程序时, 便可以使用反射来查询这些属性, 以便知道哪些方法可以用作 WebMethods。

.NET 框架包含大量的属性, 它们是在 BCL 中定义的, 这包括用于文档、多线程、Web 服务等方面的类。除了可以使用或扩展这些类外, 您还可以创建自己的自定义属性。下面是 .NET 框架中已有的一些属性:

- CLSCompliant: 指出目标与 CLS 兼容;
- Conditional: 指出方法是否可被调用, 它基于调用代码中定义的一个值;
- Obsolete: 指出某种类型已被废弃;
- WebMethod: 指出方法在 Web 服务中可用。

使用属性通常包括三步。第一步是定义属性。要使用属性, 必须先创建它——虽然 .NET 框架中包含一些预定义的属性。第二步是将属性与代码元素关联起来。第三步是在运行阶段查询属性。如果不通过查询来使用属性, 则拥有它们便毫无意义。

21.2.2 使用属性

基于 `WebMethod` 属性，您可能猜到了，属性放在要关联的代码运算的前面。您还可能猜到了，属性使用方括号括起来指示的。在第 18 天，您采用下面的方式将 `WebMethod` 属性与类中的方法关联起来：

```
[WebMethod]
public static int Add( int x, int y)
{
    return x + y;
}
```

通常将属性与代码元素关联起来的方式与此相同。表 21.2 列出了属性关联到的一些代码元素。

表 21.2 可与属性关联起来的代码元素

元 素	显式说明符
组合体	assembly
事件方法	event
字段	field
方法	method
程序模块	module
参数	param
属性	property
返回值	return
类或结构	type

从表 21.2 可以知道，属性可以被关联到很多不同的元素。请看下面的范例：

```
[MyAttribute]
class MyClass{}
```

属性 `MyAttribute` 位于一个类的前面，因此该属性将被关联到 `MyClass` 类。

现在来看另一个例子：

```
[MyAttribute]
public int MyMethod( ) { }
```

这与 `WebMethod` 属性极其类似，该属性被关联到哪种元素呢？表 21.2 列出了大量的元素。`MyAttribute` 被关联到方法吗？它也可以被关联到返回类型吗？是的，可以。

C# 让您能够明确指定要将属性关联到什么，这是通过使用表 21.2 中的显式说明符实现的。通过在属性前面加上显式说明符，并使用冒号进行分隔，可以消除这种二义性。要将 `MyAttribute` 属性与返回值关联起来，可以使用如下所示的代码：

```
[return:MyAttribute]
public int MyMethod( ) { }
```

要将其与方法关联起来，可以使用下面的代码：

```
[method:MyAttribute]
public int MyMethod( ) { }
```

提示：由于对属性使用显式说明符没有任何坏处，因此应尽可能使用它们。

21.2.3 使用多个属性

可以将多个属性与同一个代码元素关联起来，方法是分别列出每一个属性：

```
[FirstAttr]
[SecondAttr]
class myClass { }
```

虽然这里每个属性各占一行，但也可以将它们放在同一行中。另外，还可以将属性组合为单个声明。在这种情况下，需要用逗号将各个属性分隔：

```
[FirstAttr, SecondAttr]
class myClass { }
```

21.2.4 使用带参数的属性

属性可以带参数，在属性中包含参数旨在提供额外的信息。

用于属性的参数有两类：位置（positional）参数和命名（named）参数，其中位置参数也被称为非命名（unnamed）参数。

新术语：之所以被称为位置参数，是由于其位置非常重要。由于位置参数必须放在指定的位置，因此其名称变得不那么重要了。名称参数的位置则不重要。名称“名称参数”是基于这样的事实，即参数规范中必须包含参数的名称。通过包含名称，您便知道参数是什么。

可以在单个属性中同时定义位置参数和名称参数。因此，正如您猜到的，要包含位置参数，则必须首先声明它们，因为其位置至关重要。请看下面的范例：

```
[CodeStatus("Tested", Coder="Brad")]
class MyClass { }
```

CodeStatus 属性包含两个参数，包含这些参数的方式与方法调用类似，所有的参数都被放在一对括号内，就像方法的参数一样。另外，参数之间用逗号分隔。

看到上述范例后，您应该知道第一个参数是位置参数，包含它的方式与提供数据相同，这里的数据是“Tested”。第二个参数包含一个名称，该名称被设置为等于一个数据值。该名称参数名叫 Coder，与之关联的数据是“Brad”。

注意：更明确地说，位置参数只不过是数据而已，而名称参数则是被设置为数据值的字段名称

21.2.5 定义自己的属性

虽然属性看起来与程序中的 C# 有所不同，实际上它们没有什么不同。属性只不过是一种有特殊用途的类。由于属性是类，因此您可以定义并使用自己的属性。

属性是从 .NET 框架中已有的类 System.Attribute 派生而来的。可以像派生其他类那样派生属性：

```
public myAttribute : System.Attribute
{ }
```

```
...  
}
```

派生新类时，需要定义一个公有的构造函数。构造函数中的参数都将被视为位置参数。然后您需要定义属性中的其他数据成员，其中公有数据成员将被视为名称参数。具体地说，名称参数是类的公有属性（property）或字段。最后，需要包含定义类用法的信息。

21.2.5.1 限制属性

可以对属性进行限制，您可以创建这样的属性，即它只能关联到特定类型的代码或特定的目标。例如，您可以创建只能被关联到构造函数的属性，也可以创建只能被关联到方法或属性（property）的属性。这种限制是通过使用另一个属性 `AttributeUsage` 实现的。

`AttributeUsage` 被关联到您创建的属性类，它接受一个参数，该参数指出属性可被关联到的代码元素，即指出属性的用法。表 21.3 指出了属性可被限定到哪些目标。

警告：不要将表 21.2 和表 21.3 混为一谈，表 21.2 中的值用于关联属性，而表 21.3 中的值用于创建属性。显然，在程序中，它们之间有一定的关系。

表 21.3 `AttributeUsage` 目标

标 记	用 途
All	可用于任何地方
Assembly	可用于组合体
Class	可用于类
Constructor	可用于构造函数
Delegate	可用于代表
Enum	可用于枚举
Event	可用于事件
Field	可用于字段
Interface	可用于接口
Method	可用于方法
Module	可用于模块
Parameter	可用于方法的参数
Property	可用于属性（property）
ReturnValue	可用于方法的返回值
Struct	可用于结构

实际上，可以将您创建的属性关联到多个目标。限制是通过将一个指定特定目标的参数用于属性来实现的，该参数由 `AttributeTargets` 枚举中的值组成，该枚举中的值正是表 21.3 列出的标记。要包含该表中的多个属性限制标记，可以使用运算符 `|`。下面的代码演示了如何使用 `AttributeUsage`

属性将新的属性限制为只能用于结构和类：

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
```

21.2.5.2 定义属性类

定义属性类的方法与常规类类似，毕竟，属性只不过是另一种类——属性类而已。前面介绍了声明属性时使用的类头。除了类头外，还需要在类体中建立参数和元素。

对于属性类的参数有一定的限制，其类型只能是简单类型（如 bool、byte、char、short、int、long、float 和 double）和 string、System.Type 和 enum。也可以将参数定义为一维数组，只要数组的类型是前面列出的标准类型之一即可。如果属性类被声明为 object，则向其实例化对象传递值时，这个值也必须是前面指出的类型。

程序清单 21.3 使用一个自定义属性来跟踪代码清单的状态——编写者是谁，测试者是谁。

程序清单 21.3 attr1.cs：一个自定义的属性类

```
1: using System;
2:
3: [AttributeUsage(AttributeTargets.All)]
4: public class CodeStatusAttribute : System.Attribute
5: {
6:     private string STATUS;
7:     private string TESTER;
8:     private string CODER;
9:
10:    public CodeStatusAttribute( string Status )
11:    {
12:        this.STATUS = Status;
13:    }
14:
15:    public string Tester
16:    {
17:        set
18:        {
19:            TESTER = value;
20:        }
21:        get
22:        {
23:            return TESTER;
24:        }
25:    }
26:
27:    public string Coder
28:    {
29:        set
30:        {
31:            CODER = value;
32:        }
33:        get
34:        {
```

```

34:         return CODER;
35:     }
36: }
37:
38: public override string ToString()
39: {
40:     return STATUS;
41: }
42:

```

注意：C#允许您通过[xxx()]来使用名为xxxAttribute的属性。

分析：该程序清单创建了一个名为CodeStatusAttribute的属性类。第3行将该类的目标限定为All，这实际上是没做任何限制，即可用于表21.3列出的所有位置。AttributeUsage是一个接受一个位置参数的属性，您之所以知道它是一个属性，是因为它被放在方括号中。

属性类是从第4行开始的，CodeStatusAttribute类是从System.Attribute派生而来的，因此是一个属性类。该类的余下部分是一些标准代码，您应该能够理解。其中包含三个私有变量，这三个变量都是通过属性（property）进行访问的。

还有几点需要注意。在构造函数中定义的参数是位置参数。CodeStatusAttribute属性第一个参数将是Status，同时还有两个名称参数：另外两个公有数据成员——Coder 和 Tester。

21.2.5.3 使用自定义的属性

定义属性后，需要使用它。在程序中使用 CodeStatusAttribute 属性的方法与前面介绍的相同。您需要包含位置参数，还可以包含名称参数。程序清单 21.4 包含多个类，这些类使用 CodeStatusAttribute 属性来指出程序当前的状态。

程序清单 21.4 attrUsed.cs: 将 CodeStatusAttribute 用于类

```

1: // attrUsed.cs - using the CodeStatus attribute
2: //-----
3:
4: [CodeStatus("Beta", Coder="Brad")]
5: public class Circle
6: {
7:     public Circle()
8:     {
9:         // Set up and build a circle class
10:    }
11: }
12:
13: [CodeStatus("Final", Coder="Fred", Tester="John")]
14: public class Square
15: {
16:     public Square()
17:     {
18:         // Set up and build a square class
19:    }
20: }
21:

```



```

22: [CodeStatus("Alpha")]
23: public class Triangle
24: {
25:     public Triangle()
26:     {
27:         // Set up and build a triangle class
28:     }
29: }
30:
31: [CodeStatus("Final", Coder="Bill")]
32: public class Rectangle
33: {
34:     public Rectangle()
35:     {
36:         // Set up and build a rectangle class
37:     }
38: }

```

警告：该程序清单并不完整，因此无法通过编译。

分析：这些类使用了 `CodeStatusAttribute` 属性。您可能会问，第 4、13、22 和 31 行是否有错，这些地方使用的是 `CodeStatus`，而不是 `CodeStatusAttribute`。这不是错误。您可以将 `CodeStatus` 改为 `CodeStatusAttribute`，程序将正常运行，但没有必要这样做。NET 框架允许您定义属性时，名称以 `Attribute` 结尾，而使用属性时，可以省略 `Attribute`。这可以提高程序的可读性，并使用属性定义易于识别。

第 4 行使用了 `CodeStatus` 属性，并将位置参数设置为“Beta”，同时使用了一个名称参数——`Coder`，该名称参数的值被设置为“Brad”。第 13 行还包含了名称参数 `Tester`，而第 21 行包含的参数最少——只有位置参数值。

21.2.6 访问被关联的属性信息

如果不能在运行阶段访问属性信息，则使用属性将毫无意义。可以通过反射来访问属性信息，程序清单 21.5 包含了这样的代码，即可用于查看被关联到类的各种属性。

程序清单 21.5 `reflAttr.cs`：对 `CodeStatus` 属性进行反射

```

1: // reflAttr.cs -
2: //-----
3: class MyApp
4: {
5:     public static void Main()
6:     {
7:         PrintAttributes(typeof(Rectangle));
8:     }
9:
10:    public static void PrintAttributes( Type psdType )
11:    {
12:        Console.WriteLine("\nAttributes for: {0}", psdType.ToString());
13:
14:        Attribute[] attribs = Attribute.GetCustomAttributes(psdType);

```

```

15:         foreach (Attribute attr in attribs)
16:         {
17:             CodeStatus item = (CodeStatus) attr;
18:             Console.WriteLine(
19:                 "Status is {0}. Coder is {1}. Tester is {2}.",
20:                 item.ToString(), item.Coder, item.Tester);
21:         }
22:     }
23: }

```

分析：这些代码让您能够知道与类相关联的属性，该程序清单的主要部分是第10~22行的 PrintAttributes 方法。该方法接受一个 Type 对象，然后打印与该 Type 相关联的属性。Main 方法演示了如何调用 PrintAttributes 方法，并传递 Rectangle 类。第7天介绍过，类本身就是一种类型，这意味着 Rectangle 对象的类型为 Rectangle。由于不能传递对象名，因此这里使用 .NET 框架中的方法 typeof 将类名转换为一个 Type 对象。

PrintAttributes 方法首先打印传递给该方法的类型名变量 psdType，对于 Rectangle 类，为 Rectangle。

第14行创建了一个名为 attribs 的 Attribute 数组，并将传递给方法的类型的属性值赋给它。这是通过使用 Attribute 类中的 GetCustomAttributes 方法实现的，该方法返回与参数相关联的各个属性。PsdType 变量包含的是 Rectangle 类型，与该类型关联的属性有一个（见程序清单 21.4 中的第31行）。如果还有其他的属性，则这些属性也将被赋给该数组。

第15~20行使用一条 foreach 语句来遍历 attribs 数组中的各个 Attribute 值。在第15行的 foreach 语句中，定义了一个 Attribute 变量 attr，该变量的值将被设置为 attribs 数组的当前元素。对于数组中的每个 Attribute 值，都打印其可能的三个参数。这是通过首先将 attr 的值强制转换为 CodeStatus（第17行）实现的。如果您无法理解第15行，则应该复习第11和13天的课程。将 attr 强制转换为 CodeStatus 后，便适应方法、属性和字段，就像它是一个 CodeStatus 一样。

21.2.7 将各部分组合起来

至此，介绍了关于创建属性、将其与类关联起来以及在运行阶段取得信息的各方面知识。程序清单 21.6 将这些内容组合成一个可被编译并执行的程序。该程序清单由前面的三个程序清单组成，仅此而已。

程序清单 21.6 complete.cs：使用自定义属性

```

1: // complete.cs -
2: //-----
3: using System;
4:
5: [AttributeUsage(AttributeTargets.All)]
6: public class CodeStatusAttribute : System.Attribute
7: {
8:     private string STATUS;
9:     private string TESTER;
10:    private string CODER;
11:
12:    public CodeStatusAttribute( string Status )

```

```
13:     {
14:         this.STATUS = Status;
15:     }
16:
17:     public string Tester
18:     {
19:         set
20:         {
21:             TESTER = value;
22:         }
23:         get
24:         {
25:             return TESTER;
26:         }
27:     }
28:
29:     public string Coder
30:     {
31:         set
32:         {
33:             CODER = value;
34:         }
35:         get
36:         {
37:             return CODER;
38:         }
39:     }
40:
41:     public override string ToString()
42:     {
43:         return STATUS;
44:     }
45: }
46:
47: // attrUsed.cs - using the CodeStatus attribute
48: //-----
49:
50: [CodeStatus("Beta", Coder="Brad")]
51: public class Circle
52: {
53:     public Circle()
54:     {
55:         // Set up and build a circle class
56:     }
57: }
58:
59: [CodeStatus("Final", Coder="Fred", Tester="John")]
60: public class Square
61: {
62:     public Square()
```

```
63:     {
64:         // Set up and build a square class
65:     }
66: }
67:
68: [CodeStatus("Alpha")]
69: public class Triangle
70: {
71:     public Triangle()
72:     {
73:         // Set up and build a triangle class
74:     }
75: }
76:
77: [CodeStatus("Final", Coder="Bill")]
78: public class Rectangle
79: {
80:     public Rectangle()
81:     {
82:         // Set up and build a rectangle class
83:     }
84: }
85:
86: class myApp
87: {
88:     public static void Main()
89:     {
90:         PrintAttributes(typeof(Circle));
91:         PrintAttributes(typeof(Triangle));
92:         PrintAttributes(typeof(Square));
93:         PrintAttributes(typeof(Rectangle));
94:     }
95:
96:     public static void PrintAttributes( Type psdType )
97:     {
98:         Console.WriteLine("\nAttributes for: {0}", psdType.ToString());
99:
100:         Attribute[] attribs = Attribute.GetCustomAttributes(psdType);
101:         foreach (Attribute attr in attribs)
102:         {
103:             CodeStatusAttribute item = (CodeStatusAttribute) attr;
104:             Console.WriteLine(
105:                 "Status is {0}. Coder is {1}. Tester is {2}.",
106:                 item.ToString(), item.Coder, item.Tester);
107:         }
108:     }
109: }
```

该程序清单的输出如下:

```

Attributes for: Circle
Status is Beta. Coder is Brad. Tester is.

Attributes for: Triangle
Status is Alpha. Coder is . Tester is.

Attributes for: Square
Status is Final. Coder is Fred. Tester is John.

Attributes for: Rectangle
Status is Final. Coder is Bill. Tester is.

```

分析: 该程序清单首先创建了自定义属性 `CodeStatusAttribute`, 然后通过其简写名称 (`CodeStatus`) 将其用于多个类, 最后 `myApp` 类检查了各个类的属性。

第 90 和 91 行是新加的, 在前一个程序清单中, 只检查了 `Rectangle` 类的属性。该程序清单中, 对于每个类, 都调用了 `PrintAttributes` 方法。从输出可以知道, 每种类的属性都被正确打印。

注意: 虽然这里将所有的代码都放在同一个程序清单中, 但也可以将自定义属性放在一个独立的文件和/或名称空间中。然后在程序清单中使用 `using` 语句将其包含进来。

21.2.8 单次使用的属性和多次使用的属性

对于属性, 值得注意的另一点是, 试图多次将 `CodeStatus` 与同一个类关联起来, 将出错。例如, 下面的代码将出错:

```

[CodeStatus("Beta", Coder="Brad")]
[CodeStatus("Testing", Tester="Bill")]
class Rectangle{
...

```

但如果将属性修改为指出类的编写者, 情况将如何呢? 这个属性可以以位置参数的方式来包含编写者的姓名, 而其他名称参数可以包含诸如最后一个修改日期或状态等信息。这样, 对于同一个类, 可以有多个编写者。

将多个属性关联到同一个东西很容易。只需在声明属性时允许多次关联即可。在前面声明属性时, 您将下述信息作为属性声明中的一个属性:

```
[AttributeUsage(AttributeTargets)]
```

其中 `AttributeTargets` 是一个位置参数, 它规定了属性的合法目标。您也可以包含名称参数 `AllowMultiple`。如果将该参数设置为 `true`, 则可以将同一个属性多次用于同一个目标。虽然 `AllowMultiple` 的默认值为 `false`, 但也可以将其设置为 `false`。

要允许将 `CodeStatus` 属性多次用于同一个目标, 只需修改程序清单 `complete.cs` 中的一行代码, 即将第 5 行修改为如下所示即可:

```
[AttributeUsage(AttributeTargets.All, AllowMultiple=true)]
```

这样, 便可以在该程序清单中将 `CodeStatus` 属性多次用于同一个目标。

21.3 总 结

今天是最后一个课程，介绍了 C# 编程中的两个高级主题，它们让您能够在运行阶段获取编程方面的技术性信息。今天首先介绍了反射。通过反射可以了解程序中哪些方法、属性、事件和其他成员是可用的。

然后介绍了属性 (attribute)，它使得 C# 语言能以结构化的方式进行扩展。另外，属性让您能够将额外的信息与程序的某部分关联起来。您学习了如何创建自定义属性、如何将它们与类关联起来以及如何在运行阶段查询程序中关于属性的信息。

祝贺您完成了对 C# 的 21 天学习。仅在 21 个课程中，您便学习了大量的知识。需要学习的知识还很多，就像今天的主题一样，您接下来的学习将是扩展已学会的知识。属性只是另一种类，而反射使用基类库中的类和信息。理解本书介绍的大部分知识后，您便几乎能够应付任何基本的 C# 项目。成为专家（高手）的最佳方式是应用学到的知识——编写程序。您编写的程序越多，则从只懂 C# 皮毛到完全意义上的专家的时间就越短。

21.4 问与答

问：属性只能放在与之关联的元素之前，如果要将属性与组合体关联起来，应将其放在什么地方？

答：与组合体和模块关联的属性被放置在导入名称空间的语句以后，并位于代码之前。对于组合体属性，只能放在这里。

问：可将反射用于属性吗？

答：反射用于确定属性值。

21.5 作 业

下面的小测验帮助您巩固所学的知识，练习则让您实际应用所学的知识。在阅读下一课时之前，应尽可能理解这些小测验和练习的答案，答案见附录 A。

21.5.1 小测验

1. 可以使用什么来获知对象、类或其他东西的类型？
2. 哪种类型可用于存储类型值？它位于哪个名称空间中？
3. 哪种概念在运行阶段提供关于类的信息？
4. 哪种类型可用于获取关于方法参数的详细信息？
5. C# 包含了什么来帮助以后扩展该语言或帮助该语句来处理当前还没有出现的概念？
6. 第 16 天使用的 WebMethod 标记是一个反射的例子还是属性的例子？
7. 指出三种预定义的属性。
8. 属性可以与程序中的哪五样东西关联起来？
9. 用于属性中的参数有哪两类？它们之间的区别何在？
10. 如何限制属性可被关联到的目标？
11. 属性参数可以是哪些数据类型？

21.5.2 练习

1. 修改 `Refect.cs` (程序清单 21.1), 对 `Object` 类 (`System.Object`) 进行反射?
2. `Object` 类中包含哪些方法? 今天练习中反射过的类型中包含 `Object` 中的哪些方法?
3. 修改程序清单 22.2, 使用 `FieldInfo` 类型, 而不是 `MemberInfo`, 以查看程序清单中的字段值
4. 修改程序清单 `complete.cs`, 允许将同一个属性多次关联到同一个目标, 并将 `CodeStatus` 属性两次关联到同一个类。

第3周复习

祝贺您阅读完了本书。您学习了使用 C# 编程的基本知识，另外还学习了如何使用标准类库（包括基类库，BCL）中的一些类和其他类型。

您知道，通过使用已有的类，可以创建基于 Windows、基于 Web 或基于服务的应用程序。另外，您还知道，有大量的类可供使用，使用它们能立刻实现大量的功能

应用所学的知识

确保精通 C# 的最佳途径是应用所学的知识。您应编写尽可能多的 C# 程序。您将发现，您应用 C# 越多，它便变得越容易。

展示您的程序

随着不断使用 C#，您很可能创建出许多您为之骄傲或您认为对其他人很有帮助的程序或类。如果是这样，请将它发送给我，并附上详细描述其代码和特别之处的文本。如果时间允许，我将把它们张贴到网站 www.TeachYourselfCSharp.com 上。与此同时，您还可以核查递交的程序清单。至于如何递交，请参阅该网站。

附录 A

作业答案

第 1 天课程的答案

小测验

1. 指出 C#是一种非常棒的编程语言的三个理由。

- C#简单;
- C#现代;
- C#是面向对象的;
- C#功能强大而灵活;
- C#使用的单词不多;
- C#是模块化的;
- C#将流行。

2. IL 和 CLR 代表的分别是什么?

IL 代表中间语言 (Intermediate Language); CLR 代表通用语言运行阶段环境 (Common Language Runtime)。

3. 程序开发周期包括哪些步骤?

- a. 创建源代码;
- b. 编译程序;
- c. 执行程序。

4. 要使用编译器编译程序 my_prog.cs, 应输入什么命令?

```
csc my_prog.cs
```

5. C#源代码文件应使用什么扩展名?

```
.cs;
```

6. filename.txt 是一个合法的 C#源代码文件名吗?

是的, 但不推荐这样做。

7. 执行编译后的程序时, 如果它不按您期望的那样运行, 该如何做?

应核对代码, 确保没有逻辑错误。

8. 机器语言是什么?

计算机能够理解的语言。

9. 下述错误很可能位于哪一行?

```
my_prog.cs(35,6): error CS1010: Newline in constant
```

很可能位于第 35 行, 如果不是第 35 行, 则可能是第 34 行。

10. 下述错误很可能位于哪一行?

```
my_prog.cs(35,6): error CS1010: Newline in constant
```

第 6 列。

练习

1. 程序清单可能晦涩而混乱, 包含一些怪字符。

2. 该程序打印半径为 4 的圆的面积和周长。

```
Radius = 4, PI = 3.1459
The area is 50.3344
The circumference is 25.1672
```

3. 该程序打印一个由 X 组成的长方形。

```
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
```

4. 该程序的第 5 和 6 行缺少引号

```
1: class Hello
2: {
3:     static void Main()
4:     {
5:         System.Console.WriteLine("Keep Looking!");
6:         System.Console.WriteLine("You'll find it!");
7:     }
8: }
```

5. 该程序打印一个由笑脸 (而不是 X) 组成的长方形。

第2天课程的答案

小测验

1. 在 C# 程序中, 可以使用哪三种注释?

单行注释、多行注释和文档注释。

2. 如何将上述三种注释输入到 C# 程序中?

单行注释可以从任何位置开始, 以两个斜杠打头。

多行注释以 /* 打头, */ 结束。

文档注释以三个斜杠打头。

3. 空白对 C# 程序有何影响?

空白使 C# 程序易于阅读。编译时, 空白几乎全都被删除, 因此对程序的执行没有任何影响。被引号括起的文本中的空白不会被删除。

4. as、object、throw 和 catch 是关键字, 其他的不是。

5. 字面值是什么?

字面值指的是硬编码的值, 不能改变。例如, 数字 10 就是 10。用引号括起的文本也是字面值。

6. 语句是由表达式组成的;

7. 空语句是什么?

分号本身。

8. 面向对象编程最重要的概念有哪些?

封装、多态、继承和重用。

9. WriteLine() 和 Write() 之间的区别何在?

前者在新的一行开始打印, 后者在原来行接着打印。

10. 在 WriteLine() 和 Write() 中打印值, 什么被用作占位符?

位于花括号内的数字, 数字从 0 开始。

练习

1. 该程序清单不会发生编译错误。

2. 如果不带任何参数运行该程序, 则输出如下:

Format: ListIT filename

如果运行该程序时, 带一个文件名, 则该文件将被显示在屏幕上, 并加上了行号。例如, 如果带的文件名为 listIt.cs, 则结果如下:

```
1: // ListIT.cs - program to print a listing with line numbers
2: //-----
3:
4: using System;
5: using System.IO;
6:
7: class ListIT
8: {
9:     public static void Main(string[] args)
10:    {
11:        try
```

```

12:    {
13:
14:        int ctr=0;
15:        if (args.Length <= 0 )
16:        {
17:            Console.WriteLine('Format: ListIT filename');
18:            return;
19:        }
20:        else
21:        {
22:            FileStream f = new FileStream(args[0], FileMode.Open);
23:            try
24:            {
25:                StreamReader t = new StreamReader(f);
26:                string line;
27:                while ((line = t.ReadLine()) != null)
28:                {
29:                    ctr++;
30:                    Console.WriteLine('{0}: {1}', ctr, line);
31:                }
32:                f.Close();
33:            }
34:            finally
35:            {
36:                f.Close();
37:            }
38:        }
39:    }
40:    catch (System.IO.FileNotFoundException)
41:    {
42:        Console.WriteLine ('ListIT could not find the file {0}', args[0]);
43:    }
44:
45:    catch (Exception e)
46:    {
47:        Console.WriteLine('Exception: {0}\n\n', e);
48:    }
49:    }
50: }

```

3. 该程序打印数字 1~10，并在前面填充 0，如下所示：

```
001 002 003 004 005 006 007 008 009 010
```

4. 编译时，该程序不会发生错误；但运行时将出错——引发异常。问题出在第 8 行，打印值时，第一个占位符的编号应该是 0，而不是 1。

```
8:        System.Console.WriteLine('\nA fun number is {0}', 123 );
```

5. 下面是两个可能的答案。当然，您需要使用您的姓名替换其中的 your name。

```
System.Console.WriteLine('your name');
```

或

```
System.Console.WriteLine('your name');
```

第3天课程的答案

小测验

1. C#中的按值数据类型有哪些?

int、uint、long、ulong、bool、char、short、ushort、float、double、decimal、sbyte 和 ubyte

2. 带符号变量和无符号变量之间的区别何在?

无符号变量只能存储正值, 而带符号的变量可存储负值。

3. 要存储数值 55, 可使用的最小数据类型是什么?

byte。

4. short 变量能够存储的最大值是多少?

带符号时为 32767, 无符号时为 65535。

5. 字符 B 对应的数值是什么?

66。

6. 一个字节包含多少位?

8 位。

7. 哪些字面值可以被赋给布尔型变量?

true 或 false。

8. 指出三种引用数据类型。

类、字符串、接口、数组和代表。

9. 哪种浮点数据类型的精度最高?

decimal。

10. 在.NET 中, 与 C#中的 int 数据类型对应的是什么数据类型?

System.Int32。

练习

1. 修改程序清单 3.6 的取值范围, 以打印小写字母。

```
1: // Ex03_01.cs
2: //-----
3:
4:
5: using System;
6:
7: class chars
8: {
9:     public static void Main()
10:    {
11:        int ctr;
12:        char ch;
13:
```

```

14:     Console.WriteLine('\nNumber   Value\n');
15:
16:     for( ctr = 97; ctr <= 122; ctr = ctr + 1)
17:     {
18:         ch = (char) ctr;
19:         Console.WriteLine( '{0} is {1}', ctr, ch);
20:     }
21: }
22: }

```

2. 编写一行代码，声明一个名为 `xyz` 的 `float` 变量，并将值 123.456 赋给该变量。

```
Float xyz = 123.456;
```

3. 下述哪些变量名是合法的？

- a) `x` 合法
- b) `PI` 合法
- c) `12months` 非法，不能以数字打头
- d) `sizeof` 非法，不能使用 C# 的关键字
- e) `nine` 合法

4. 下面的程序有问题，请在编辑器中输入该程序，并编译它。哪一行会导致错误消息？

```
BugBuster3_4.cs(13,22): error CS0029: Cannot implicitly convert type 'double' to 'decimal'
```

5. 这是选做题。提示，别忘了加上后缀。

第 4 天课程的答案

小测验

1. 哪个字符用于执行乘法运算？

星号 (*)。

2. `10 % 3` 的结果为多少？

1。

3. `10 + 3 * 2` 的结果是多少？

16，而不是 26。

4. 条件运算符是什么？

条件运算符指的是 `&&` (与) 和 `||` (或)。还有三个按位条件运算符: `&` (与)、`|` (或) 和 `~` (非)。

5. 哪个 C# 关键字可用于改变程序的流程？

`if`。

6. 单目运算符和双目运算符的区别何在？

前者使用一个变量，后者使用两个变量。

7. 显式数据类型转换和隐式数据类型转换之间有何区别？

显式转换要求您添加执行转换的代码，而隐式转换将自动进行。

8. 可以将 `long` 值转换为 `int` 值吗？

可以。但需要确保 long 值位于 int 的范围之内，否则结果将是错误的。另外，这种转换需要显式地进行。

9. 条件运算有哪些可能的结果？

可能的结果为真或假。

10. 位移运算符有何功能？

移位运算符将值的位左移或右移。

练习

1. 17。

2. 4。

3. 下面是一个可能的答案：

```
1: class exercise
2: {
3:     static void Main()
4:     {
5:         int value = 1;
6:
7:         if ( value > 65 )
8:             ;
9:         System.Console.WriteLine("The value is greater than 65!");
10:    }
11: }
12: }
```

4. 下面是其中关键的代码行（var 的数据类型为 char）：

```
if ( var == 't' || var == 'T' )
{
    // do something
}
```

5. 代码如下：

```
MyShort = (short) MyLong;
```

6. 第 7 行不能以分号结束。

7. 代码如下：

```
ShortVal = (short) IntVal;
```

8. 代码如下：

```
LongVal = (long) DecVal;
```

9. 代码如下：

```
charVal = (char) ch;
```

第5天课程的答案

小测验

1. 为重复执行代码行多次，C#提供了哪些命令？

`while`、`do`、`for` 和 `foreach` 都可用于重复执行代码行多次。另外结合使用 `goto` 语句和标签也可完成这样的工作，但不推荐使用 `goto` 语句。

2. `while` 语句中的语句将至少执行多少次？

如果条件为 `false`，则一次也不会执行。

3. `do` 语句中的语句将至少执行多少次？

至少执行一次。

4. `x == 1`。

5. `x++`。

6. 什么语句用于结束 `switch` 语句中的 `case` 表达式？

`break`。

7. 标签中使用的是什么标点？

冒号。

8. 在 `for` 语句中，使用哪个标点符号来分隔表达式？

分号用于将初始化器、条件和递增器分开；逗号用于分隔多个表达式。

9. 嵌套指的是什么？

将一个命令完全放在另一个命令的里面。

10. 什么命令用于跳到下一次循环。

`continue`。

练习

1. 下面是一个可能的答案。这是一个完整的程序，而不仅仅是 `if` 语句。

```
// Ex5-1. Exercise 1 for Day 5
//-----

class taxstatus
{
    public static void Main()
    {
        char file_type = 'm';

        if ( file_type == 's' )
        {
            System.Console.WriteLine("The filer is single");
        }
        else if ( file_type == 'm' )
        {
            System.Console.WriteLine("The filer is married filing at the single rate");
        }
        else if ( file_type == 'j' )
        {
        }
    }
}
```



```

    {
        System.Console.WriteLine("The filer is married filing at the joint rate");
    }
    else
    {
        System.Console.WriteLine("The file type is not valid");
    }
}

```

2. 该 if 语句合法，因为 x 等于 2，但 y 不大于 3，因此 x 的值被设置为 9。

3. 下面的程序清单从 99 数到 1。

注意：Write 方法的功能与 WriteLine 基本相同，唯一的差别是 WriteLine 总是在新的一行开始输出，而 Write 不会，

```

// Ex5-3.cs. Exercise 3 for Day 5
// Count from 1 to 99
//-----

class while99
{
    public static void Main()
    {
        int ctr = 1;

        while ( ctr <= 99 )
        {
            System.Console.Write("{0}", ctr);
            ctr++;
        }
    }
}

```

4. 下面是从 99 数到 1 的 for 循环：

```

// Ex5-4.cs. Exercise 4 for Day 5
// Count from 1 to 99
//-----

class for99
{
    public static void Main()
    {
        int ctr;

        for ( ctr = 1; ctr <= 99; ctr++)
            System.Console.Write("{0} ", ctr);
    }
}

```

5. if 语句的条件后面有一个分号，这将导致 if 体总会执行，因此需要删除该分号。下面是编译

该程序时将发生的错误:

```
Ex5-5.cs(10,26): warning CS0642: Possible mistaken null statement
Ex5-5.cs(14,7): error CS1525: Invalid expression term 'else'
Ex5-5.cs(15,11): error CS1002: ; expected
```

下面是修改后的程序:

```
// Ex5-5fix.cs. Exercise 5 for Day 5 (corrected BUG BUSTER)
//-----

class score
{
    public static void Main()
    {
        int score = 99;

        if ( score == 100 )
        {
            System.Console.WriteLine("You got a perfect score!");
        }
        else
            System.Console.WriteLine("Bummer, you were not perfect!");
    }
}
```

6. 下面是一个可能的答案:

```
// Ex5-6.cs. Exercise 6 for Day 5
//-----

class allinone
{
    public static void Main()
    {
        int ctr;

        // One solution:

        for (ctr = 1; ctr <= 10; System.Console.WriteLine("{0}", ctr++))
            ; // empty statement.

        // Another solution:

        for (ctr = 0; ++ctr <= 10; System.Console.WriteLine("{0}", ctr))
            ; // empty statement.
    }
}
```

7. 下面是一个可能的答案:

```
// Ex5-7.cs. Exercise 7 for Day 5
//-----
```

```

class names
{
    public static void Main()
    {
        string name = "Kalee";

        System.Console.WriteLine("Starting the switch... {0}", name);

        switch (name)
        {
            case "Robert":
                System.Console.WriteLine("Hi Bob!");
                break;
            case "Richard":
                System.Console.WriteLine("Hi Rich!");
                break;
            case "Barbara":
                System.Console.WriteLine("Hi Barb!");
                break;
            case "Kalee":
                System.Console.WriteLine("You Go Girl!");
                break;
            default:
                System.Console.WriteLine("Hi {0}", name);
                break;
        }
        System.Console.WriteLine("The switch statement is now over!");
    }
}

```

8. 下面是一个可能的答案:

```

// ex_roll.cs- Exercise 5.8. Using the switch statement.
//-----

class ex_roll
{
    public static void Main()
    {
        int roll = 0;
        int ctr = 0;
        int range = 6;

        int nbr_1 = 0;    //variables used to hold totals
        int nbr_2 = 0;
        int nbr_3 = 0;
        int nbr_4 = 0;
        int nbr_5 = 0;
        int nbr_6 = 0;
    }
}

```

```
// Create random number variable
System.Random rnd = new System.Random();

for (ctr = 0; ctr < 100; ctr++)
{
    // The next line set the roll to a random number from 1 to the
    // value set in range

    roll = (int) rnd.Next(1, range + 1);

    System.Console.WriteLine("Roll {0} is {1}", ctr+1, roll);

    switch (roll)
    {
        case 1:
            nbr_1++;
            break;
        case 2:
            nbr_2++;
            break;
        case 3:
            nbr_3++;
            break;
        case 4:
            nbr_4++;
            break;
        case 5:
            nbr_5++;
            break;
        case 6:
            nbr_6++;
            break;
        default:
            System.Console.WriteLine("Roll is not 1 through 6");
            break;
    }
}

System.Console.WriteLine("\n\nThe results are:");
System.Console.WriteLine("ones: {0}", nbr_1);
System.Console.WriteLine("twos: {0}", nbr_2);
System.Console.WriteLine("threes: {0}", nbr_3);
System.Console.WriteLine("fours: {0}", nbr_4);
System.Console.WriteLine("fives: {0}", nbr_5);
System.Console.WriteLine("sixes: {0}", nbr_6);
}
```

第6天课程的答案

小测验

1. 面向对象程序的四个特征是什么？

多态、封装、继承和重用。

2. 类中可以存储哪两种重要的东西？

数据和方法。还可以存储类型。

3. 被声明为公有的数据成员和没有被声明为公有的数据成员之间的差别何在？

前者可被应用程序访问，而后者是类所私有的，只有类中的代码才能访问。

4. 给数据成员加上关键字 `static` 有何作用？

导致使用该类创建的所有对象共享一个该数据成员的拷贝。

5. 程序清单 6.2 的应用程序类的名称是什么？

`lineApp`。

6. 哪些命令被用于实现属性？

`set` 和 `get`。

7. `value` 有何用途？

用于 `set` 属性的代码中，包含传递给 `set` 属性的值。

8. `Console` 是类、数据成员、名称空间、例程还是数据类型？

是 `System` 名称空间中的一个类。

9. `System` 是类、数据成员、名称空间、例程还是数据类型？

是一个名称空间。

10. 使用哪个关键字来包含名称空间？

`using`。

练习

1. 创建一个类来保存圆心和半径：

```
class circle
{
    public int x;
    public int y;
    public double radius;
}
```

2. 代码如下：

```
1: // ex0602.cs
2: //-----
3:
4: using System;
5:
6: class circle
7: {
8:     int center_x;
9:     int center_y;
```

```

10:     double Radius;
11:
12:     public int x
13:     {
14:         get { return center_x; }
15:         set { center_x = value; }
16:     }
17:     public int y
18:     {
19:         get { return center_y; }
20:         set { center_y = value; }
21:     }
22:     public double radius
23:     {
24:         get { return Radius; }
25:         set { Radius = value; }
26:     }
27: }
28:
29: class MyApp
30: {
31:     public static void Main()
32:     {
33:         circle myCircle = new circle();
34:
35:         myCircle.x = 10;
36:         myCircle.y = 10;
37:         myCircle.radius = 8;
38:
39:         Console.WriteLine('Center: {0},{1}', myCircle.x, myCircle.y);
40:         Console.WriteLine('Radius: {0}', myCircle.radius);
41:         Console.WriteLine('Circumference: {0}', (double) (2 * 3.14159 * myCircle.radius));
42:     }
43: }

```

其运行结果如下:

```

Center: (10,10)
Radius: 8
Circumference: 50.26544

```

3. 创建一个类, 它存储一个 `int` 数据成员 `MyNumber`, 并给该数据成员创建属性, 当该数据成员被存储时, 将其乘以 100; 当其被读取时, 将其除以 100.

```

class MyNumber
{
    int Nbr;

    public int value
    {
        get

```

```

        }
        return (Nbr / 100);
    }
    set
    {
        Nbr = value * 100;
    }
}
}

```

4. 第 5 行试图包含一个类，而 `using` 只能用于包含名称空间。而 `Console` 是一个类，因此这将导致错误。另外，第 19、20 和 21 行也会出错，因为这些行在使用 `Console` 类中的方法时，没有指定类名 `Console`。更正后的程序如下：

```

1:  // A bug buster program
2:  // fixed
3:  //-----
4:  using System;
5:
6:
7:  class name
8:  {
9:      public string first;
10: }
11:
12: class NameApp
13: {
14:     public static void Main()
15:     {
16:         // Create a name object
17:         name you = new name();
18:
19:         Console.Write("Enter your first name and press enter: ");
20:         you.first = Console.ReadLine();
21:         Console.WriteLine("\nHello {0}!", you.first);
22:     }
23: }

```

5. 编写一个 `die` 类，它存储骰子的面数 (`sides`) 以及当前掷骰子得到的点数 (`value`)。

```

class die
{
    public int sides;
    public int value;
}

```

6. 在程序中使用练习 5 的类声明两个骰子对象，并设置这些对象的 `sides` 数据成员，然后将骰子的点数设置为一个随机值。

```

1:  // dice1.cs - A class with two data members
2:  //-----

```

```
3:
4: using System;
5:
6: class die
7: {
8:     public int sides;
9:     public int value;
10: }
11:
12: class diceApp
13: {
14:     public static void Main()
15:     {
16:         die dice1 = new die();
17:         die dice2 = new die();
18:
19:         dice1.sides = 6;
20:         dice2.sides = 12;
21:
22:         // The next two lines set the roll to a random number
23:         // based on the number of sides.
24:         Random rnd = new Random();
25:         dice1.value = (int) ((rnd.NextDouble() * dice1.sides) + 1);
26:         dice2.value = (int) ((rnd.NextDouble() * dice2.sides) + 1);
27:
28:         Console.WriteLine("dice 1; sides = {0}, value = {1}",
29:                             dice1.sides, dice1.value);
30:         Console.WriteLine("dice 2; sides = {0}, value = {1}",
31:                             dice2.sides, dice2.value);
32:     }
33: }
```

第7天课程的答案

小测验

1. 方法的两个重要组成部分是什么?

方法头和方法体。

2. 函数和方法之间有何区别?

没有任何区别，在 C# 这两个术语可以互换。

3. 方法可以返回多少个值?

方法只能返回一个值，但可以使用 `ref` 或 `out` 变量来修改调用程序中的多个变量。

4. 使用哪个关键字从方法返回一个值?

`return`。

5. 方法可以返回哪些数据类型?

可以返回任何数据类型。另外，也可以给方法传递任何数据类型，其中包含对象。方法也可以

没有返回数据类型。

6. 访问类的成员方法时，使用的什么样的格式？例如，如果使用 `myClass` 类实例化了一个名为 `myObject` 的对象，而该类包含一个名为 `myMethod` 的方法，下面哪种访问该方法的方式是正确的？

a. `myClass.myMethod`：如果方法 `myMethod` 被声明为静态的，则是正确的；否则是错误的。

b. `myObject.myMethod`：这是针对对象实例调用类方法的标准方式。如果该方法被声明为静态的，则这种调用方法是错误的。

c. `myMethod`：仅当该调用与方法位于同一个类中时才是正确的，如果该方法是在其他类中声明的，则这将是错误的。

d. `myClass.myObject.myMethod`：不正确。

7. 按引用传递变量和按值传递变量之间的区别何在？

按值传递发送的是变量的一个副本，这意味着调用方法中原来的变量不会被修改；按引用传递发送的是原来变量的位置，这意味着该变量的值可能被修改。

8. 构造函数在何时被调用？

类构造函数在实例化（创建）每个对象时被调用。如果构造函数被声明为静态的，则在第一个对象被实例化前被调用。

9. 没有参数的析构函数的语法是什么样的？

```
~xyz()
{
    // body
}
```

10. 析构函数在何时被调用？

在最后一个对象不再被使用后，程序结束前被调用。

练习

1. 编写公有方法 `xyz` 的方法头，该方法接受两个参数，不返回任何值。

```
public void xyz()
```

2. 为方法 `myMethod` 编写方法头。该方法接受三个参数。第一个名为 `myVal`，其数据类型为 `double`，并按值传递；第二个是一个输出变量，名为 `myOutput`；第三个是按引用传递的，数据类型为 `int`，名为 `myReference`。另外该方法是公有的，其返回类型为 `byte`。

```
public byte MyMethod(double myVal, out string myOutput, ref int myReference)
```

3. 下面是一种可能的解决方案：

```
1: // ex_circ.cs - A simple circle class
2: //-----
3:
4: public class Circle
5: {
6:     public int x;
7:     public int y;
8:     public int radius;
9:
10:    public double area()
```

```
11:  {
12:      double theArea;
13:      theArea = 3.14159 * radius * radius;
14:      return theArea;
15:  }
16:
17:  public double circumference()
18:  {
19:      double theCirc;
20:      theCirc = 2 * 3.14159 * radius;
21:      return theCirc;
22:  }
23:
24:  public Circle()
25:  {
26:      x = 5;
27:      y = 5;
28:      radius = 1;
29:  }
30:
31:  public void print_circle_info()
32:  {
33:      System.Console.WriteLine("\nCircle: Center = {0},{1}", x, y);
34:      System.Console.WriteLine("      Radius = {0}", radius);
35:      System.Console.WriteLine("      Area   = {0}", area());
36:      System.Console.WriteLine("      Circum = {0}", circumference());
37:  }
38: }
39:
40: class CircleApp
41: {
42:     public static void Main()
43:     {
44:         Circle first = new Circle();
45:         Circle second = new Circle();
46:
47:         first.x = 10;
48:         first.y = 14;
49:         first.radius = 3;
50:
51:         first.print_circle_info();
52:         second.print_circle_info();
53:     }
54: }
```

其輸出如下：

```
Circle: Center = (10,14)
      Radius = 3
      Area   = 28.27431
```

```

Circum = 18.84954
Circle.Center = (5,5)
Radius = 1
Area = 3.14159
Circum = 6.28318

```

4. 下面的代码片段有问题，哪一行将导致错误消息？

该方法的返回类型被声明为 `void`，因此它不返回任何值，所以 `return` 语句将导致错误。修改后的代码如下：

```

public void myMethod()
{
    System.Console.WriteLine("I'm a little teapot short and stout");
    System.Console.WriteLine("Down came the rain and washed the spider
out");
}

```

5. 下面是一个可能答案：

```

1: // ex dice.cs- A class with two data members and member methods
2: //-----
3:
4: class die
5: {
6:     public int sides;
7:     public int value;
8:
9:     // The following declares a data member called rnd that is an
10:    // object of type System.Random.
11:    static System.Random rnd = new System.Random();
12:
13:    public int roll()
14:    {
15:        value = (int) ((rnd.NextDouble() * sides) + 1);
16:        return value;
17:    }
18:
19:    // A constructor to default values in the data members.
20:    public die()
21:    {
22:        sides = 6;
23:        value = 0;
24:    }
25: }
26:
27: class diceclass
28: {
29:     public static void Main()
30:     {
31:         die dice1 = new die();
32:         die dice2 = new die();

```

```

33:
34:     dice1.sides = 6;
35:     dice2.sides = 12;
36:
37:     // The next two lines set the value data member to random number
38:     // based on the number of sides. This is done with the roll()
39:     // method.
40:     dice1.roll();
41:     dice2.roll();
42:
43:     System.Console.WriteLine("dice 1; sides = {0}, value = {1}",
44:                               dice1.sides, dice1.value);
45:     System.Console.WriteLine("dice 2; sides = {0}, value = {1}",
46:                               dice2.sides, dice2.value);
47: }
48: }

```

其输出如下:

```

dice 1; sides = 6, value = 4
dice 2; sides = 12, value = 9

```

第 8 天课程的答案

小测验

1. 值数据类型和引用数据类型的差别何在? 结构属于哪一种?

结构是一种值数据类型。值数据类型存储实际被赋给的值; 引用数据类型存储的是关于值所在位置的信息。

2. 结构和类的区别何在?

主要区别在于, 前者是值数据类型, 后者为引用数据类型。另一个区别是, 前者没有不带参数的默认构造函数, 也没有析构函数。

3. 结构的构造函数和类的构造函数之间有何差别 (或者它们之间有差别吗)?

结构不能有不带参数的默认构造函数。

4. 哪个关键字用于声明枚举?

enum。

5. 在枚举中可以存储哪些数据类型?

byte、sbyte、int、uint、short、ushort、long 和 ulong。

6. 数组第一个元素的索引值是多少?

0。

7. 如果访问数组元素时使用的索引值大于数组的元素数目, 将出现什么情况?

将发生运行错误。

8. 被声明为 myArray[4,3,2] 的数组包含多少个元素, 如果这是一个字符数组, 将占用多少内存?

有 24 个元素, 占用 48 个字节的内存。

9. 如何知道数组的长度?

可以使用数组的数据成员 `Length`，它返回数组中的元素数。

10. 判断正误（如果错误，请指出错在哪里）：`foreach` 的结构与 `for` 语句相同。

`foreach` 语句的结构不同于 `for` 语句。`foreach` 语句声明一个数据元素，用于取得数组中的元素。其格式如下：

```
foreach(datatype varname in arrayName)
```

练习

1. 下面是一个可能的答案：

```
1: // ex_line.cs- A line structure which contains point structures. (ex 8.1)
2: //-----
3:
4: struct point
5: {
6:     private int point_x;
7:     private int point_y;
8:
9:     public int x
10:    {
11:        get { return point_x; }
12:        set { point_x = value; }
13:    }
14:     public int y
15:    {
16:        get { return point_y; }
17:        set { point_y = value; }
18:    }
19: }
20:
21: struct line
22: {
23:     private point line_starting;
24:     private point line_ending;
25:
26:     public point starting
27:     {
28:         get { return line_starting; }
29:         set { line_starting = value; }
30:     }
31:     public point ending
32:     {
33:         get { return line_ending; }
34:         set { line_ending = value; }
35:     }
36: }
37:
38: class lineApp
39: {
40:     public static void Main()
```

```

41:  {
42:      line myLine = new line();
43:
44:      point tmp_point = new point();
45:
46:      tmp_point.x = 1;
47:      tmp_point.y = 4;
48:      myLine.starting = tmp_point;
49:
50:      tmp_point.x = 10;
51:      tmp_point.y = 11;
52:      myLine.ending = tmp_point;
53:
54:      System.Console.WriteLine("Point 1: ({0},{1})",
55:                               myLine.starting.x, myLine.starting.y);
56:      System.Console.WriteLine("Point 2: ({0},{1})",
57:                               myLine.ending.x, myLine.ending.y);
58:  }
59: }

```

2. 这是选做题。

3. 问题出在下面的一行代码:

```
Element *= Element;
```

您在 foreach 中创建的变量是只读的, 因此不能给它赋值。

4. 下面是一个可能的答案:

```

1:  // score.cs- Using an array to find the score for a class (Ex 8.4)
2:  //-----
3:
4:  using System;
5:
6:  public class ClassScore
7:  {
8:      public static void Main()
9:      {
10:         // Set up array to hold 30 scores...
11:         int[] scores = new int[30];
12:         // Create a total variable for calculations...
13:         int ttl = 0;
14:
15:         // Set up random variable...
16:         System.Random rnd = new System.Random();
17:
18:         Console.Write("Initializing scores.");
19:
20:         // set random scores into array...
21:         for( int ctr = 0; ctr < 30; ctr++ )
22:         {
23:             scores[ctr] = (int) ((rnd.NextDouble() * 100) + 1);

```

```

24:         Console.Write(".");
25:     }
26:     Console.WriteLine("...done.\n");
27:
28:     // Display scores to console...
29:     for( int ctr = 0; ctr < 30; ctr++ )
30:     {
31:         Console.WriteLine("Student {0} score: {1}", ctr+1, scores[ctr]);
32:         ttl += scores[ctr];
33:     }
34:     Console.WriteLine("=====");
35:     Console.WriteLine("Average Score = {0}", (ttl/30));
36: }
37: }

```

其运行结果如下:

Initializing scores.....done.

Student 1 score: 30

Student 2 score: 19

Student 3 score: 41

...

Student 28 score: 98

Student 29 score: 74

Student 30 score: 7

=====

Average Score = 54

5. 下面是一个可能的解决方案:

```

1: // score2.cs- Using a array to find the score for a class (Ex 8.5)
2: //-----
3:
4: using System;
5:
6: public class ClassScore
7: {
8:     public static void Main()
9:     {
10:         // Set up array to hold 30 scores...
11:         int [,] scores = new int[15,30];
12:         // Create a total variable for calculations....
13:         int ttl = 0;
14:         int testTotal = 0;
15:
16:         // Set up random variable...
17:         System.Random rnd = new System.Random();
18:
19:         Console.Write("Initializing scores.");
20:

```

```

21:     // set random scores into array...
22:     for ( int test = 0; test < 15; test++ )
23:     {
24:         // Loop through students for each test...
25:         for( int student = 0; student < 30; student++ )
26:         {
27:             scores[test,student] = (int) ((rnd.NextDouble() * 100) + 1);
28:             Console.Write(".");
29:         }
30:     }
31:     Console.WriteLine("...done.\n");
32:
33:     //Loop through each test...
34:     for( int test = 0; test < 15; test++ )
35:     {
36:         testTotal = 0; // set total accumulator to zero for a new test.
37:         for( int student = 0; student < 30; student++ )
38:         {
39:             // Console.WriteLine("Student {0} score: {1}", student+1,
scores[test,student]);
40:             testTotal += scores[test,student];
41:         }
42:         // print out average test score...
43:         Console.WriteLine("Average score for test {0} is {1}", test+1, testTotal/30);
44:         ttl += testTotal; // Add test total to overall total
45:     }
46:     Console.WriteLine("=====");
47:     Console.WriteLine("Average score for all tests = {0}", (ttl/(15*30)));
48: }
49: }

```

其运行结果如下:

```

Initializing
scores.....
.....
.....
.....done.

Average score for test 1 is 52
Average score for test 2 is 48
Average score for test 3 is 52
Average score for test 4 is 55
Average score for test 5 is 49
Average score for test 6 is 42
Average score for test 7 is 46
Average score for test 8 is 46
Average score for test 9 is 31

```



```

Average score for test 10 is 48
Average score for test 11 is 47
Average score for test 12 is 56
Average score for test 13 is 56
Average score for test 14 is 46
Average score for test 15 is 59
=====

```

```

Average score for all tests = 49

```

6. 下面一种可能的解决方案:

```

1: // score3.cs- Using an array to find the score for a class (Ex 8.6)
2: //-----
3:
4: using System;
5:
6: public class ClassScore
7: {
8:     public static void Main()
9:     {
10:         // Set up array to hold 30 scores...
11:         int [,] scores = new int[5,15,30];
12:         // Create a total variable for calculations....
13:         int ttl = 0;
14:         int yearTotal = 0;
15:         int testTotal = 0;
16:
17:         // Set up random variable...
18:         System.Random rnd = new System.Random();
19:
20:         Console.Write("Initializing scores.");
21:
22:         // set random scores into array...
23:         for( int year = 0; year < 5; year++ )
24:         {
25:             // loop through tests for a given year...
26:             for ( int test = 0; test < 15; test++ )
27:             {
28:                 // Loop through students for each test...
29:                 for( int student = 0; student < 30; student++ )
30:                 {
31:                     scores[year,test,student] = (int) ((rnd.NextDouble() * 100) + 1);
32:                     Console.Write(".");
33:                 }
34:             }
35:         }
36:         Console.WriteLine("...done.\n");
37:
38:         for( int year = 0; year < 5; year++ )
39:         {

```

```

40:     yearTotal = 0;
41:     //Loop through each test...
42:     for( int test = 0; test < 15; test++ )
43:     {
44:         testTotal = 0; // set total accumulator to zero for a new test.
45:         for( int student = 0; student < 30; student++ )
46:         {
47:             Console.WriteLine("Student {0} score: {1}",
48:                                 student+1, scores[year,test,student]);
49:             testTotal += scores[year,test,student];
50:         }
51:         // print out average test score...
52:         Console.WriteLine("Average score for test {0} is {1}",
53:                             test+1, testTotal/30);
54:         yearTotal += testTotal; // Add test total to overall total
55:     }
56:     Console.WriteLine("====> YEAR {0} Average: {1} <====",
57:                         year+1, yearTotal/(15*30));
58:     ttl += yearTotal;
59: }
60: Console.WriteLine("=====");
61: Console.WriteLine("Average score for all tests = {0}", {ttl/(5*15*30)});
62:
63: }
64: }

```

第9天课程的答案

小测验

1. 重载函数是封装、继承、多态还是重用的方式之一？

多态。

2. 成员函数可以被重载多少次？

对成员函数的重载次数没有限制，只要每种重载定义有独特的特征标即可。

3. 下面哪些成员可被重载：

- a. 数据成员；
- b. 成员方法；
- c. 构造函数；
- d. 析构函数。

成员函数和构造函数可以被重载。析构函数和数据成员不能被重载。

4. 哪个关键字用于使得方法能够接受不同数目的参数？

params。

5. 如何使得方法可以接受不同数目、不同（甚至未知）数据类型的参数？

可以将参数的数据类型声明为 object，这样便可以接受任何数据类型。

6. 要从命令行接受不同数目的参数，需要包含哪个关键字？

Main 方法并不要求您使用关键字 `params`。要接受命令行参数，只需声明一个数组即可，这通常是一个字符串数组。

7. 类成员的默认作用域是什么？

私有的。

8. 限定符 `public` 和 `private` 之间的差别何在？

后者限制在类外进行访问，而前者允许在类外进行访问。

9. 如何禁止使用类实例化对象？

声明一个私有的构造函数。

10. `using` 关键字有哪两种用途？

用于避免使用全限定名称空间名以及为名称空间和类指定别名。

练习

1. 为名为 `abc` 的公有函数编写方法头，该函数接受两个 `short` 参数，返回值类型为 `byte`。

```
public byte abc( params short [] args)
```

2. 编写一行接受命令行参数的代码。

```
public static void Main( string [] args)
```

3. 如果有一个名为 `aClass` 的类，可以加入什么代码来禁止该类被用于实例化对象？

可以添加一个私有的构造函数：

```
private aClass() { }
```

4. 该程序没有问题。可以为名称空间或类型指定别名，但不能为方法指定别名。

5. 创建一个名称空间，它包含一个类和另一个名称空间，被包含的名称空间也包含一个类。然后创建一个使用这两个类的应用程序类。下面是一个可能的答案：

```
1: // ex0605.cs - Exercise 5 of Day 6
2: //-----
3:
4: using System;
5:
6: namespace Numbers
7: {
8:     public class one
9:     {
10:         public static int value = 1;
11:         private one() {}
12:     }
13:     public class two
14:     {
15:         public static int value = 2;
16:         private two() {}
17:     }
18:
19:     namespace Special
20:     {
```

```
21:     public class PI
22:     {
23:         public static double value = 3.1459;
24:         private PI() {}
25:     }
26: }
27: }
28:
29: class MyApp
30: {
31:     public static void Main()
32:     {
33:         Console.WriteLine("One is {0}", Numbers.one.value);
34:         Console.WriteLine("Two is {0}", Numbers.two.value);
35:         Console.WriteLine("PI is {0}", Numbers.Special.PI.value);
36:     }
37: }
```

其运行结果如下:

```
One is 1
Two is 2
PI is 3.1459
```

6. 下面是一个可能的答案:

```
1: // overload.cs - Overloading functions
2: //-----
3:
4: using System;
5:
6: public class myFunc
7: {
8:
9:     public myFunc()
10:    {
11:        Console.WriteLine("Signature: myFunc()");
12:    }
13:
14:     public myFunc( int x )
15:    {
16:        Console.WriteLine("Signature: myFunc( int )");
17:    }
18:
19:     public myFunc( float x )
20:    {
21:        Console.WriteLine("Signature: myFunc( float )");
22:    }
23:
24:     public myFunc( ref int x )
25:    {
26:        Console.WriteLine("Signature: myFunc( ref int )");
```

```
27:     }
28:
29:     public myFunc ( ref float x)
30:     {
31:         Console.WriteLine("Signature: myFunc( ref float )");
32:     }
33: }
34:
35: class myApp
36: {
37:     public static void Main()
38:     {
39:         int i = 99;
40:         float f = 100.00F;
41:
42:         myFunc first  = new myFunc();
43:         myFunc second = new myFunc( 1 );
44:         myFunc third  = new myFunc( 2.2F);
45:         myFunc fourth = new myFunc( ref i );
46:         myFunc fifth  = new myFunc( ref f );
47:         myFunc sixth  = new myFunc( 123L );
48:         myFunc seventh = new myFunc( (byte) 12 );
49:     }
50: }
```

其运行结果如下:

```
Signature: myFunc()
Signature: myFunc( int )
Signature: myFunc( float )
Signature: myFunc( ref int )
Signature: myFunc( ref float )
Signature: myFunc( float )
Signature: myFunc( int )
```

第 10 天课程的答案

小测验

1. 哪些关键字用于处理异常?

try、catch、finally 和 throw

2. 下面的哪些情况应通过异常处理进行处理, 哪些应通过常规代码进行处理?

a. 用户输入的值不在指定的范围之内;

这最适合使用常规代码进行处理。

b. 无法正确地读取文件。

最适合采用异常处理。

c. 传递给方法的参数包含一个无效的值。

最适合使用程序逻辑进行处理。

d. 传递给方法的参数的类型非法。

最适合采用异常处理。

3. 什么会导致异常?

如果程序没能捕获严重的编程错误, 将导致异常。另外, `throw` 语句也能引发异常。

4. 异常是在什么时候发生的?

a. 编写程序时。

b. 编译时。

c. 运行时。

d. 最终用户发出请求时。

异常是在运行阶段发生的, 但如果编译器知道代码将引发异常 (如溢出), 则将发生编译错误。

编写代码或用户请求时, 不会发生异常。

5. 哪个关键字用于对异常做出反应?

关键字 `catch` 用于对异常做出反应。`try` 语句用于将异常发送给 `catch` 语句。

6. `finally` 代码块何时执行?

`finally` 语句块在 `try-catch` 执行完毕后执行。无论 `try-catch` 是否捕获到错误, `finally` 语句块都将执行。

7. 单个 `try` 语句可以有多少个 `catch` 语句与之对应?

可以有 0 或更多个。对应的 `catch` 语句的个数没有任何限制, 但 `catch` 语句的排列次序应该是最具体的到最抽象的。

8. `catch` 语句的次序有关系吗? 为什么?

是的。`catch` 语句的排列次序应该是最具体的到最抽象的。如果将通用的 `catch` 语句放在最前面, 它将在具体的 `catch` 语句之前捕获异常。通常只有一个 `catch` 子句被执行。

9. 大多数预定义的常用异常都是在哪个名称空间中定义的?

`System` 名称空间中定义了许多预定义的名称空间。

10. `throw` 命令的功能是什么?

`throw` 命令让您能够引发或再次引发异常。

练习

1. 答案如下:

```
try
{
    GradePercentage = MyValue/Total
}
```

2. 在该程序清单中, 一个具体的 `catch` 语句被放在一个更为通用的 `catch` 语句后面。因此该捕获 `IndexOutOfRangeException` 的语句将永远不会执行, 这将导致编译错误。

3. 请参见练习 4 的答案。

4. 一个可能的解决方案如下:

```
1: // throwit2.cs
2: // Throwing your own error.
3: //=====
4: using System;
```

```
5:
6: class NegativeNumberException : Exception
7: {
8:     public NegativeNumberException()
9:     {
10:    }
11:
12:     public NegativeNumberException( string e ) : base
13:     {
14:    }
15:
16:     public NegativeNumberException( string e, Exception inner ) :
17:         base ( e, inner )
18:     {
19:    }
20: }
21:
22: class MyAppClass
23: {
24:     public static void Main()
25:     {
26:         int result;
27:
28:         try
29:         {
30:             result = MyMath.SubtractEm( 10, 2 );
31:             Console.WriteLine( "Result of SubtractEm(10, 2) is {0}", result);
32:
33:             result = MyMath.SubtractEm( 2, 10 );
34:             Console.WriteLine( "Result of SubtractEm(2, 10) is {0}", result);
35:         }
36:
37:         catch (NegativeNumberException msg)
38:         {
39:             Console.WriteLine("Sorry, you can't subtract to a result less than zero!");
40:             Console.WriteLine("\nPassed Error Message: \n{0}", msg);
41:         }
42:
43:         catch (Exception e)
44:         {
45:             Console.WriteLine("Exception caught: {0}", e);
46:         }
47:
48:         Console.WriteLine("At end of program");
49:     }
50: }
51:
52: class MyMath
53: {
54:     static public int SubtractEm(int x, int y)
```

```

55:     {
56:         if( y > x )
57:             throw( new NegativeNumberException() );
58:
59:         return( x - y );
60:     }
61: }

```

第 11 天课程的答案

小测验

1. 在 C# 中，可以使用多少个类来派生出一个新类？

只能从一个类派生而来。

2. 下述哪个术语的含义与基类相同？

- a. 父类。
- b. 派生类。
- c. 子类。

父类和基类的含义相同。父类和派生类的含义不同。

3. 哪种存取限定符用于禁止在类的外面访问数据？哪种存取限定符只允许数据在其所属的类及其派生类中使用？

`private` 用于保护类中的数据，`protected` 限制数据成员只能在基类及其派生的类中使用。

4. 如何覆盖基类的方法？

通过在派生类中使用 `new` 关键字声明一个同名的方法。

5. 可以在基类中使用哪个关键字来确保派生类创建自己的方法版本？

在基类中使用 `abstract` 关键字。

6. 哪个关键字用于禁止类被继承？

`sealed`。

7. 指出两个所有的类都有的方法。

`ToString` 和 `GetType`，这是从基类 `Object` 那里继承来的。

8. 哪个类是终极基类，所有的类都是从它派生而来的？

`Object` 类。

9. 装箱有何功能？

将值类型转换为引用类型（对象）。

10. 关键字 `as` 有何用途？

用于将值转换为某种数据类型。与常规强制转换相比，其优点在于：如果被强制转换的值无法转换，则 `as` 关键字将值设置为 `null`，而不是引发异常。

练习

1. 编写为 `ABC` 类声明构造函数的方法头，它接受两个 `int` 参数 `ARG1` 和 `ARG2`。该构造函数调用基类的构造函数，并将 `ARG2` 传递给它。调用是在方法头中完成的。

```

public ABC( int ARG1, int ARG2 ) : base( ARG2 )
{

```



```
}
```

2. 修改后的类如下:

```
1: sealed class aLetter
2: {
3:     private static char A_ch;
4:
5:     public char ch
6:     {
7:         get { return A_ch; }
8:         set { A_ch = value; }
9:     }
10:
11:     static aLetter()
12:     {
13:         A_ch = 'X';
14:     }
15: }
```

3. 必须把对象 `you` 显式地转换为 `Person` 对象, 方法是将第 9 行修改为下列的两行之一:

```
me = you as Person;
```

或

```
me = (Person) you;
```

4. 这是选做题。

第 12 天课程的答案

小测验

1. 哪个方法可用于将数据类型 `int` 转换为字符串, 并进行格式化?

`ToString` 方法。这是 `Object` 类中的一个方法, 因此所有的类都有这样的方法。

2. `string` 类中的哪个方法可用于将信息格式化为一个新的字符串?

`String` 类有一个名为 `Format` 的静态方法, 可用于格式化信息。

3. 哪个说明符可用于指示将数字格式化为金额?

`C` 或 `c`。

4. 哪个说明符可用于指示将数字格式化为包含逗号和一位小数 (如 123,890.5)?

`N` 或 `n`。要格式化成带逗号和小数点, 可使用 `N1`。

5. 如果 `x` 的值为 123456789.876, 则下述代码的输出是什么?

```
Console.WriteLine("X is {0:'a value of ' #, #.'}", x);
```

123,456,789.8。

6. 在下述情况下, 应使用什么样的说明符: 如果为正数, 则将其显示为至少包含 5 位的 decimal 数; 如果为负, 则至少包含 8 位; 如果为 0, 则显示文本 `<empty>`。

`{0:D5;D8;<empty>}`。其中, `D5;D8;<empty>` 是说明符。

7. 如何取得当前的日期?

使用 `DateTime` 类的静态属性 `Today`。

8. 字符串和 `StringBuilder` 对象之间的关键区别在哪里?

字符串不能修改。方法在操纵字符串时，必须创建一个新的字符串对象；`StringBuilder` 对象包含的字符串信息可以被修改。

9. 哪个特殊字符用作字符串格式符，它对字符串有何影响?

字符串格式符是 `@`，它告诉编译器，以字面意义来解释字符串，而不是将其解释为转移字符。

10. 如何从字符串得到数字型值?

方法之一是使用名称空间 `System` 中的 `Convert` 类。

练习

1. 编写一行代码，将数字格式化为至少包含三位和两位小数。

```
Nbr.ToString("000.00")
```

2. 编写以“星期几，月份，日和四位年份”格式（如 `Monday, January 1, 2002`）打印日期值的代码。

其中的答案之一是：

```
Console.WriteLine("{0:D}", dt);
```

其中 `dt` 是一个 `DateTime` 对象。

3. 下面是一个可能的答案：

```
1: // readln02.cs - Exercise 12.3
2: //-----
3: using System;
4: using System.Text;
5:
6: class MyApp
7: {
8:     public static void Main()
9:     {
10:         StringBuilder Input = new StringBuilder();
11:         string buff;
12:
13:         Console.WriteLine("Enter text. When done, press Ctrl+Z or enter a blank line:");
14:
15:         buff = Console.ReadLine();
16:         while ( (buff != "") && (buff != null) )
17:         {
18:             Input.Append(buff);
19:             Input.Append("\n");
20:             buff = Console.ReadLine();
21:         }
22:         Console.WriteLine("\n\n=====>\n");
23:         Console.Write( Input );
24:         Console.WriteLine("\n\n");
25:     }
26: }
```

4. 下面是一个可能的答案:

```
1: // conv_ex.cs - Exercise 12.4
2: // This listing will cut the front of the string
3: // off when it finds a space, comma, or period.
4: // Letters and other characters will still cause
5: // bad data.
6: //-----
7: using System;
8: using System.Text;
9:
10: class MyApp
11: {
12:     public static void Main()
13:     {
14:         string buff;
15:         int age;
16:         // The following sets up an array of characters used
17:         // to find a break for the Split method.
18:         char[] delim = new char[] { ' ', ',', '.', '.' };
19:         // The following is an array of strings that will
20:         // be used to hold the split strings returned from
21:         // the split method.
22:         string[] nbuff = new string[4];
23:
24:         Console.Write("Enter your age: ");
25:
26:         buff = Console.ReadLine();
27:
28:         // break string up if it is in multiple pieces.
29:
30:         nbuff = buff.Split(delim, 2 );    // Exception handling not added
31:
32:         // Now convert....
33:
34:         try
35:         {
36:             age = Convert.ToInt32(nbuff[0]);
37:
38:             if( age < 21 )
39:                 Console.WriteLine("You are under 21.");
40:             else
41:                 Console.Write("You are 21 or older.");
42:         }
43:         catch( ArgumentException)
44:         {
45:             Console.WriteLine("No value was entered... (equal to null)");
46:         }
47:         catch( OverflowException)
48:         {
```

```

49:         Console.WriteLine("You entered a number that is too big or too small.");
50:     }
51:     catch( FormatException)
52:     {
53:         Console.WriteLine("You didn't enter a valid number.");
54:     }
55:     catch( Exception e )
56:     {
57:         Console.WriteLine("Something went wrong with the conversion.");
58:         throw;
59:     }
60: }
61: }

```

5. 这是选做题。

第 13 天课程的答案

小测验

1. 接口是引用类型还是值类型?

接口是一种类似于抽象类的引用类型。

2. 接口有何用途?

接口定义了类中需要包含的内容，但不定义实际的功能。

3. 接口成员被声明为公有的、私有的还是保护的?

不需要使用其中的任何一种限定符。

4. 接口和类之间的主要差别在哪里?

接口不实现任何成员，而只提供定义。

5. 结构可使用什么样的继承?

结构不能继承，但可以实现接口。

6. 接口中可以包含哪些类型?

属性、事件、虚拟方法和索引器。

7. 如何声明一个这样的公有类：它名为 `Aclass`，从名为 `baseClass` 类派生而来，同时实现一个名为 `IMyInterface` 的接口?

```
public class AClass : baseClass, IMyInterface { ... }
```

8. 可以同时继承多少个类?

一个。

9. 可以同时继承（实现）多少个接口?

可以同时实现 0 或更多个接口。

10. 显式接口实现是如何完成的?

通过在定义成员时指定接口名。另外，调用方法时也必须进行强制转换。

练习

1. 下面是一个可能的答案：

```
interface IId
{
    long ID
    {
        get;
        set;
    }
}
```

2. 下面是一个可能的答案:

```
interface IPosition
{
    bool Location(Point loc);
}
```

3. 该代码片段确实有问题——接口试图实现两个数据成员 (Width 和 Height)。在接口中是不能实现数据成员的。

4. 一个可能的答案如下:

```
1: // rect.cs - Exercise 13.4.
2: //-----
3:
4: using System;
5:
6: public interface IShape
7: {
8:     double Area();
9:     double Circumference();
10:    int    Sides();
11: }
12:
13: public class Rectangle : IShape
14: {
15:     public int width;
16:     public int length;
17:
18:     public double Area()
19:     {
20:         return (double) width * length;
21:     }
22:
23:     public double Circumference()
24:     {
25:         return ((double) ((2 * width) + (2 * length)));
26:     }
27:
28:     public int Sides()
29:     {
30:         return 4;
31:     }
```

```
32:
33:     public Rectangle()
34:     {
35:         width = 0;
36:         length = 0;
37:     }
38: }
39:
40: public class MyApp
41: {
42:     public static void Main()
43:     {
44:         Rectangle rect = new Rectangle();
45:         rect.width = 11;
46:         rect.length = 4;
47:
48:         Console.WriteLine("Displaying Rectangle information:");
49:         displayInfo(rect);
50:         Console.WriteLine("Width: {0}", rect.width);
51:         Console.WriteLine("Length: {0}", rect.length);
52:
53:         for( int L = 0; L < rect.length; L++ )
54:         {
55:             Console.WriteLine("\n");
56:             for( int w = 0; w < rect.width; w++ )
57:             {
58:                 Console.Write("X");
59:             }
60:             Console.WriteLine("\n");
61:         }
62:     }
63:
64:     static void displayInfo( IShape myShape )
65:     {
66:         Console.WriteLine("Area: {0}", myShape.Area());
67:         Console.WriteLine("Sides: {0}", myShape.Sides());
68:         Console.WriteLine("Circumference: {0}",
myShape.Circumference());
69:     }
70: }
```

第 14 天课程的答案

小测验

1. 您呆在起居室，此时电话响了，您起身接电话。电话铃声与下面哪个概念最为类似？

- a. 索引器;
- b. 代表;

- c. 事件;
- d. 事件处理程序;
- e. 异常处理程序。

电话铃声与事件的概念最为类似。

2. 接电话与下面的哪个概念最为类似?

- a. 索引器;
- b. 代表;
- c. 事件;
- d. 事件处理程序;
- e. 异常处理程序。

接电话与事件处理程序最为类似。

3. 索引器的主要功能是什么?

索引器让您能够使用索引表示法来访问类/对象中的信息。

4. 使用哪个关键字来声明索引器?

索引器的创建方式与属性类似,只是使用的是关键字 `this`,而不是属性名。

5. 索引器的定义与下面的哪种定义类似?

- a. 类定义;
- b. 对象定义;
- c. 属性定义;
- d. 代表定义;
- e. 事件定义

索引器定义类似于属性定义。

6. 创建和使用事件包含哪些步骤?

创建并使用事件的过程包含几个步骤——为事件建立代表;创建一个类来将参数传递给事件处理程序;声明事件本身的代码;创建事件发生时执行的代码(事件处理程序)以及导致事件发生的代码。

7. 哪种运算符用于添加事件处理程序?

运算符 `+=` 用于添加事件处理程序; `-=` 用于删除事件处理程序。

8. 给同一个事件指定多个事件处理程序被称为什么?

多点传送 (multicasting)。

9. 下面哪些是正确的(注意:可能没有一个是正确的,也可能都是正确的)?

事件是一个基于代表的实例。

代表是一个基于事件的实例。

事件是一个代表实例。不要被“实例”搞糊涂了。

10. 可以在类中的什么位置实例化事件?

可以在方法或属性内实例化事件。

练习

1. 一个可能的答案如下:

```
1: // indexer.cs - Using an indexer
```

```

2: //-----
3:
4: using System;
5:
6: public class SimpleClass
7: {
8:     int[] numbers;
9:
10:    public SimpleClass(int size)
11:    {
12:        numbers = new int[size];           // declare size elements
13:        for ( int x = 0; x < size; x++ )    // initialize values to 0.
14:            numbers[x] = 0;
15:    }
16:    // exception handling or programming logic should be added to the
17:    // following to make sure the index is not out of range.
18:    public int this[int index]
19:    {
20:        get
21:        {
22:            return numbers[index];
23:        }
24:        set
25:        {
26:            numbers[index] = value;
27:        }
28:    }
29: }
30:
31: public class TestApp
32: {
33:    public static void Main()
34:    {
35:        int size = 10;
36:        SimpleClass myObj = new SimpleClass(size);
37:
38:        for ( int x = 0; x < size; x++ )
39:            myObj[x] = x * x;
40:
41:        for ( int x = 0; x < size; x++ )
42:            Console.WriteLine("Ctr: {0}, Value {1}", x, myObj[x]);
43:
44:    }
45: }

```

2. 一个可能的答案如下:

```

1: // indx2b.cs - Using an array instead of an indexer
2: //-----
3:

```



```
4: using System;
5:
6: public class SpellingList
7: {
8:     public string[] words = new string[size];
9:     static public int size = 10;
10:
11:     public SpellingList()
12:     {
13:         for (int x = 0; x < size; x++)
14:             words[x] = String.Format("Word{0}", x);
15:     }
16: }
17:
18: public class TestApp
19: {
20:     public static void Main()
21:     {
22:         SpellingList myList = new SpellingList();
23:
24:         myList.words[3] = "====";
25:         myList.words[4] = "Brad";
26:         myList.words[5] = "was";
27:         myList.words[6] = "Here!";
28:         myList.words[7] = "====";
29:
30:         for (int x = 0; x < SpellingList.size; x++)
31:             Console.WriteLine(myList.words[x]);
32:     }
33: }
```

3. 可以通过将 DoSort 方法修改为静态的来实现。下面是完整的程序清单:

```
1: // delegib.cs - Using a delegate
2: //-----
3:
4: using System;
5:
6: public class SortClass
7: {
8:     static public int val1;
9:     static public int val2;
10:
11:     public delegate void Sort(ref int a, ref int b);
12:
13:     static public void DoSort(Sort ar)
14:     {
15:         ar(ref val1, ref val2);
16:     }
17: }
```

```

18:
19: public class SortProgram
20: {
21:     public static void Ascending( ref int first, ref int second )
22:     {
23:         if (first > second )
24:         {
25:             int tmp = first;
26:             first = second;
27:             second = tmp;
28:         }
29:     }
30:
31:     public static void Descending( ref int first, ref int second )
32:     {
33:         if (first < second )
34:         {
35:             int tmp = first;
36:             first = second;
37:             second = tmp;
38:         }
39:     }
40:
41:     public static void Main()
42:     {
43:         SortClass.Sort up = new SortClass.Sort(Ascending);
44:         SortClass.Sort down = new SortClass.Sort(Descending);
45:
46:         SortClass.val1 = 310;
47:         SortClass.val2 = 220;
48:
49:         Console.WriteLine("Before Sort: val1 = {0}, val2 = {1}",
50:             SortClass.val1, SortClass.val2);
51:         SortClass.DoSort(up);
52:
53:         Console.WriteLine("Before Sort: val1 = {0}, val2 = {1}",
54:             SortClass.val1, SortClass.val2);
55:         SortClass.DoSort(down);
56:
57:         Console.WriteLine("After Sort: val1 = {0}, val2 = {1}",
58:             SortClass.val1, SortClass.val2);
59:     }
60: }

```

4. 一个可能的答案如下:

```

1: // delegate.cs - Using a delegate
2: //-----
3:
4: using System;

```

```
5:
6: public class SortClass
7: {
8:     public static int [] vals = new int[10];
9:     public static Type arrType;
10:
11:     public delegate void Sort( int[] vals );
12:
13:     public void DoSort(Sort ar)
14:     {
15:         ar(vals);
16:     }
17: }
18:
19: public class SortOption
20: {
21:
22:     public static void Ascending( int[] vals )
23:     {
24:         int tmp;
25:
26:         for (int first = 0; first < (vals.Length - 1); first++ )
27:         {
28:             for ( int second = 1 + first ; second < vals.Length; second++ )
29:             {
30:                 if (vals[first] > vals[second] )
31:                 {
32:                     tmp = vals[first];
33:                     vals[first] = vals[second];
34:                     vals[second] = tmp;
35:                     Console.Write("<.");
36:                 }
37:             }
38:         }
39:     }
40:
41:     public static void Descending( int[] vals )
42:     {
43:         int tmp;
44:
45:         for (int first = 0; first < (vals.Length - 1); first++ )
46:         {
47:             for ( int second = 1 + first; second < vals.Length; second++ )
48:             {
49:                 if (vals[first] < vals[second] )
50:                 {
51:                     tmp = vals[first];
52:                     vals[first] = vals[second];
53:                     vals[second] = tmp;
54:                     Console.Write(">.");
```

```
55:         }
56:     }
57: }
58: }
59:
60: public static void Main()
61: {
62:     SortClass.Sort up = new SortClass.Sort(Ascending);
63:     SortClass.Sort down = new SortClass.Sort(Descending);
64:
65:     SortClass doIT = new SortClass();
66:
67:     SortClass.vals[0] = 325;
68:     SortClass.vals[1] = 31;
69:     SortClass.vals[2] = 23;
70:     SortClass.vals[3] = 1234;
71:     SortClass.vals[4] = 12;
72:     SortClass.vals[5] = 91;
73:     SortClass.vals[6] = 100;
74:     SortClass.vals[7] = 3;
75:     SortClass.vals[8] = 2000;
76:     SortClass.vals[9] = 369;
77:
78:     for ( int x = 0; x < 10; x++ )
79:         Console.WriteLine("Original value {0}: {1}", x, SortClass.vals[x]);
80:
81:     // A check here could be done to determine if the sort should
82:     // be ascending or descending. Based on the answer, you could
83:     // then declare either the up or the down object (instead of
84:     // declaring it above).
85:
86:     Console.Write("\nSorting...");
87:
88:     doIT.DoSort(up);
89:
90:     Console.Write("\n\n");
91:
92:     for ( int x = 0; x < 10; x++ )
93:         Console.WriteLine("Sorted value {0}: = {1}", x, SortClass.vals[x]);
94:     }
95: }
```

其运行结果如下:

```
Original value 0: 325
Original value 1: 31
Original value 2: 23
Original value 3: 1234
Original value 4: 12
Original value 5: 91
```

Original value 6: 100
 Original value 7: 3
 Original value 8: 2000
 Original value 9: 369

Sorting...<.<.<.<.<.<.<.<.<.<.<.<.<.<.<.

Sorted value 0: = 3
 Sorted value 1: = 12
 Sorted value 2: = 23
 Sorted value 3: = 31
 Sorted value 4: = 91
 Sorted value 5: = 100
 Sorted value 6: = 325
 Sorted value 7: = 369
 Sorted value 8: = 1234
 Sorted value 9: = 2000

5. 下面是发生 ToUpperVowels 事件时执行的代码:

```
static void ToUpperVowels(object source, CharEventArgs e)
{
    switch( e.CurrChar )
    {
        case 'a':    e.CurrChar = 'A';
                    break;
        case 'e':    e.CurrChar = 'E';
                    break;
        case 'i':    e.CurrChar = 'I';
                    break;
        case 'o':    e.CurrChar = 'O';
                    break;
        case 'u':    e.CurrChar = 'U';
                    break;
    }
}
```

要添加该事件处理程序, 可以使用下面的代码:

```
tester.TestChar += new CharEventHandler(ToUpperVowels);
```

第 15 天课程的答案

小测验

1. 一秒钟包含多少次滴答 (tick)?
1000 次。
2. 定时器使用下面哪种东西?
a. 代表;

b. 事件;

c. 异常。

定时器引发事件，而事件使用代表。也就是说，a 和 b 都是正确的。

3. 哪个标准化组织负责 C# 和基类库的标准化工作?

ECMA。

4. 使用 `Environment.GetCommandLineArgs` 和 `Main(string args[])` 之间的差别何在?

前者将程序名作为第一个参数返回。

5. 何时创建 `Math` 类的一个实例 (`Math` 对象)?

永远不需要。`Math` 类包含的成员都是静态的，因此根本不需要创建 `Math` 对象。

6. 哪个类或方法可用于确定文件是否存在?

`File.Exists` 方法。该方法让您能够确定使用哪个命令来打开或创建文件。

7. 文件和流之间的区别何在?

流只是一个信息流而已，与之相关联的并不一定是文件或文本。

8. 哪个 `FileMode` 值可用于新建一个文件?

`Append`、`Create`、`CreateNew` 或 `OpenOrCreate`

9. 哪些类用于 XML?

`System.XML` 名称空间包含大量用于处理 XML 文件和数据类。其中有 `DataDocumentNavigator`、`DocumentNavigator`、`XmlAttribute`、`XmlAttributeCollection`、`XmlCDATASection`、`XmlCharacterData`、`XmlComment`、`XmlConvert`、`XmlDataDocument`、`XmlDeclaration`、`XmlDocument`、`XmlDocumentFragment`、`XmlDocumentType`、`XmlElement`、`XmlEntity`、`XmlEntityReference`、`XmlException`、`XmlImplementation`、`XmlLinkedException`、`XmlNamedNodeMap`、`XmlNamespaceManager`、`XmlNameTable`、`XmlNodeNavigator`、`XmlNode`、`XmlNodeList`、`XmlNodeReader`、`XmlNotation`、`XmlParserContext`、`XmlProcessingInstruction`、`XmlQualifiedName`、`XmlReader`、`XmlResolver`、`XmlSignificantWhitespace`、`XmlText`、`XmlTextReader`、`XmlTextWriter`、`XmlUriResolver`、`XmlValidatingReader`、`XmlWhitespace` 和 `XmlWriter` 等

练习

1. 下面是可能的答案之一:

```
1: // binWritel.cs -
2: // Exception handling left out to keep listing short.
3: //-----
4: using System;
5: using System.IO;
6:
7:
8: public struct Person
9: {
10:     public string FirstName;
11:     public string LastName;
12:     public int    Age;
13:     public bool   Member;
14: }
15:
```

```
16: class MyStream
17: {
18:     public static void Main(String[] args)
19:     {
20:         if( args.Length < 1 )
21:         {
22:             Console.WriteLine("Must include file name.");
23:         }
24:         else
25:         {
26:             FileStream myFile = new FileStream(args[0],
27:             FileMode.CreateNew);
28:             BinaryWriter bwFile = new BinaryWriter(myFile);
29:             Person client;
30:
31:             client.FirstName = "Bradley";
32:             client.LastName = "Jones";
33:             client.Age = 21;
34:             client.Member = true;
35:
36:             bwFile.Write( client.FirstName );
37:             bwFile.Write( client.LastName );
38:             bwFile.Write( client.Age );
39:             bwFile.Write( client.Member );
40:
41:             bwFile.Close();
42:             myFile.Close();
43:         }
44:     }
45: }
```

2. 下面是可能的答案之一:

```
1: // math2.cs - Using a Math routine
2: //-----
3: using System;
4:
5: class myMathApp
6: {
7:     public static void Main()
8:     {
9:         int val;
10:        char disp;
11:
12:        for (double ctr = 0.0; ctr <= 10; ctr += .2)
13:        {
14:            val = (int) Math.Round( ( 10 * Math.Cos(ctr)) );
15:            for( int ctr2 = -10; ctr2 <= 10; ctr2++ )
16:            {
```

```

17:         if (ctr2 == val)
18:             disp = 'X';
19:         else
20:             disp = ' ';
21:
22:         Console.Write("{0}", disp);
23:     }
24:     Console.WriteLine(" ");
25: }
26: }
27: }

```

3. 下面是可能的答案之一:

```

1: // writing1.cs - Writing to a text file.
2: // Exception handling left out to keep listing short.
3: //-----
4: using System;
5: using System.IO;
6:
7: public class WritingApp
8: {
9:     public static void Main(String[] args)
10:    {
11:        if( args.Length < 1 )
12:        {
13:            Console.WriteLine("Must include file name.");
14:        }
15:        else
16:        {
17:            Console.WriteLine("Enter text to be stored in the file.");
18:            Console.WriteLine("Enter a blank line to end input.");
19:
20:            StreamWriter myFile = File.CreateText(args[0]);
21:            string buffer;
22:
23:            do
24:            {
25:                buffer = Console.ReadLine();
26:                myFile.WriteLine(buffer);
27:
28:            } while ( buffer != "" );
29:
30:            Console.WriteLine("Done Writing");
31:
32:            myFile.Close();
33:        }
34:    }
35: }

```


4. 在程序清单的最后，应该关闭流/文件。

第 16 天课程的答案

小测验

1. 大部分 Windows 控件都位于哪个名称空间中？

System.Windows.Forms。

2. 哪个方法用于显示窗体？

可以将窗体传递给 Application.Run 方法，或调用窗体实例的 ShowDialog 方法。

3. 将控件加入到窗体中包括哪三步？

创建控件对象，设置对象的属性，将控件加入到窗体中。

4. 将文件 xyz.cs 编译为 Windows 程序时，需要使用的命令行命令是什么？

csc /t:winexe xyz.cs or csc /target:winexe xyz.cs.

5. 如果要在编译文件 xyz.cs 时包含组合体 myAssmb.dll，应该使用的命令行命令是什么？

csc /r:myAssmb.dll xyz.cs or csc /reference:myAssmb.dll xyz.cs

6. Form 类的 Show() 有何功能？使用这种方法存在什么样的问题？

该方法显示窗体，然后转到下一行代码。问题在于该方法不等待用户输入或执行其他任何操作，而是直接转到下一行执行。

7. 下面哪种东西会导致 Application.Run 结束？

- a. 方法；
- b. 事件；
- c. 到达程序的最后一行；
- d. 该方法永远不会结束；
- e. 以上答案都不对。

事件处理程序导致 Application.Run 结束。

8. 哪些颜色可用于窗体？要使用这些颜色，需要包含哪个名称空间？

可使用的颜色多种多样，它们位于名称空间 System.Drawing 中。表 16.2 列出了这些颜色，如果您不喜欢其中的颜色，可以使用 BCL 中的方法创建自己的颜色。

9. 哪个属性用于设置标签的文本值？

Text。

10. 文本框和标签之间有何差别？

文本框让用户输入数据，而标签用于显示数据。

练习

1. 下面是可能的答案之一：

```
1: using System.Windows.Forms;
2:
3: public class AnApp : Form {
4:     public static void Main( string[] args )    {
5:         Application.Run(new AnApp());
6:     }
```

```
7: }
```

2. 下面是可能的答案之一:

```
1: // form3b.cs - Exercise 16.2
2: //-----
3:
4: using System.Windows.Forms;
5: using System.Drawing;
6:
7: public class frmApp : Form
8: {
9:     public static void Main( string[] args )
10:    {
11:        frmApp myForm = new frmApp();
12:        myForm.Text = "Exercise 16.2";
13:
14:        myForm.Width = 200;
15:        myForm.Height = 200;
16:
17:        myForm.StartPosition = FormStartPosition.CenterScreen;
18:
19:        Application.Run(myForm);
20:    }
21: }
```

3. 下面是可能的答案之一。请注意，在将字符串转换为整数时，包含了异常处理。

```
1: // Ex1603.cs -
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8:
9: public class frmGetNumber : Form
10: {
11:
12:
13:     private Button btnOK;
14:     private Label lblPrompt;
15:     private Label lblResponse;
16:     private TextBox txtNbr;
17:
18:     public frmGetNumber()
19:     {
20:         InitializeComponent();
21:     }
22:
23:     private void InitializeComponent()
```

```
24: {
25:     this.FormBorderStyle = FormBorderStyle.Fixed3D;
26:     this.Text = "One To Ten";
27:     this.StartPosition = FormStartPosition.CenterScreen;
28:
29:     // Instantiate the controls...
30:     lblPrompt = new Label();
31:     lblResponse = new Label();
32:
33:     txtNbr = new TextBox();
34:     btnOK = new Button();
35:
36:     // Set properties
37:
38:     lblPrompt.AutoSize = true;
39:     lblPrompt.Text = "Enter a Number:";
40:     lblPrompt.Location = new Point( 20, 20);
41:
42:     txtNbr.Width = 50;
43:     txtNbr.Location = new Point(140, 20);
44:
45:
46:     lblResponse.Width = 250;
47:     lblResponse.Height = 20;
48:     lblResponse.Text = "Enter a number from 0 to 1000.";
49:     lblResponse.TextAlign = ContentAlignment.MiddleCenter;
50:     lblResponse.Location =
51:         new Point(((this.Width/2) - (lblResponse.Width / 2 )), 140);
52:
53:     this.Controls.Add(lblPrompt); // Add label to form
54:     this.Controls.Add(lblResponse);
55:     this.Controls.Add(txtNbr);
56:
57:     btnOK.Text = "Done";
58:     btnOK.BackColor = Color.LightGray;
59:     btnOK.Location = new Point(((this.Width/2) - (btnOK.Width / 2)),
60:                               (this.Height - 75));
61:
62:     this.Controls.Add(btnOK); // Add button to form
63:
64:     // Event handlers
65:     btnOK.Click += new System.EventHandler(this.btnOK_Click);
66:     txtNbr.TextChanged += new
System.EventHandler(this.txtChanged_Event);
67: }
68:
69: protected void btnOK_Click( object sender, System.EventArgs e)
70: {
71:     Application.Exit();
72: }
```

```
73:
74:     protected void txtChanged_Event( object sender, System.EventArgs e)
75:     {
76:         try
77:         {
78:             int iNbr = Convert.ToInt32(txtNbr.Text);
79:
80:             if ( iNbr < 0 )
81:             {
82:                 lblResponse.Text = "Number is less than zero";
83:             }
84:             else if (iNbr > 1000)
85:             {
86:                 lblResponse.Text = "Number is greater than 1000";
87:             }
88:             else
89:             {
90:                 lblResponse.Text = String.Format("Number is {0}", txtNbr.Text);
91:             }
92:         }
93:
94:         catch(ArgumentException)
95:         {
96:             lblResponse.Text = "No number.";
97:         }
98:         catch(FormatException)
99:         {
100:             lblResponse.Text = "Not a number.";
101:         }
102:         catch(OverflowException)
103:         {
104:             lblResponse.Text = "Number too big or too small.";
105:         }
106:     }
107:
108:     public static void Main( string[] args )
109:     {
110:         Application.Run( new frmGetNumber() );
111:     }
112: }
```

4. 该程序使用的是 `Form.Show` 方法, 而该方法显示窗体, 然后继续运行后面的代码。不幸的是, 窗体还没有显示出来, 就已到程序的结尾处。这将导致用户还未见到窗体之前, 程序便结束了, 更不用说对窗体执行什么操作了。下面是修改后的程序:

```
1: using System.Windows.Forms;
2:
3: public class frmHello : Form
4: {
5:
6:     public static void Main( string[] args )
```

```

7:     {
8:         frmHello frmHelloApp = new frmHello();
9:         Application.Run(frmHelloApp);
10:    }
11: }

```

第 17 天课程的答案

小测验

1. 哪个类用于创建单选按钮控件?

RadioButton。

2. 哪个名称空间包含诸如单选按钮和列表框等控件?

System.Windows.Forms。

3. 如何设置窗体中控件的 Tab 顺序?

通过设置每个控件的 TabIndex 属性, Tab 顺序决定于每个控件的 TabIndex 属性, 其起始值为 0。

4. 在列表框中加入选项包括哪几步?

调用 BeginUpdates, 使用 Items.Add 添加选项, 调用 EndUpdate。

5. MainMenu 和 ContextMenu 对象之间的区别何在?

前者是位于窗体顶端的菜单, 后者是弹出式菜单。

6. 在简单的对话框中显示简单消息的简易方式是什么?

使用 MessageBox 类。

7. 基类库中包含哪些可用的对话框?

ColorDialog、PrintPreviewDialog、FontDialog 和 FileDialog

8. 如果要显示一个窗体, 并且不允许应用程序中的其他窗体被显示或激活, 应该使用哪种方法?

ShowDialog 方法。

9. 使用 Show 方法时, 可以同时显示多少过窗体?

任意多个。

练习

1. 编写这样的代码: 将名为 btn1 和 btn2 的单选按钮控件加入到一个名为 grpbox 的组合框中。

```

private GroupBox grpbox = new GroupBox();
private RadioButton btn1 = new RadioButton();
private RadioButton btn2 = new RadioButton();

this.grpbox.Controls.Add(btn1);
this.grpbox.Controls.Add(btn2);

```

2. 要在菜单 MYMENU 中加入一条线段, 需要使用什么代码?

```
MYMENU.MenuItems.Add("-");
```

3. 下面是可能的答案之一:

```
1: // ex1703.cs - using the color dialogs
2: //-----
3:
4: using System;
5: using System.Windows.Forms;
6: using System.Drawing;
7:
8: public class ColorForm : Form
9: {
10:     private MainMenu myMainMenu;
11:
12:
13:     public ColorForm()
14:     {
15:         InitializeComponent();
16:     }
17:
18:     private void InitializeComponent()
19:     {
20:         this.Text = "Exercise - Colors!";
21:         this.StartPosition = FormStartPosition.CenterScreen;
22:         this.FormBorderStyle = FormBorderStyle.Sizable;
23:         this.Width = 400;
24:
25:         myMainMenu = new MainMenu();
26:
27:         MenuItem menuItemFile = myMainMenu.MenuItems.Add("&File");
28:         menuItemFile.MenuItems.Add(new MenuItem("Color",
29:             new EventHandler(this.Color_Selection)));
30:
31:         menuItemFile.MenuItems.Add(new MenuItem("Exit",
32:             new EventHandler(this.Exit_Selection)));
33:         this.Menu = myMainMenu;
34:     }
35:
36:     protected void Exit_Selection( object sender, System.EventArgs e )
37:     {
38:         Application.Exit();
39:     }
40:
41:     protected void Color_Selection( object sender, System.EventArgs e )
42:     {
43:         ColorDialog myColorDialog = new ColorDialog();
44:
45:         if ( myColorDialog.ShowDialog() != DialogResult.Cancel )
46:         {
47:             this.BackColor = myColorDialog.Color;
48:         }
49:     }
50:
```

```
51: public static void Main( string[] args )
52: {
53:     Application.Run( new ColorForm() );
54: }
55: }
```

4. 该程序能够通过编译,并能运行,因此并没有问题。菜单项 Update Date 有一个指示器,指出可以使用 Alt+D 键。但快捷键被定义为 Shortcut.CtrlH。虽然使用两个不同的值可能有些混乱,但并不会导致错误。

5. 这是选做题。

6. 可以在程序清单中加入下面的代码。完整的程序清单请参阅源代码下载文件 (EX1706.cs)。需要在程序清单中加入下述代码:

```
68: protected void FileAbout_Selection( object sender, System.EventArgs e)
69: {
70:     // display an about form
71:     frmAbout myAboutForm = new frmAbout();
72:     myAboutForm.ShowDialog();
73: }
```

另外,还需加入下述代码,以实际创建对话框:

```
132: public class frmAbout : Form
133: {
134:     private Label Description;
135:     private Button btnOK;
136:
137:     public frmAbout()
138:     {
139:         InitializeComponent();
140:     }
141:
142:     private void InitializeComponent()
143:     {
144:         Description = new Label(); // Create label
145:         Description.Text = "My first about box";
146:         Description.AutoSize = true;
147:         Description.Location = new Point( 50, 40);
148:         Description.BackColor = this.BackColor;
149:
150:         btnOK = new Button();
151:         btnOK.Text = "OK";
152:         btnOK.Width = 80;
153:         btnOK.Location = new Point( 60, 80);
154:         btnOK.Click += new System.EventHandler( this.OK_Selection );
155:
156:         this.Controls.Add(btnOK);
157:         this.Controls.Add(Description);
158:
159:         this.Text = "About STY Menus";
```

```

160:      this.Width = 200;
161:      this.Height = 140;
162:      this.StartPosition = FormStartPosition.CenterScreen;
163:      this.FormBorderStyle = FormBorderStyle.Fixed3D;
164:  }
165:
166:  protected void OK_Selection( object sender, System.EventArgs e)
167:  {
168:      this.Close();
169:  }
170: }

```

第 18 天课程的答案

小测验

1. 何为 Web 服务?

一个即使位于 Web 的其他地方也能被执行的进程。

2. 帮助客户程序与 Web 服务进行通信的文件被称为什么?

Web 代理。

3. 哪个程序用于创建与 Web 服务器通信的代码?

WSDL EXE。

4. 如何区分 Web 服务和 ASP.NET 页面?

前者的扩展名为 .asmx，而后者的扩展名为 .aspx。

5. 如何执行 ASP.NET 页面?

将其复制到 Web 服务器中，然后使用浏览器访问文件。

6. 哪两类控件可用于 Web 表单?

服务器端控件和 HTML 服务器控件。

7. 程序清单 18.6 使用的是 HTML 控件、HTML 服务器控件、还是 Web 服务器控件?

Web 服务器控件，因为它们前面有 asp。

8. 标准 HTML 控件和 HTML 服务器控件的区别何在?

后者在服务器上执行，而前者由浏览器执行。HTML 服务器控件通常生成标准的 HTML 控件。

9. 与标准 HTML 表格标记对应的是哪种服务器控件?

HtmlTable。

10. 如何区分服务器端 HTML 控件和标准 HTML 控件?

服务器端控件包含属性 runat=server。

练习

1. 用作 Web 服务的 C# 程序的第一行代码是什么? 假设 Web 服务类名为 DBInfo，其第一个方法名为 GetData。

```
<%@WebService Language="C#" Class="DBInfo"%>
```

2. 要将方法用于 Web 服务中，需要对其做哪些修改?

必须在方法前面加上[WebMethod]。

3. 在 Web 服务 Calc 加入方法 Multiply 和 Divide。

```
1: <%@WebService Language="C#" Class="Calc"%>
2:
3: //-------------------------------------
4: // Ex-WebCalc.asmx
5: //-------------------------------------
6:
7: using System;
8: using System.Web.Services;
9:
10: public class Calc : WebService
11: {
12:     [WebMethod]
13:     public int Add( int x, int y )
14:     {
15:         return x + y;
16:     }
17:
18:     [WebMethod]
19:     public int Subtract( int x, int y )
20:     {
21:         return x - y;
22:     }
23:
24:     [WebMethod]
25:     public int Multiply( int x, int y )
26:     {
27:         return x * y;
28:     }
29:
30:     [WebMethod]
31:     public int Divide( int x, int y )
32:     {
33:         int answer;
34:
35:         if ( y == 0 )
36:             answer = 0;
37:         else
38:             answer = x / y;
39:
40:         return answer;
41:     }
42: }
```

4. 新建一个客户程序，使用练习 3 创建的方法 Multiply 和 Divide。

```
1: // Ex-webclient.cs
2: // Calling a Web service
3: //-------------------------------------
```

```

4:
5: using System;
6:
7: /: public class myApp
8: {
9:     public static void Main()
10:    {
11:        Calc cSrv = new Calc();
12:
13:        Console.WriteLine("cSrv.Add( 11, 33); = {0}",
14:                           cSrv.Add(33, 11));
15:        Console.WriteLine("cSrv.Subtract(33, 11); = {0}",
16:                           cSrv.Subtract(33,11));
17:
18:        Console.WriteLine("cSrv.Multiply( 4, 11); = {0}",
19:                           cSrv.Multiply(4, 11));
20:        Console.WriteLine("cSrv.Divide(55, 11); = {0}",
21:                           cSrv.Divide(55,11));
22:        Console.WriteLine("cSrv.Divide(55, 0); = {0}",
23:                           cSrv.Divide(55,0));
24:    }
25: }

```

5. 这是选做题。

6. 这是选做题。

第 19 天课程的答案

小测验

1. 何为调试?

调试指的是确定并消除程序中的问题的过程。

2. 预处理编译指令以分号结尾吗?

不。预处理编译指令不以分号结束。

3. 是哪种虫子使得术语调试 (debugging, 除掉虫子) 与计算机永远地关联在一起?

在电视节目《百万富翁》中, 这是一个价值一百万的问题。答案是蛾子。

4. 编译器能够发现哪种错误?

编译器能够发现语法错误, 但不能发现运行错误 (逻辑错误)。

5. 逐行查看代码时, 需要核对程序的哪些部分?

应核对整个程序清单, 逐行阅读它们。使用面向对象环境和面向对象编程的优点之一是, 可以在独立的程序清单中编写类。这样可以封装其功能。逐行查看类的代码后, 使用它时便无需再查看。您能够确定调用是正确的, 返回的类型也是预期的。

6. 哪种语言的编译指令最少: C、C++ 还是 C#?

C#。C#删除了许多编译指令, 以避免滥用编译指令导致的编码错误。

7. 用于在程序中定义符号和取消符号定义的是哪些编译指令?

#define 和#undef。

8. 哪种标记用于在命令行定义符号？

/define 或/define。

9. 哪个编译指令让您能够修改编译器使用的行号？哪个值可用于将行号恢复为实际行号？

#line 和#line default

10. 基类库中包含哪些用于帮助调试和其他诊断的类？

System.Diagnostics 中的 Trace 和 Debug 类。

练习

1. 编写定义用于预处理的符号的代码，该符号名为 SYMBOL。

可以在命令行使用/define SYMBOL 或/define SYMBOL，或者在代码中使用#define SYMBOL。

2. 编写使行号从 1000 开始的代码。

#line 1000。

3. 该程序能够通过编译并运行，它显示“Hello Goofy World”。该程序与下面的代码等效：

```
1: // fun2.cs - Same as the fun.cs listing. .
2: //-----
3: using System;
4: #warning This listing is not practical...
5:
6: public class
7: myApp { public static void
8: Main(){Console.WriteLine(
9: "Hello"
10: + " Goofy "
11: + "World"
12: );}
13: }
```

下面是同样的代码，但格式更好：

```
1: // fun2.cs - Same as the fun.cs listing...
2: //-----
3: using System;
4: #warning This listing is not practical...
5:
6: public class myApp
7: {
8:     public static void Main()
9:     {
10:         Console.WriteLine( "Hello" + " Goofy " + "World");
11:     }
12: }
```

4. 只能在程序的开始位置撤销对符号的定义，因此应该删除第 34 行，如下所示：

```
1: // bugbustd.cs -
2: //-----
3:
```

```

4: using System;
5:
6: public class ReadingApp
7: {
8:     #if MYLINES
9:     #line 100
10:    #endif
11:    public static void Main(String[] args)
12:    {
13:        Console.WriteLine("In Main....");
14:        myMethod1();
15:        myMethod2();
16:        Console.WriteLine("Done with Main");
17:    }
18:
19:    #if MYLINES
20:    #line 200
21:    #endif
22:    static void myMethod1()
23:    {
24:        Console.WriteLine("In Method 1");
25:    }
26:
27:    #if MYLINES
28:    #line 300
29:    #endif
30:    static void myMethod2()
31:    {
32:        Console.WriteLine("in Method 2");
33:    }
34: }

```

5. 这是选做题。

6. 这是选做题。

第 20 天课程的答案

小测验

1. 在同一个类中，可以对同一个运算符重载多少次？

可被重载多次，更详细的信息请参阅下一个测验题的答案。

2. 什么因素决定运算符可被重载的次数？

独特的特征标数目。只要每次重载的方法的特征标是独特的，便可以不断重载。

3. 要重载运算符==，必须重载哪些方法？

必须同时重载运算符!=、方法 Equals()和 GetHashCode()。

4. 下面哪些是使用重载运算符的绝佳范例？

a. 将加法运算符重载为将两个字符串对象组合起来。

- b. 将减法运算符重载为计算两个 MapLocation 对象之间的距离。
- c. 将加法运算符重载为将值加 1，同时将++运算符重载为将值加 2。

答案 a。将两个字符串相加相当于串联它们是说得通的。答案 b 也可能是可行的，因为两个 MapLocation 相减相当于计算它们之间的距离也是说得通的。但答案 c 是不可行的，使用单目运算符 + 来执行递增运算也许可行，但其含义并非显而易见。而使用递增运算符来增加一倍是可笑的，因此这绝对不是一个好的重载范例。

5. 如何重载运算符/=?

重载除法运算符/将导致运算符/=自动被重载。

6. 如何重载运算符[]?

使用索引器。

7. 哪些关系运算符可以被重载?

<、>、<=和>=。

8. 哪些单目运算符可以被重载?

+、-、++、--、!、~、true 和 false。

9. 哪些双目运算符可以被重载?

+、-、*、/、%、&、|、^、<<和>>。

10. 哪些运算符不能被重载?

=、..、?:、&&、||、new、is、sizeof 和 typeof。

11. 重载运算符时，必须使用哪些限定符?

static 和 public。

练习

1. 将加法运算符重载为将两个 XYZ 对象相加时，对应的方法头是什么?

```
static public XYZ operator+ ( XYZ first, XYZ second)
```

2. 修改程序清单 20.3，添加一个减法重载方法。该方法接受两个 AChar 对象，返回的结果为这两个 AChar 对象存储的字符值之间的数值型差。

```
1: // Ex20-02.cs - Overloading an operator
2: //-----
3:
4: using System;
5:
6: public class AChar
7: {
8:     private char CH;
9:
10:     public AChar() { this.CH = ' '; }
11:     public AChar(char val) { this.CH = val; }
12:
13:     public char ch
14:     {
15:         get{ return this.CH; }
16:         set{ this.CH = value; }
17:     }
```

```

18:
19:     static public AChar operator+ ( AChar orig, int val )
20:     {
21:         AChar result = new AChar();
22:         result.ch = (char)(orig.ch + val);
23:         return result;
24:     }
25:     static public AChar operator- ( AChar orig, int val )
26:     {
27:         AChar result = new AChar();
28:         result.ch = (char)(orig.ch - val);
29:         return result;
30:     }
31:     // Following added for exercise:
32:     static public int operator- ( AChar first, AChar second )
33:     {
34:         int result;
35:         result = first.ch - second.ch;
36:         return result;
37:     }
38: }
39:
40: public class myAppClass
41: {
42:     public static void Main(String[] args)
43:     {
44:         AChar aaa = new AChar('a');
45:         AChar bbb = new AChar('b');
46:
47:         Console.WriteLine("Original value: {0}, {1}", aaa.ch, bbb.ch);
48:         Console.WriteLine("Difference between {0} and {1} is {2}",
49:             aaa.ch, bbb.ch, (aaa - bbb));
50:
51:         aaa = aaa + 25;
52:         bbb = bbb - 1;
53:
54:         Console.WriteLine("Final values: {0}, {1}", aaa.ch, bbb.ch);
55:         Console.WriteLine("Difference between {0} and {1} is {2}",
56:             aaa.ch, bbb.ch, (aaa - bbb));
57:     }
58: }

```

3. 该代码片段有问题，但可能不是您所发现的。显然，该程序清单重载的是运算符`>=`，但核查是否小于等于。这在编译时不会出错，但用户在使用时将感到迷惑。

该程序清单真正的错误在于返回值。重载关系运算符时需要返回一个布尔值，而这里使用并返回的是一个整数。这才是问题所在。

4. 修改程序清单 20.7，添加两个分别将 `Salary` 对象与 `int` 值和 `long` 值进行比较的方法。

```
1: // Ex20-05.cs - Overloading
```

```
2: //-----
3:
4: using System;
5: using System.Text;
6:
7: public class Salary
8: {
9:     private int AMT;
10:
11:     public Salary() { this.amount = 0; }
12:     public Salary(int val) { this.amount = val; }
13:
14:     public int amount
15:     {
16:         get{ return this.AMT; }
17:         set{ this.AMT = value; }
18:     }
19:
20:     public override bool Equals(object val)
21:     {
22:         bool retval;
23:
24:         if( ((Salary)val).amount == this.amount )
25:             retval = true;
26:         else
27:             retval = false;
28:
29:         return retval;
30:     }
31:
32:     public override int GetHashCode()
33:     {
34:         return this.ToString().GetHashCode();
35:     }
36:
37:     static public bool operator == ( Salary first, Salary second )
38:     {
39:         bool retval;
40:
41:         retval = first.Equals(second);
42:
43:         return retval;
44:     }
45:     static public bool operator == ( Salary first, int second )
46:     {
47:         bool retval;
48:
49:         if (first.amount == second)
50:             retval = true;
51:         else
```

```
52:         retval = false;
53:
54:     return retval;
55: }
56: static public bool operator == ( Salary first, long second )
57: {
58:     bool retval;
59:
60:     if ((long)first.amount == second)
61:         retval = true;
62:     else
63:         retval = false;
64:
65:     return retval;
66: }
67:
68: static public bool operator != ( Salary first, Salary second )
69: {
70:     bool retval;
71:
72:     retval = !(first.Equals(second));
73:
74:     return retval;
75: }
76: static public bool operator != ( Salary first, int second )
77: {
78:     bool retval;
79:
80:     if (first.amount != second)
81:         retval = true;
82:     else
83:         retval = false;
84:
85:     return retval;
86: }
87: static public bool operator != ( Salary first, long second )
88: {
89:     bool retval;
90:
91:     if ((long)first.amount != second)
92:         retval = true;
93:     else
94:         retval = false;
95:
96:     return retval;
97: }
98:
99: public override string ToString()
100: {
101:     return( this.amount.ToString() );
```



```
102:     }
103:
104: }
105:
106: public class myAppClass
107: {
108:     public static void Main(String[] args)
109:     {
110:         string tmpstring;
111:
112:         Salary mySalary = new Salary(24000);
113:         Salary yourSalary = new Salary(24000);
114:         Salary PresSalary = new Salary(200000);
115:         int IntSalary = 30000;
116:         int IntSalary2 = 24000;
117:         long LongSalary = 30000L;
118:         long LongSalary2 = 24000L;
119:
120:         Console.WriteLine("Original values: {0}, {1}, {2}",
121:             mySalary, yourSalary, PresSalary);
122:
123:         if (mySalary == IntSalary)
124:             tmpstring = "equals";
125:         else
126:             tmpstring = "does not equal";
127:
128:         Console.WriteLine("\nMy salary ({0}) {1} IntSalary ({2})",
129:             mySalary, tmpstring, IntSalary );
130:
131:         if (mySalary == IntSalary2)
132:             tmpstring = "equals";
133:         else
134:             tmpstring = "does not equal";
135:
136:         Console.WriteLine("\nMy salary ({0}) {1} IntSalary2 ({2})",
137:             mySalary, tmpstring, IntSalary2 );
138:
139:         if (mySalary == LongSalary)
140:             tmpstring = "equals";
141:         else
142:             tmpstring = "does not equal";
143:
144:         Console.WriteLine("\nMy salary ({0}) {1} LongSalary ({2})",
145:             mySalary, tmpstring, LongSalary );
146:
147:         if (mySalary == LongSalary2)
148:             tmpstring = "equals";
149:         else
150:             tmpstring = "does not equal";
151:
```

```

152:      Console.WriteLine("\nMy salary ({0}) {1}; LongSalary2 ({2})",
153:          mySalary, tmpstring, LongSalary2 );
154:
155:
156:     }
157: }

```

第 21 天课程的答案

小测验

1. 可以使用什么来获取对象、类或其他东西的类型?

`typeof` 和 `Type.GetType`。

2. 哪种类型可用于存储类型值? 它位于哪个名称空间中?

`Type`, 它位于名称空间 `System` 中。

3. 哪种概念在运行阶段提供关于类的信息?

反射。

4. 哪种类型可用于获取关于方法参数的详细信息?

`ParameterInfo` 类型。

5. C#包含了什么来帮助以后扩展该语言或帮助该语言来处理当前还没有出现的概念?

属性 (`attribute`)。

6. 第 15 天使用的 `WebMethod` 标记是一个反射的例子还是属性的例子?

实际上, 两者都有一点, 但更像属性。`WebMethod` 是一个属性, 属性是通过使用反射来评估的。

7. 指出三种预定义的属性。

该问题的答案可以有多个。今天的课程中列出的四个预定义的属性是 `CLSCompliant`、`Conditional`、`Obsolete` 和 `WebMethod`。

8. 属性可以与程序中的哪五样东西关联起来?

属性可以被关联到组合体、事件处理程序、字段、方法、程序模块、参数、属性 (`property`)、返回值、类或结构。

9. 用于属性中的参数有哪两类? 它们之间的区别何在?

位置参数和名称参数。前者位于最前面, 并位于指定的位置; 后者是可选的, 位于位置参数的后面, 其次序无关紧要。

10. 如何限制属性可被关联到的目标?

将 `AttributeUsage` 设置为一个 `AttributeUsage` 目标。

11. 属性参数可以是哪些数据类型?

`bool`、`byte`、`char`、`short`、`int`、`long`、`float`、`double`、`string`、`System.Type` 或 `enum`。还可以是对象或一维数组; 但它们的类型必须是前面列出的。

练习

1. 修改 `Refect.cs` (程序清单 21.1), 对 `Object` 类 (`System.Object`) 进行反射?

```

1: using System;
2: using System.Reflection;
3:

```

```

4: class Mymemberinfo
5: {
6:     public static int Main()
7:     {
8:         //Get the Type and MemberInfo.
9:         string testclass = "System.Object";
10:
11:         Console.WriteLine("\nFollowing is the member info for class: {0}",
12:                             testclass);
13:
14:         Type MyType = Type.GetType(testclass);
15:
16:         MemberInfo[] Mymemberinfoarray = MyType.GetMembers();
17:
18:         //Get the MemberType method and display the elements
19:
20:         Console.WriteLine("\nThere are {0} members in {1}",
21:                             Mymemberinfoarray.GetLength(0),
22:                             MyType.FullName);
23:
24:         for ( int counter = 0;
25:             counter < Mymemberinfoarray.GetLength(0);
26:             counter++ )
27:         {
28:             Console.WriteLine( "{0}. {1} Member type - {2}",
29:                                 counter,
30:                                 Mymemberinfoarray[counter].Name,
31:                                 Mymemberinfoarray[counter].MemberType.ToString());
32:         }
33:         return 0;
34:     }
35: }

```

2. Object 类中包含哪些方法？今天反射过的类型中包含 Object 中的哪些方法？

所有的对象都是从 Object 派生而来的，从今天的程序清单（21.1 和 21.2）的输出可知，其他类型都包含 Object 类型中的所有方法。反射 Object 类型时，得到的结果如下：

Following is the member info for class: System.Object

```

There are 7 members in System.Object
0. GetHashCode Member type - Method
1. Equals Member type - Method
2. ToString Member type - Method
3. Equals Member type - Method
4. ReferenceEquals Member type - Method
5. GetType Member type - Method
6. .ctor Member type - Constructor

```

3. 修改程序清单 22.2，使用 FieldInfo 类型，而不是 MemberInfo，以查看程序清单中的字段值。

```
1: // Ex2103.cs
```

```
2: //-----
3: using System;
4: using System.Reflection;
5:
6: namespace Reflect
7: {
8:
9:     class Mymemberinfo
10:    {
11:        int MYVALUE;
12:        public char efg = 'a';
13:        public long hij = 10000L;
14:
15:        public int myValue
16:        {
17:            set { MYVALUE = value; }
18:        }
19:
20:        public static int Main()
21:        {
22:            //The following is the class being checked
23:            string testclass = "Reflect.Mymemberinfo";
24:
25:            Console.WriteLine(
26:                "\nFollowing is the member info for class: {0}",
27:                testclass );
28:
29:            Type MyType = Type.GetType(testclass);
30:
31:            FieldInfo[] MymemberFieldarray = MyType.GetFields();
32:
33:            //Get the MemberType method and display the elements
34:
35:            Console.WriteLine("\nThere are {0} members in {1}",
36:                MymemberFieldarray.GetLength(0),
37:                MyType.FullName);
38:
39:            for ( int counter = 0;
40:                counter < MymemberFieldarray.GetLength(0);
41:                counter++ )
42:            {
43:                Console.WriteLine( "{0}. {1} Member type - {2}",
44:                    counter,
45:                    MymemberFieldarray[counter].Name,
46:                    MymemberFieldarray[counter].MemberType.ToString());
47:            }
48:            return 0;
49:        }
50:    }
51: }
```

4. 修改程序清单 complete.cs, 允许将同一个属性多次关联到同一个目标, 并将 CodeStatus 属性两次关联到同一个类。

下面的可能的答案之一:

```
1: // Ex2104.cs -
2: //-----
3: using System;
4:
5: [AttributeUsage(AttributeTargets.All, AllowMultiple=true)]
6: public class CodeStatusAttribute : System.Attribute
7: {
8:     private string STATUS;
9:     private string TESTER;
10:    private string CODER;
11:
12:    public CodeStatusAttribute( string Status )
13:    {
14:        this.STATUS = Status;
15:    }
16:
17:    public string Tester
18:    {
19:        set
20:        {
21:            TESTER = value;
22:        }
23:        get
24:        {
25:            return TESTER;
26:        }
27:    }
28:
29:    public string Coder
30:    {
31:        set
32:        {
33:            CODER = value;
34:        }
35:        get
36:        {
37:            return CODER;
38:        }
39:    }
40:
41:    public override string ToString()
42:    {
43:        return STATUS;
44:    }
45: }
46:
```

```
47: // attrUsed.cs - using the CodeStatus attribute
48: //-----
49:
50: [CodeStatus("Final", Coder="Brad")]
51: [CodeStatus("First", Tester="Fred")]
52: public class Circle
53: {
54:     public Circle()
55:     {
56:         // Set up and build a circle class
57:     }
58: }
59:
60: [CodeStatus("Final", Coder="Fred", Tester="John")]
61: [CodeStatus("Done")]
62: public class Square
63: {
64:     public Square()
65:     {
66:         // Set up and build a square class
67:     }
68: }
69:
70: [CodeStatus("Alpha")]
71: public class Triangle
72: {
73:     public Triangle()
74:     {
75:         // Set up and build a triangle class
76:     }
77: }
78:
79: [CodeStatus("Final", Coder="Bill")]
80: [CodeStatus("User Testing", Tester="Melissa")]
81: public class Rectangle
82: {
83:     public Rectangle()
84:     {
85:         // Set up and build a rectangle class
86:     }
87: }
88:
89: class myApp
90: {
91:     public static void Main()
92:     {
93:         PrintAttributes(typeof(Circle));
94:         PrintAttributes(typeof(Triangle));
95:         PrintAttributes(typeof(Square));
96:         PrintAttributes(typeof(Rectangle));
```

```
97:     }
98:
99:     public static void PrintAttributes( Type psdType )
100:    {
101:        Console.WriteLine("\nAttributes for: {0}", psdType.ToString());
102:
103:        Attribute[] attribs = Attribute.GetCustomAttributes(psdType);
104:        foreach (Attribute attr in attribs)
105:        {
106:            CodeStatusAttribute item = (CodeStatusAttribute) attr;
107:            Console.WriteLine(
108:                "Status is {0}. Coder is {1}. Tester is {2}.",
109:                item.ToString(), item.Coder, item.Tester);
110:        }
111:    }
112: }
```

附录 B

C#关键字

关键字在 C# 中有特殊的含义和用途，并被保留。下面是 C# 中的关键字。

abstract

一个限定符，用于规定一个类只能作为另一个类的基类。

as

一个用于在兼容的类型之间进行转换的运算符。将左边的值强制转换为右边的类型。

base

一个使得基类中的值和类型可以被访问的关键字。

bool

一种取值可以是 true 或 false 的逻辑数据类型，相当于 .NET 框架中的 System.Boolean。

break

一个用于控制程序流程的关键字，让程序能够退出循环或条件语句块（switch 或 if）。

byte

一种使用一个字节的内存来存储无符号整数的数据类型，取值为 0 ~ 255；相当于 .NET 框架中的 System.Byte。

case

一个程序流程关键字，定义 switch 语句中的逻辑条件。

catch

程序中 try-catch 错误处理逻辑的一部分。catch 语句块用于指定要处理的异常以及这种异常发生时执行的代码。

char

一种使用 2 字节内存来存储单个 Unicode 字符的数据类型，相当于 .NET 框架中的 System.Char。

checked

一个程序流程关键字，指出需要对整型算术运算进行溢出检查，并做相应的转换。

class

一种可包含数据和方法定义的引用数据类型。类可以包含构造函数、常量、字段、方法、属性、索引器、运算符和嵌套类型。

const

一个用于数据成员或变量的限定符。使用后，相应数据类型的值将为常量，不可修改。

continue

一个程序流程关键字，让程序能够自动进行循环到下一次迭代。

decimal

一种使用 16 个字节内存来存储浮点数的数据类型。decimal 变量的精度比其他浮点类型要高，因此更适合用于存储金额。后缀 m 或 M 用于表示字面值为 decimal 型。这种数据类型相当于.NET 框架中的 System.Decimal。

default

一个标签，当 switch 语句中没有任何 case 语句匹配时，将跳到这里。

delegate

一种引用数据类型，可根据指定的方法特征标接受方法。这种方法特征标是基于代表声明的（类似于诸如 C 和 C++ 语言中的函数指针）。

do

一种循环程序流程结构，导致语句或语句块重复执行，直到语句块末尾的条件为 true。通常也被称为 do...while 语句，因为语句块结尾的条件位于 while 语句中。

double

一种使用 8 个字节的内存来存储浮点数的数据类型，后缀 d 或 D 用于表示 double 字面值。double 相当于.NET 框架中的 System.Double。

else

一种条件程序流程语句，包含当前面的 if 条件为 false 时，将执行的语句或语句块。

enum

一种值数据类型，可以存储大量预先定义的常量值。

event

用于指定事件的关键字，使得程序中发生事件时，指定的代表将被调用。

explicit

一个用于为用户自定义类型指定显式转换运算符的关键字。

extern

一个指出方法为外部的（即位于当前 C# 代码外面）的关键字。

false

一个布尔型字面值，也可用作可被重载的运算符。

finally

try-catch 语句的一部分，在 try 语句块结束后运行，通常用于清理 try 语句块中分配的资源。

fixed

一个用于不可管理（unmanaged）代码中的关键字，用来在内存中锁定引用类型，防止被无用单元收集器删除。

float

一种使用 4 个字节的内存来存储浮点数的数据类型，前缀 f 或 F 用于指示 float 字面值。Float 相当于.NET 框架中的 System.Single。

for

一个用于循环的程序流程语句，包含初始化器、条件和迭代器。for 结构的语句块中的语句将不断重复执行，直到条件为 false。初始化器在 for 语句开始执行时运行，而 for 语句块每执行一次，

迭代器也将执行一次。

foreach

一种循环程序流程结构，让您能够遍历集合或数组。

get

一个特殊的关键字，用于创建取得属性值的存取器，它不是保留字。

goto

一种程序流程结构，从当前位置跳到被标记的位置。

if

一种程序流程结构，当条件为 true 时，执行一个代码块。

implicit

一个用于声明用户自定义类型转换运算符的关键字，可以不指定（将隐式地被使用）。

in

一个与 foreach 结合使用的关键字，指示 foreach 要遍历的集合或数组。

int

一种使用 4 个字节内存来存储带符号整数的数据类型，取值范围为 -2147483648 ~ 2147483647，相当于 .NET 框架中的 System.Int32。对于字面值，如果不带任何后缀，且位于 int 的范围之内，则为 int 型

interface

一个用于声明这样的引用类型的关键字，即定义了一组成员，但没有实现它们。

internal

一个访问限定符，使用数据类型只能在同一个组合体中的文件中被访问。

is

一个用于在运行阶段确定对象是否为特定类型的运算符。

lock

一个用于使代码块至关重要的关键字，该代码块不允许多个线程同时访问它。

long

一种使用 8 个字节的内存来存储带符号整数的数据类型，取值范围为 -9223372036854775808 ~ 9223372036854775807，相当于 .NET 框架中的 System.Int64。后缀 l 或 L 用于表示 long 字面值。

namespace

一个让您能够将类型分组的关键字。用于帮助避免名称冲突，并使得引用类型更容易。

new

一种用于创建对象的运算符，也可用作限定符，来隐藏从基类继承的成员。

null

一个字面值，表示一个不指向任何东西的引用值。

object

一种基于 .NET 框架中的 System.Object 类的类型，其他类型都是从 object 派生而来的。

operator

一个用于在类或结构中重载运算符功能的关键字。

out

一个参数限定符，使得参数引用变量可被用来从方法返回一个值。在方法中，必须给这种变量

赋值。

`override`

一个用于提供新的方法或属性实现的关键字。新的实现将覆盖基类中特征标相同的方法或属性。

`params`

一个参数限定符，指出参数可以包含可变数量的值。这种限定符只能用于方法参数列表中的最后一个参数。

`private`

一个访问限定符，指出结构或类中的方法、属性或其他成员只能在该类或结构中进行访问。

`protected`

一个访问限定符，指出类中的方法、属性或其他成员只能在该类及其派生类中进行访问。

`public`

一个访问限定符，指出类中的方法、属性或其他成员可以在任何地方进行访问。

`readonly`

一个数据成员限定符，指出数据成员（在声明时或在构造函数中）被初始化后，其值不可修改。

`ref`

一个参数限定符，指出对参数变量的修改将反映到被传递的变量中。

`return`

一个用于从方法返回一个值的关键字。执行该关键字后，程序流程将回到调用方法中。

`sbyte`

一个使用一个字节的内存来存储带符号整数的数据类型，取值范围为-128 ~ 127，它相当于.NET 框架中的 `System.SByte`。

`sealed`

一个类限定符，禁止从该类派生出其他类。

`set`

一个特殊的关键字，用于创建设置属性值的存取器，不是保留字。

`short`

一种使用两字节的内存来存储符号整数的数据类型，取值范围为-32768 ~ 32767，它相当于.NET 框架中的 `System.Int16`。

`sizeof`

一个用于确定值类型占用内存空间大小（单位为字节）的运算符。

`stackalloc`

一个用于在堆栈中分配内存块的关键字。内存块的大小取决于其中的数据类型和表达式。分配的内存被赋给指针，且不受无用单元收集的影响。

`static`

一个用于指示对于某种类型的所有实例只存储一个值的关键字，适用于字段、方法、属性、运算符和构造函数。

`string`

一种用于存储 Unicode 字符的数据类型，相当于.NET 框架中的 `System.String`。

`struct`

一种可包含数据和方法定义的值数据类型。结构可以包含构造函数、常量、字段、方法、属性、

索引器、运算符和嵌套类型。

switch

一种根据变量的值改变程序流程的程序流程结构。程序流程可跳到 case 语句或 default 语句。

this

一种用于非静态方法中的关键字，将变量和类或结构的当前实例关联起来。

throw

一种用于引发异常的程序流程语句，指出出现了不正常的情况。与 try 和 catch 一起使用。

true

一个布尔型字面值。也可用作可被重载的运算符。

try

用于异常处理的关键字，try 语句块中包含可能引发异常的代码，该关键字与 catch 和 finally 一起使用。

typeof

一种返回对象的数据类型的运算符，返回的是.NET 中的数据类型（如一个 System.Type 对象）。

uint

一种使用 4 个字节的内存存储无符号整数的数据类型，取值范围为 0 ~ 4294967295，这种数据类型相当于.NET 框架中的 System.UInt32。后缀 u 或 U 用于指示 uint 字面值。

ulong

一种使用 8 个字节的内存存储无符号整数的数据类型，取值范围为 0 ~ 18446744073709551615。这种数据类型相当于.NET 框架中的 System.UInt64，后缀 ul（大小写没关系）用于指示 ulong 字面值。

unchecked

一种用于指示不对整型数据类型的溢出情况进行检查的运算符或语句。

unsafe

一个用于指示代码在管理环境中执行时将不安全的关键字。例如，对于使用指针的代码，应该包含在 unsafe 中。

ushort

一种使用两个字节的内存来存储无符号整数的数据类型，取值范围为 0 ~ 65535。这种数据类型相当于.NET 框架中的 System.UInt16。

using

一个用于创建名称空间别名的关键字，对于名称空间中的数据类型，也可用于避免使用全限定名称。

value

属性存取器 set 设置的变量的名称，不是保留字。

virtual

一个用于方法或属性的限定符，指出该方法或属性可以被覆盖。

void

一个用于表示类型的关键字，指出不需要使用数据类型。在方法声明中，void 可用于声明方法不返回值。

while

一个循环程序流程结构，使得只要条件为真，语句或语句块都将不断重复执行。

附录 C

C#命令行编译器标记

可以设置 C#命令行编译器的选项，使用 `/help` 标记运行命令行编译器，可以查看其所有选项。

输出选项

`/out:<file>`

指出最终的输出文件的名称。如果省略，则输出文件的名称与第一个源文件相同。

`/target:<type>`或`/t:<type>`

指出要创建的程序的类型，其中 `<type>` 可能的取值为：

- `exe`：控制台可执行文件（默认值）；
- `winexe`：创建 Windows 可执行文件；
- `library`：创建一个库；
- `module`：创建一个可被加入到组合体中的模块。

`/define:<symbol list>`或`/d: <symbol list>`

用于定义可用于处理编译指令的符号，类似于在源文件的开始使用编译指令 `#define <symbol>`。

`/doc:<file>`

指定应创建的 XML 文档，该文档的名称为 `<file>`。

输入选项

`/recurse:<wildcard>`

指出当前目录及其子目录中所有与通配符匹配的文件都应包含进来。

`/reference:<file list>`或`/r:<file list>`

指示需要引用指定的组合体文件中的元数据。

`/addmodule:<file list>`

将指定的模块链接到当前的组合体中。

资源选项

`/win32res:<file>`

指定 Win32 资源文件（.res）。

`/win32icon:<file>`

指定将图标用于输出。

`/resource:<resinfo>或/res:<resinfo>`

嵌入指定的资源。

`/linkresource:<resinfo>或/linkres:<resinfo>`

将指定的资源链接到组合体。

代码生成选项

`/debug:[+|-]`

指出是否包含还是忽略调试信息。

`/debug:{full|pdbonly}`

指定调试类型，其中 full 将调试器附加到一个正在运行的程序，这是默认值。

`/optimize[+|-]或/o[+|-]`

指示是否进行优化。

`/incremental[+|-]或/incr[+|-]`

指示是否启用逐步编译功能。

错误和警告选项

`/warnaserror[+|-]`

导致警告被视为错误，这意味着如果有警告，将不会生成最终文件。其中+启用这项功能，-关闭这项功能（默认值）。

`/warn:<n>或/w<n>`

将警告等级设置为 0~4。所有的警告都有等级，但只有高于设定等级的警告才会被显示

`/nowarn:<warning list>`

禁止指定的警告消息。

编程语言选项

`/checked[+|-]`

设置为+, 则对溢出进行检查; 设置为-, 则不检查溢出。

`/unsafe[+|-]`

如果启用 (+), 则允许“不安全”的代码; 如果关闭 (-), 则不允许。

杂项

`@<file>`

读取一个响应文件 (<file>), 其中包含更多的选项。

`/help` 或 `/?`

显示与本附录类似的帮助信息。

`/nologo`

禁止显示编译器的版权信息。

`/noconfig`

防止 CSC.RSP 文件被自动包含进来。

高级选项

`/baseaddress:<address>`

指示要创建的库的基地址 (<address>)。

`/bugreport:<file>`

创建一个名为<file>的 bug 报告文件。

`/codepage:<n>`

指定打开源文件时使用的代码页。

`/utf8output`

以 UTF-8 编码方式输出编译器消息。

`/main:<type>` 或 `/m:<type>`

指出包含入口的类型 (类), 通常为 Main 方法, 其他可能的入口被忽略。

`/fullpaths`

指示编译器生成全限定路径。

/filealign:<n>

指出输出文件段采用的对齐方式。

/nostdlib[+|-]

指出不应使用或引用标准库 (`mscorlib.dll`)。

/lib:<file list>

指出用于查找引用的其他路径。

附录 D

不同的计数系统

作为一名计算机程序员，有时候需要与二进制和十六进制的数字打交道。这个附录将解释这些计数系统及其工作原理。为帮助理解，我们首先来复习一下常见的十进制。

十进制

十进制的基数为 10，您天天都使用。这种计数系统中的数字，如 342 被表示为 10 的幂。第 1 位（最右边的一位）为 10 的 0 次方，第二位是 10 的一次方，等等。任何数的 0 次方都为 1，而任何数的一次方都是它自己。因此，对于 342：

$$3 \quad 3 \times 10^2 = 3 \times 100 = 300$$

$$4 \quad 4 \times 10^1 = 4 \times 10 = 40$$

$$2 \quad 2 \times 10^0 = 2 \times 1 = 2$$

总数为 342

十进制使用 10 个不同的数字：0 ~ 9。下述规则适用于十进制和其他任何计数系统：

- 数字被表示为基数的幂；
- 以 n 为基数的计数系统需要 n 个数。

下面来看看其他计数系统。

二进制

二进制的基数为 2，因此需要两个数：0 和 1。二进制对于计算机程序员很有帮助，因为它可用于表示数字式的开和关，而计算机芯片和内存正是以这样的方式工作的。下面是一个二进制数的例子，以垂直方式书写的 1011，同时给出您更为熟悉的十进制表示。

$$1 \quad 1 \times 2^3 = 1 \times 8 = 8$$

$$0 \quad 0 \times 2^2 = 0 \times 4 = 0$$

$$1 \quad 1 \times 2^1 = 1 \times 2 = 2$$

$$1 \quad 1 \times 2^0 = 1 \times 1 = 1$$

总数为 11（十进制）

二进制有个缺点：表示大数时很繁琐。

十六进制

十六进制的基数为 16，因此需要 16 个数：0-9 和 A-F（表示 10-15）。下面是一个十六进制数的例子（2DA），其对应的十进制数为：

$$2 \times 16^2 = 3 \times 256 = 512$$

$$D \times 16^1 = 13 \times 16 = 208$$

$$A \times 16^0 = 10 \times 1 = 10$$

总数为 730（十进制）

十六进制对于计算机工作而言很有用，因为它基于 2 的幂。十六进制数的每一位相当于二进制的四位，因此每两位相当于二进制的八位。表 D.1 列出了一些十六进制数，及其对应的二进制和十进制数。

表 D.1 十六进制数及对应的二进制和十进制数

十六进制数	十进制数	二进制数
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111
10	16	00010000
FD	240	11110000
FF	255	11111111