

C#是一种面向对象的编程语言，为 ASP.NET 提供了强有力的支持。在微软的 Visual Studio 2010 中，C#与 ASP.NET 很好地结合在一起，使网页设计者可以更快捷、更高效地完成 Web 应用程序开发工作。因此，掌握 C#语言也成为掌握 Web 应用程序开发的基本要求之一。本章将带领读者走进 C#，开启学习 C#编程语言的大门。本章重点介绍 C#语法基础知识，包括 C#语法规则、基本数据类型、运算符、程序控制流程、C#面向对象基础等。语法是编程的基础，因此一定要深刻理解并掌握 C#的语法定义和规则，为后续程序的编写打下良好的基础。

学习目标

- ☒ 了解 C#的特点。
- ☒ 掌握 C#中保留字、标识符、常量及变量的使用方法。
- ☒ 掌握值类型和引用类型。
- ☒ 掌握 C#中常用的运算符。
- ☒ 掌握选择与循环两类程序控制语句。
- ☒ 了解 C#面向对象程序设计中类与对象等基本概念。

3.1 C#基本语法

计算机程序设计语言的发展，经历了从机器语言、汇编语言、高级语言到面向对象程序设计语言的历程。目前，应用较为广泛的高级语言（面向对象程序设计语言）有 C、C#、Java 等。C#是运行在.NET 平台上的一种面向对象的语言，它属于编译解释型语言，其原始代码经过编译成为被.NET 框架识别的编码，并运行在.NET 平台上。

3.1.1 C#特点及语法规则

1. C#的特点

C#是一种完全的面向对象编程语言，其语法和特性借鉴了 C++、Java 等语言的优点，并摒弃了这些语言的劣势。目前来说，C#是对面向对象特性支持最好的语言之一，它是专门为.NET 应用而开发的语言，与.NET 框架完美结合。在.NET 类库的支持下，C#能够全面

地表现.NET Framework 的各种优点。总的来说, C#具有以下突出的优点。

- (1) 语法简洁、自由。
- (2) 保留了 C++的强大功能, 彻底的面向对象设计。
- (3) 丰富的 Web 服务器控件, 与 Web 应用紧密结合。
- (4) 强大的安全性机制。
- (5) 完善的错误、异常处理机制。
- (6) 灵活的版本处理技术。
- (7) 支持跨平台, 兼容性强。

注意: C#虽不能脱离.NET 平台而单独运行, 但它本身并不是.NET 平台体系中的一部分。 .NET 是一个跨语言的平台, 除了支持 C#外, 还支持 VB、C++等语言。

2. C#的语法规则

C#中的程序代码都必须放在一个类中。定义类时使用 `class` 关键字, 格式如下。

```
[修饰符] class 类名  
{ 程序代码 }
```

说明:

(1) 程序代码分为结构定义语句和功能执行语句。结构定义语句用于声明类或方法, 功能执行语句用于实现具体功能, 每条功能执行语句必须用分号结束。

例如:

```
Console.WriteLine("这是第一个 C#程序!");
```

(2) C#语言严格区分大小写。不能将 `class` 写成 `Class`, `Computer` 和 `computer` 是两个完全不同的标识符。

(3) 为增强程序可读性, 应注意代码整齐美观、层次清晰。

3.1.2 关键字与标识符

1. 关键字

关键字是事先定义好并赋予了特殊含义的单词。C#中的关键字有: `abstract`、`as`、`bool`、`byte`、`case`、`class`、`if` 等。每一个关键字都是对编译器具有特殊意义的预定义保留标识符。例如, `void` 关键字用于指定方法不返回值; `class` 是声明类时使用的关键字; `static` 是声明静态的类、字段、方法等使用的关键字。关键字不能在程序中用作标识符, 除非它们有一个@前缀。例如, `@if` 是有效的标识符, 但 `if` 不是, 因为 `if` 是关键字。

注意: C#中所有关键字都是小写的。

2. 标识符

标识符是用来标记类名、方法名、参数名、变量名等的符号, 它由字母、数字、下划线和@符号组成, 不能以数字开头, 而且不能是关键字。例如: `username`、`username123`、`user_name` 和 `_userName` 都是合法的标识符, 而 `123username`、`class`、`98.3`、`Hello World` 都是非法的标识符。

定义标识符时建议遵循以下规则。

- (1) 类名、方法名、属性名首字母大写，称为大驼峰命名法或帕斯卡命名法。例如：ArrayList、LineNumber、Age。
- (2) 字段名、变量名首字母小写，之后每个单词首字母大写，称为小驼峰命名法。例如：age、userName。
- (3) 常量名所有字母都大写，单词间用下划线连接。例如：DAY_OF_MONTH。
- (4) 尽量使用有意义的英文单词定义标识符。例如：用 userName 表示用户名，passWord 表示密码。

3.1.3 常量与变量

1. 常量

常量是指程序执行过程中其值不变的数据，可分为直接常量和符号常量。同变量一样，常量也用来存储数据。它们的区别在于，常量一旦初始化就不再发生变化，也可以理解为符号化的常数。使用常量可以使程序变得更加灵活易读。常量的声明和变量类似，需要指定其数据类型、常量名以及初始值，并需要使用 `const` 关键字。常量的类型可以是任何一种 C# 的数据类型，常量的声明格式如下。

访问修饰符 `const` 常量数据类型 常量名=常量值；

例如：

```
[public] const double PI=3.1415; 该语句声明了一个 double（双精度）型的常量 PI
```

小贴士：常量的定义中访问修饰符可以省略，默认为 `public`。常量名一般采用大写字母组成。

C# 中的常量有整型常量、浮点数常量、布尔常量、字符常量等，需要注意以下几点。

(1) 字符常量表示一个字符，用一对单引号('')引起来，可以是字母、数字、标点符号以及由转义序列表示的特殊字符。例如，'a'、'1'、'\u0000'等。C# 采用 Unicode 字符集，因此字符以 \u 开头。例如，'\u0000' 表示空白字符。

(2) 字符串常量表示由一个或多个字符组成的字符串，用一对双引号("")引起来。例如，"Hello"。

(3) 布尔常量有两个，分别为 `true` 和 `false`，表示关系运算或逻辑运算的结果，用于区分事物的真假。

(4) `null` 常量表示对象的引用为空。

2. 变量

变量是存放临时数据的内存单元，变量名是内存单元的标识符，而变量值是内存单元中存储的数据。实际上变量就是程序运行过程中其值可以改变的量。在 C# 中，变量必须先定义，后使用。且定义变量时必须声明其类型，赋值时必须赋予和变量同一类型的值。C# 中变量名的命名规范如下。

(1) 必须以字母或下划线开头。

(2) 只能由字母、数字、下划线组成，不能包含空格、标点符号、运算符以及其他

符号。

(3) 不能与 C#关键字和库函数同名，如 `class`、`new` 等。

(4) 可以使用 `@` 开始。

小贴士：在 C# 中，变量一定会被定义在某对大括号中，该大括号所包含的代码区域便是该变量的作用域。

3.1.4 注释语句

为增强可读性，可用注释对程序中的功能代码进行解释说明。C# 中的注释有三种：单行注释、多行注释、文档注释。

1. 单行注释

对某行代码进行解释，用符号 “//” 表示。

例如：

```
int a=6;    //定义了一个 int 型变量
```

2. 多行注释

注释内容为多行，以符号 “/*” 开头，以符号 “*/” 结尾。

例如：

```
/* int c = 10;
int x = 5; */
```

3. 文档注释

对类或方法进行说明和描述。在类或方法前输入 3 个 “/”，手动填写描述信息，生成文档注释。

注意：注释不会被编译，多行注释中可以嵌套单行注释，但多行注释不能相互嵌套。

3.2 C#的数据类型

数据类型表示了数据在内存中的存储方式，在声明变量或者常量之前，需要首先定义其数据类型。由于 .NET Framework 是一种跨语言的框架，为了在各种语言之间交互操作，部分 .NET Framework 指定了类型中最基础的部分，称为通用类型系统（Common Type System, CTS）。.NET 将不同编程语言的数据类型进行抽象，编译后都转换为 CTS 类型，以便不同语言的变量相互交换信息。如 VB.NET 中 `integer` 类型和 C# 中 `int` 类型都被转换为 `System.Int32`。

C# 支持 CTS，其数据类型包括基本类型，即类型中最基础的部分，如 `int`、`char`、`float` 等，也包括比较复杂的类型，如 `string`、`decimal` 等。作为完全面向对象的语言，C# 中的所有数据类型都是一个真正的类，具有格式化、序列化以及类型转换等方法。根据在内存中存储位置的不同，C# 中的数据类型可分为值类型和引用类型两类。值类型就好比书架上的

书籍，看的时候，直接拿过来就可以了。而引用类型就相当于存储在计算机系统上的书目或索引，查到的只是某一本书的索引和信息，而不是真正书架上的书。如果想要获取这本书，就需要根据书的索引信息，找到书架上的位置，然后去取。在 C# 中，值类型的数据长度固定，存放于栈内；引用类型的数据长度可变，存放于堆内。

3.2.1 值类型

C# 内置的值类型包括所有简单数据类型、结构类型和枚举类型。值类型的变量直接包含它们的数据。将一个值类型变量赋给另一个值类型时，只复制它包含的值，一个值类型的变化不会引起另一个变量的变化。

值类型变量的声明语法如下。

类型名称 变量列表；

例如：

```
char a,b,c; int i,j;
```

注意：C# 中，变量在使用前需要对其初始化。例如：a=new char(); i=1;

1. 简单数据类型

C# 支持的简单数据类型有整型、浮点型、小数型、字符型和布尔型，各类型的名称、说明以及取值范围等详见表 3-1。

表 3-1 C# 中简单数据类型

类 型	名 称	CTS 类型	说 明	取 值 范 围
整型	byte	System.Byte	8 位无符号整数	0~255
	sbyte	System.SByte	8 位有符号整数	-128~127
	short	System.Int16	16 位有符号整数	-32768~32767
	ushort	System.UInt16	16 位无符号整数	0~65535
	int	System.Int32	32 位有符号整数	$-2^{31} \sim 2^{31}-1$
	uint	System.UInt32	32 位无符号整数	$0 \sim 2^{32}-1$
	long	System.Int64	64 位有符号整数	$-2^{63} \sim 2^{63}-1$
	ulong	System.UInt64	64 位无符号整数	$0 \sim 2^{64}-1$
浮点型	float	System.Single	32 位浮点值，7 位精度	$\pm 1.5 \times 10^{-45} \sim \pm 3.4 \times 10^{38}$
	double	System.Double	64 位浮点值，15~16 位精度	$\pm 5.0 \times 10^{-324} \sim \pm 1.7 \times 10^{308}$
小数型	decimal	System.Decimal	128 位数据，28~29 位精度	$\pm 1.0 \times 10^{-28} \sim \pm 7.9 \times 10^{28}$
字符型	char	System.Char	16 位 Unicode 字符	U+0000~U+ffff
布尔型	bool	System.Boolean	8 位空间，1 位数据	true 或 false

2. 结构类型

结构类型（struct）是包含多个基本类型或复合类型的统一体，在 C# 中可以使用 struct 关键字创建结构。结构类型的声明格式如下。

```
[attributes] [modifiers] struct identifier [:interfaces] body [;]
```

例如，一个学生信息结构声明如下。

```
public struct Student
{
    public long id;           //学号
    public string name;       //姓名
    public double score;      //成绩
}
```

小贴士：本书语法声明格式中，放在方括号[]内的均为可选项，可以省略。

3. 枚举类型

枚举类型（enum）是值类型的一种特殊形式，它从 System.Enum 继承而来，是一组指定常量的集合，可以通过枚举来实现常用变量与值的一种映射关系。枚举类型由名称、基础类型和一组字段组成，每种枚举类型均有一种基础类型，该基础类型可以是除 char 类型以外的任何整型。枚举类型的声明格式如下：

```
[attributes] [modifiers] enum identifier [:base-type] {enumerator-list} [;]
```

例如：

```
public enum Month {Jan, Feb, March, April, May, Jun, July, Sep, Oct, Nov, Dec};
```

枚举类型实际上是一个整数类型，用于定义一组基本整数数据，并可以给每个整数指定一个便于记忆的名字。例如，以下代码声明了一个关于星期的枚举类型。

```
public enum Week {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

3.2.2 引用类型

C#不允许在安全代码中使用指针，因此要处理堆中的数据就需要使用引用数据类型。和值类型相比，引用类型不存储它们所代表的实际数据，而是存储对实际数据地址的引用。引用类型的变量又称为对象，当多个变量引用同一个对象时，对一个变量的操作可以影响到其他变量。引用类型变量的声明格式和初始化与值类型相似，同时使用 new 关键字实例化引用数据类型的对象，并指向堆中的对象数据。

声明引用类型变量的语法结构为：类型名 变量列表；

例如：

```
object obj=new object();
```

在 C#中，引用类型包括对象类型、类类型、字符串类型、接口、数组等。本节主要介绍内置对象类型、字符串类型和数组，有关类的相关知识将在 3.5 节面向对象部分做详细讲解。

1. 对象类型

对象类型 object 在 .NET 框架中是 System.Object 的别名，它是其他类型的基类。在 C# 的统一类型系统中，所有类型（预定义类型、用户定义类型、引用类型和值类型）都是直

接或间接从 `object` 继承的。可以将任何类型的值赋给 `object` 类型的变量。

例如：

```
object obj=12; //将整型值赋予对象类型的变量 obj
```

2. 字符串类型

字符串类型 `string` 是 C# 中定义的一个专门用于对字符串进行操作的类，更常见的做法是使用 `string` 关键字来声明一个字符串变量。`string` 关键字是 `System.String` 类的别名，编译时 C# 中的 `string` 类型会被编译成 .NET Framework 的 `String` 类型。可通过使用 `String` 类的构造函数或使用字符串串联运算符 (+) 来创建一个 `string` 对象。

例如：

```
string a="hello"; string b="h"; b=b+"ello";
```

`String` 类有许多属性和方法用于 `string` 对象的操作。例如，`Length` 属性返回当前 `string` 对象中的字符数，`Concat(string str1, string str2)` 方法用于连接两个 `string` 对象。

3. 数组

数组是一组相同类型的数据组成的集合，每个数组有固定的大小，并且其中的数组元素的类型必须保持一致。

1) 数组的定义

数组可以有多个维度。常用的一维数组的维数为 1，二维数组的维数为 2。每个数组的下标始于 0。

定义一维数组的语法为：

方法 1：类型[] 数组名=new 类型[] {元素 1, 元素 2, ...};

方法 2：类型[] 数组名={元素 1, 元素 2, ...};

例如：

```
int[] arr=new int[10]; //定义长度为 10 的整型数组，元素默认初值都为 0  
string[] myArray = {"sun", "bin", "zhou"}; //定义长度为 3 的 string 型数组同时赋初值
```

定义二维数组的语法为：

方法 1：类型[,] 数组名=new 类型[长度 1,长度 2];

方法 2：类型[,] 数组名={{元素 1},{元素 2},...};

例如：

```
int[,] myArray=new int[3,4]; //定义一个 3×4 的二维数组，初值都为 0  
int[,] numArray = {{0,1}, {2,3}, {4,5}}; //定义一个 3×2 的二维数组，同时赋初值
```

注意：数组可以存储整数、字符串或任何一种用户提出的对象，但同一个数组中的各个元素变量必须是同一类型。

2) 数组的访问

访问数组的元素包括读取或设置某个元素的值。最基本的方法是通过下标定位元素。数组的下标范围为 0~Length-1，其中 `Length` 代表数组的长度。访问数组时下标不能超

范围。

一维数组使用示例：

```
int[] arr = { 1, 2, 3, 4, 5 };           //定义一维数组
for (int i = 0; i < arr.Length; i++)     //使用 for 循环遍历数组的元素
{ Console.WriteLine(arr[i]); }          //通过数组下标访问元素
```

二维数组使用示例：

```
int[,] arr = new int[3, 4] { { 1, 2, 3, 4 }, { 2, 3, 3, 4 }, { 3, 4, 3, 4 } };
int sum = 0;
for (int i = 0; i < arr.GetLength(0); i++) //遍历数组元素
{ int groupSum = 0;
  for (int j = 0; j < arr.GetLength(1); j++)
  { groupSum = groupSum + arr[i, j]; }      //求数组中每一行元素的和
  sum = sum + groupSum; }                  //求数组中所有元素的和
```

小贴士：数组对象的 `GetLength(0)` 属性返回二维数组中第一维的长度，`GetLength(1)` 属性返回二维数组第二维的长度。

3) ArrayList 的使用

`ArrayList` 是一种较为复杂的数组，它能实现可变大小的一维数组。由于数组本身需要固定长度，所以往往不是很灵活。`ArrayList` 则可以动态增加或者减少内部集合所存储的数据对象个数。`ArrayList` 的默认初始容量为 0，容量会根据需要通过重新分配自动增加。`ArrayList` 与数组的最大不同在于声明 `ArrayList` 对象的时候可以不需要指定集合的长度，而在后续使用中动态进行添加。

可以使用 `ArrayList` 的构造函数来创建一个新的列表，常用的形式有以下两种。

```
public ArrayList();
public ArrayList(int capacity);
```

其中，参数 `capacity` 可以指定所创建列表的初始容量。如果不指定，则初始容量为 .NET 的默认值 16。

例如：

```
ArrayList arr1=new ArrayList(); ArrayList arr2=new ArrayList(100);
```

可以通过 `ArrayList` 的 `Add` 和 `AddRange` 方法向列表中添加数据。两者的区别在于：`Add` 一次只能添加一个元素，而 `AddRange` 一次可以添加多个元素，这多个元素原来同样需要放在一个集合或数组中。

例如：

```
ArrayList aList1 = new ArrayList();
aList1.Add("a"); aList1.Add("b"); //aList1 中为"ab"
ArrayList aList2 = new ArrayList();
for (int i = 0; i < 10; i++) { aList2.Add(i.ToString()); } //aList2 中为"0123456789"
aList2.AddRange(aList1); //将 aList1 插入到 aList2 末尾，结果为"0123456789ab"
```



```
foreach(string p in aList2) {Response.Write(p);} //输出 aList2 的内容
```

注意：使用 ArrayList 时，需在代码段引入命名空间的位置加上 using System.Collections; 引入相应的命名空间。

3.2.3 数据类型转换

在高级语言中，只有具有相同数据类型的对象才能够互相操作。很多时候，为了进行不同类型数据的运算(如整数和浮点数的运算)，需要把数据从一种类型转换为另一种类型，即进行类型转换。C#有两种数据类型的转换方式：隐式转换和显式转换。

1. 隐式转换

隐式转换又称自动类型转换。当两个不同类型的操作数进行运算时，编译器会试图对其进行自动类型转换，使两者变为同一类型。进行自动类型转换要同时满足两个条件。

(1) 源操作数和目标操作数的两种数据类型彼此兼容。

(2) 目标类型的取值范围大于源类型的取值范围。

例如：

```
int a=123456; float b=a;
```

以上代码中，系统自动进行了隐式转换，将 int 型变量 a 转换为 float 型并赋值给变量 b。在 C#中，允许将以下源数据类型隐式转换为目标数据类型，详见表 3-2。

表 3-2 C#中支持的隐式转换

源数据类型	目标数据类型
sbyte	short、int、long、float、double、decimal
byte	short、ushort、int、uint、ulong、float、double、decimal
short	int、long、float、double、decimal
ushort	int、uint、long、ulong、float、double、decimal
int	long、float、double、decimal
uint	long、ulong、float、double、decimal
long、ulong	float、double、decimal
float	double
char	ushort、int、uint、long、ulong、float、double、decimal

2. 显示转换

然而，不同的数据类型具有不同的存储空间，如果试图将一个需要较大存储空间的数据隐式转换为存储空间较小的数据，就会出现错误。

例如，int a=4; short b=a; 当试图执行上述代码时将会发生错误，原因是源操作数为 int 型，其范围大于目标操作数的 short 型。也即，当两种类型彼此不兼容，或目标类型取值范围小于源类型时，无法进行隐式转换，此时需采用显示转换。显示转换又称强制类型转换，其语法结构为：

目标类型 目标变量 = (目标类型) 源操作数

例如：

```
double a=3.14; int b=(int) a;
```

小贴士：取值范围较大的数据向取值范围较小的数据强制转换时，易丢失精度。

3.2.4 装箱与拆箱

隐式转换和显示转换属于不同值类型之间的转换，而如果要在值类型和引用类型之间转换，则需要了解装箱和拆箱的操作。确切地说，装箱和拆箱的过程就是值类型与 `object` 类型之间的转换。

1. 装箱

装箱是把值类型转换为 `object` 类型。

例如：

```
int a=3; object b=(object) a; //装箱过程
```

2. 拆箱

拆箱是把 `object` 类型转换为值类型。

例如：

```
object a=3; int b=(int) a; //拆箱过程
```

3.2.5 数据类型检查

为了避免在数据类型转换时出现异常，有必要在类型转换之前进行类型检查。C#中提供了两个有关类型检查的运算符：`is` 和 `as`。`is` 运算符是类型转换前的兼容性检查，而 `as` 运算符则是在兼容的引用类型之间执行转换。

1. is 运算符

`is` 运算符用来检查对象是否与给定的类型兼容或一致，若兼容则返回 `true`，否则返回 `false`。`is` 运算符的语法格式如下所示。

变量 `is` 数据类型

例如，下面的代码用来检查变量 `i` 是否是整数类型。

```
int i = 10;        //声明变量
if (i is int) {执行操作};    //类型检查
```

2. as 运算符

`as` 运算符的作用是执行显式类型转换，但是与前面介绍的类型转换不同，`as` 运算符首先会检查类型是否兼容，如果兼容则开始执行类型转换，如果不兼容则返回 `null`，而不会像强制类型转换一样抛出异常。所以，使用 `as` 运算符可以进行安全的类型转换，而不会在运行时因为异常而终止程序的运行。

`as` 运算符的语法格式同 `is` 运算符相同，`as` 关键字的前面是变量，后面是类型，其语法

格式如下所示。

变量 as 数据类型

例如：

```
object obj = "hello";  
string str = obj as string; //类型兼容，转换成功，str 变量的值为字符串"hello"  
int i = obj as int;         //类型不兼容，返回 null
```

3.3 运算符与表达式

C#中提供了大量的运算符，运算符是指定在表达式中执行什么操作的符号。表达式是可以计算且结果为单个值的代码片段，表达式可以包含文本值、方法调用、运算符及其操作数。运算符通常是编译器定义的，每种语言的编译器不同，所以不同语言的运算符写法也可能不同。C#中常用的运算符有赋值运算符、算术运算符、关系运算符、逻辑运算符等。

3.3.1 赋值运算符

赋值运算符是最常用的运算符，其作用是将赋值符号右边变量的值赋给左边的变量。在 C#中，最基本的赋值运算符是等号。除此之外，为了应对比较复杂的情况，C#中还定义了其他的赋值运算符，详见表 3-3。

表 3-3 赋值运算符及其含义

赋值运算符	含 义	示 例	结 果
=	赋值	a=3; b=2	a=3; b=2;
+=	加等于	a=3; b=2; a+=b;	a=5; b=2;
-=	减等于	a=3; b=2; a-=b;	a=1; b=2;
=	乘等于	a=3; b=2; a=b;	a=6; b=2;
/=	除等于	a=3; b=2; a/=b;	a=1; b=2;
%=	模等于	a=3; b=2; a%=b;	a=1; b=2;

以下是几个简单的赋值语句的例子。

```
int i=5; char char1='S'; boolean flag=false;
```

在赋值运算符中，除了“=”之外，其他的都是特殊的赋值运算符。例如，a+=b 相当于 a+b=b，先进行加法运算，再赋值。同理，-=、*=、/=、%=赋值操作符的使用依此类推。

注意：C#中可通过一条赋值语句对多个变量进行赋值。

例如：

```
int x, y, z; x = y = z = 5;    //同时为三个变量赋值
```

3.3.2 算术运算符

算术运算符作用于整型或浮点型数据的运算，包括了加、减、乘、除等比较熟悉的四则运算方式。除此之外，还包括了累加和递减的运算。C#中定义的算术运算符及其含义见表 3-4。

表 3-4 算术运算符及其含义

算术运算符	含 义	示 例	结 果
+	加法运算/正号	5+5	10
-	减法/取负运算	6-4	2
*	乘法运算	5*5	25
/	除法运算	7/5	1
%	取余数	7%5	2
++	自增（前）	a=2; b=++a;	a=3; b=3;
++	自增（后）	a=2; b=a++;	a=3; b=2;
--	自减（前）	a=2; b=--a;	a=1; b=1;
--	自减（后）	a=2; b=a--;	a=1; b=2;

使用算术运算符时要注意以下几点。

（1）自增和自减运算符（++或--），放在操作数之前则先自增或自减，再进行其他运算；放在操作数之后则先进行其他运算，再完成自增或自减运算。

（2）除法运算（/）中，除数和被除数都为整数，则结果为整数；若其中一个为小数，则运算结果为小数。

（3）取模（%）运算中，运算结果的正负取决于被模数（%左边的数）的符号，与模数（%右边的数）符号无关。例如：(-5)%3=-2，而 5%(-3)=2。

3.3.3 关系运算符

关系运算符用来比较两个值，它返回布尔类型的值 true 或 false。常用的关系运算符及其含义见表 3-5。

表 3-5 关系运算符及其含义

关系运算符	含 义	示 例	结 果
<	小于	5<3	false
>	大于	5>3	true
<=	小于等于	5<=3	false
>=	大于等于	5>=3	true
==	等于	5==3	false
!=	不等于	5!=3	true

注意：不能将关系运算符“==”误写成赋值运算符“=”。

3.3.4 逻辑运算符

逻辑运算符用于对布尔型数据进行操作，结果仍是布尔型。常用的逻辑运算符及其含义见表 3-6。

表 3-6 逻辑运算符及其含义

逻辑运算符	含 义	示 例	结 果
&	与运算	a&b	当 a 和 b 都为 true 时返回 true，否则返回 false
	或运算	a b	当 a 和 b 都为 false 时返回 false，否则返回 true
!	非（取反）	!a	a 为 true，返回 false；a 为 false，返回 true
^	异或	a^b	当 a 和 b 不不同时返回 true，否则返回 false
&&	短路与运算	a&& b	结果与 a&b 相同。 只是当 a 为 false 时，b 不再参与运算
	短路或运算	a b	结果与 a b 相同。 只是当 a 为 true 时，b 不再参与运算

使用逻辑运算符时需注意以下几点。

(1) 逻辑运算符可以对布尔型表达式进行运算。例如： $(x > 3) \& (y != 0)$ 。

(2) “&”和“&&”都表示与操作，当且仅当操作符两边的操作数都为 true，结果才为 true。两者的区别在于，逻辑与运算“&”无论操作符左边的表达式为 true 还是 false，操作符右边的表达式都会进行运算；而短路与运算“&&”当操作符左边为 false 时，操作符右边的表达式不进行运算，而直接返回结果 false，故称“短路与”。

(3) “|”和“||”都表示或操作，任何一边的操作数为 true，其结果就为 true。和短路与运算相似，短路或运算“||”中，当左边操作数为 true 时，右边表达式不再进行运算，直接返回结果 true。

3.3.5 其他常用运算符

除赋值运算符、算术运算符、关系运算符和逻辑运算符之外，C#中还有几个常用的运算符，如创建新对象时使用的运算符 new。

1. 创建对象运算符 new

在面向对象编程中，创建一个类的实例时使用 new 运算符，语法格式为：

类名 对象名=new 类名();

例如：

```
Person p=new Person(); //创建了 Person 类的一个实例对象，名称为 p
```

2. 限定运算符 .

在面向对象编程中，获取对象的方法或属性时需要使用限定运算符“.”，语法格式为：

对象名称.属性名称

或

对象名称.方法名称;

例如:

p.name="张三" ; p.age=18; //将对象 p 的 name 属性设为张三, 将 age 属性设为 18

例如:

button1.Text="确定"; //将按钮 button1 对象的文本属性 Text 设置为“确定”

例如:

button1.Hide(); //调用按钮 button1 对象的 Hide() 方法, 隐藏该对象

3. 条件运算符 ?:

条件运算符可以实现一个双分支的 if 语句, 其语法格式为:

(expr1) ? (expr2) : (expr3);

其含义为, 若表达式 expr1 为真, 则执行 expr2 中的运算, 整个表达式返回 expr2 的值; 若表达式 expr1 为假, 则执行 expr3 中的运算, 整个表达式返回 expr3 的值。

例如:

max=(a>b) ? a : b ; //若 a>b, 则 max=a, 否则 max=b

3.3.6 运算符的优先级

运算符的优先级是指在表达式中哪一个运算符应该首先计算。算术四则运算时“先乘除, 后加减”便是运算符优先级的体现。C#根据运算符的优先级确定表达式的求值顺序, 优先级高的运算先做, 优先级低的操作后做, 相同优先级的操作从左到右依次做, 同时用小括号控制运算顺序, 任何在小括号内的运算都最先进行。C#中运算符的优先级别见表 3-7。

表 3-7 C#中运算符的优先级别

级 别	运 算 符
1	. [] () new
2	++ --
3	* / %
4	+ -
5	< <= > >=
6	= = !=
7	&
8	^
9	
10	&&
11	
12	? :

3.4 程序控制结构

同其他高级语言类似，在处理事物之间逻辑关系的时候，都需要使用程序控制结构。C#中的程序控制结构除了顺序、选择以及循环结构之外，还支持跳转语句。

3.4.1 顺序结构

顺序结构是最简单的程序控制结构，该结构是按照从上到下的顺序执行程序中的每一条语句。

【例 3.1】 编写控制台应用程序，完成圆的面积的计算。程序运行结果如图 3-1 所示，详细代码参见 ex3-1。

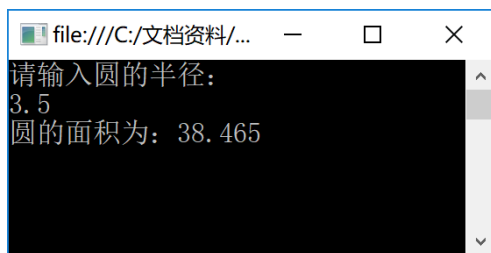


图 3-1 例 3.1 运行结果图---顺序结构示例

核心代码如下：

```
double r, s;  
Console.WriteLine("请输入圆的半径: ");  
r = Convert.ToDouble(Console.ReadLine());  
s = 3.14 * r * r;  
Console.WriteLine("圆的面积为: " + s);  
Console.ReadKey();
```

小贴士：本例中使用了 `Convert.ToDouble()` 函数，作用是将控制台使用 `ReadLine()` 方法读取到的数据强制转换为 `Double` 型数据，再进行运算。

3.4.2 选择结构

选择结构又称为条件分支结构，其作用是根据条件表达式的不同取值做出判断，在程序中传递控制权。C#中的选择结构语句主要有 `if` 语句和 `switch` 语句，当对同一个变量的不同值做条件判断时，使用 `switch` 语句效率会更高一些。

1. if 语句

`if` 语句是最常用的条件语句，通过判断布尔表达式的值，选择执行后面的内嵌语句。

if 语句有三种格式，即单分支的 if 语句、双分支的 if 语句以及多分支的 if 语句。

1) 单分支的 if 语句

语法结构为：

```
if (条件表达式)
{
    语句块;
}
```

在此结构中，当“条件表达式”的值为真时，执行大括号内语句块的内容；当“条件表达式”的值为假时，跳过此结构执行后面的语句。

单分支 if 语句示例如下：

```
int i=5;
if (i<10) {i++;} //代码执行结束之后，i 的值为 6
```

2) 双分支的 if 语句

语法结构为：

```
if (条件表达式)
{
    语句块 1;
}
else
{
    语句块 2;
}
```

在此结构中，如果“条件表达式”为真，则执行语句块 1，否则执行语句块 2。

双分支的 if 语句示例如下：

```
int i=10;
if (i % 2==0)
{ Console.WriteLine("此数是偶数" ); }
else
{ Console.WriteLine("此数是奇数" ); }
```

3) 多分支的 if 语句

语法结构为：

```
if (条件表达式 1)
{
    语句块 1
}
else if (条件表达式 2)
{
    语句块 2
}
```



```
else  
{  
    ...  
}
```

在此结构中，程序从上往下依次判断每一个条件表达式，如果哪个为真，则执行它对应的语句块，如果没有一个条件为真，就执行最后一个 **else** 中的语句块。

多分支的 **if** 语句示例如下：

```
int grade = 75;  
if (grade >= 90)  
{ Console.WriteLine("该成绩等级为优秀!"); }  
else if (grade >= 80)  
{ Console.WriteLine("该成绩等级为良好!"); }  
else if (grade >= 70)  
{ Console.WriteLine("该成绩等级为中等!"); }  
else if (grade >= 60)  
{ Console.WriteLine("该成绩等级为及格!"); }  
else  
{ Console.WriteLine("该成绩等级为差!"); }
```

【例 3.2】 创建一个控制台程序，根据用户输入的成绩，判断该成绩的等级。程序运行结果如图 3-2 所示，详细代码参见 ex3-2。

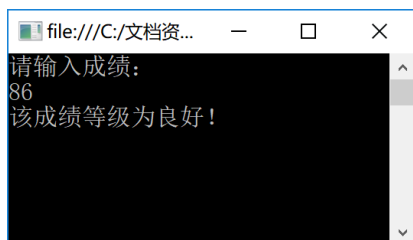


图 3-2 例 3.2 运行结果图——多分支选择结构示例

2. switch 语句

当程序面临的分支较多，或者对同一个表达式的值进行判断时，可以使用 **switch** 语句代替多分支的 **if** 语句，以使得分支结构更加清晰。上例中，当对一个学生的成绩等级进行评估时，会有多个选择：60 分以下、60~69 分、70~79 分、80~89 分、90~100 分，需要根据分数给予不同的等级评定，这时就可以使用 **switch** 语句进行分支。

switch 语句的语法结构为：

```
switch (表达式)  
{  
    case 目标值 1:  
        语句块 1;  
        break;  
    case 目标值 2:
```

```
        语句块 2;  
        break;  
    ...  
    default:  
        语句块 0;  
        break;  
}
```

在上述结构中，使用 `switch` 关键字描述表达式，使用 `case` 关键字描述目标值 `value`，当表达式的值和某个目标值匹配时，执行其 `case` 后对应的语句块。例如，表达式的值为目标值 1 时，执行语句块 1；表达式的值为目标值 2 时，执行语句块 2；依此类推。若表达式的值与目标值 1 到目标值 `n` 都不相同，则执行 `default` 关键字之后的语句块 0。

`switch` 语句使用示例如下：

```
int week = 2;  
switch (week)  
{  
    case 1:  
    case 2:  
    case 3:  
    case 4:  
    case 5:    //当 week 满足值 1~5 中任意一个时，处理方式相同  
        Console.WriteLine("今天是工作日");  
        break;  
    case 6:  
    case 7:    //当 week 满足值 6、7 中任意一个时，处理方式相同  
        Console.WriteLine("今天是休息日");  
        break;  
}
```

注意：`case` 之后取多个目标值时，若执行同样的语句，则只需在最后一个目标值之后书写一次需要执行的语句块。且在 `switch` 语句中，每个要执行的语句块之后都有一个 `break` 语句，当 `switch` 之后的表达式满足 `case` 中的某个目标值时，则执行其后的代码块，然后执行 `break` 语句，退出 `switch` 结构。

【例 3.3】 创建控制台程序，输入一个年份和一个月份，判断该年是否是闰年？这个月属于哪个季节？有多少天？程序运行结果如图 3-3 所示，详细代码参见 ex3-3。

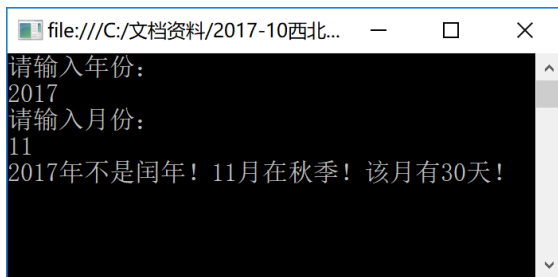


图 3-3 例 3.3 运行结果图---选择结构综合示例

核心代码如下:

```
static void Main(string[] args)
{
    int year, month;
    string stryear, strmonth, strday;
    bool flag;
    stryear = strday = strmonth = "";
    Console.WriteLine("请输入年份: ");
    year = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("请输入月份: ");
    month = Convert.ToInt32(Console.ReadLine());
    if (((year % 4 == 0) & (year % 100 != 0)) | (year % 400 == 0))
    {
        flag = true; stryear = "是闰年! ";
    }
    else { flag = false; stryear = "不是闰年! ";}
    switch (month)
    {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            strday = "该月有 31 天! ";
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            strday = "该月有 30 天! ";
            break;
        case 2:
            if (flag==true)
            {strday = "该月有 29 天! ";}
            else
            {strday = "该月有 28 天! ";}
            break;
        default:
            strday = "您输入的月份无效! ";
            break;
    }
    switch (month)
    {
        case 3:
        case 4:
```

```
case 5:
    strmonth = "在春季! ";
    break;
case 6:
case 7:
case 8:
    strmonth = "在夏季! ";
    break;
case 9:
case 10:
case 11:
    strmonth = "在秋季! ";
    break;
case 12:
case 1:
case 2:
    strmonth = "在冬季! ";
    break;
default:
    strmonth = "月份无效! ";
    break;
}
Console.WriteLine(year+"年"+stryear+month+"月"+strmonth+strday);
Console.ReadKey();
}
```

3.4.3 循环结构

循环语句是一种重要的程序控制结构，其特点是：在给定条件成立时，反复执行某程序段，直到条件不成立为止。给定的条件称为循环条件，反复执行的程序段称为循环体。在 C# 中，循环语句分别是：for 语句、while 语句、do-while 语句和 foreach 语句。

1. for 语句

通常用在循环次数已知的情况下。语法结构为：

```
for (循环变量初始化表达式; 循环条件表达式; 步长操作表达式)
{
    //重复执行的循环体语句块;
}
```

for 语句的执行过程为：首先对循环变量赋初值，判断循环条件表达式是否为真；如果为真则执行循环体，否则跳出循环；执行完一次循环体之后，对循环变量增加一个步长，再进行循环条件判断，如果为真则执行，否则跳出循环；依次执行，直到循环条件表达式为假。

for 语句使用示例如下：

```
int sum = 0;    //定义变量 sum，用于存储累加所得的和
for (int i = 1; i <= 10; i++)    //i 的值会在 1~10 之间变化
{ sum += i; }    //实现 1~10 这十个数累加，结果存至 sum 中
```

注意：循环体放在一对大括号 {} 中。

2. while 语句

当循环次数未知，而程序需要重复执行某种功能，直到达到某种条件才停止，可以使用 while 循环结构。while 循环用来在指定的条件内，不断地重复指定的动作。语法结构如下：

```
while (条件表达式)
{
    //要重复执行的语句块
}
```

语句执行时，反复进行条件判断，只要条件表达式成立，则执行循环体 {} 内的语句，直到条件不成立，while 循环结束。

while 语句使用示例如下：

```
int sum, i;    //定义变量
sum=0; i=1;    //变量赋初值
while (i<=10)
{ sum+=i; i+=1; }
```

3. do-while 语句

与 while 语句功能类似，其语法结构为：

```
do
{
    //重复执行的语句块;
}
while (条件表达式)
```

do-while 语句使用示例：

```
int sum, i;    //定义变量
sum=0; i=0;
do {sum+=i; i+=1;}
while (i<=10)
```

与 while 语句示例相似，以上代码通过 do-while 语句同样完成了 1~10 这十个数求和。

小贴士：do-while 语句与 while 语句可以相互转化。两者的不同之处在于，while 语句是先判断循环条件是否满足，若满足则执行循环体，若不满足则直接退出循环。而 do-while 语句执行时先执行一次循环体语句块，再判断 while 后的条件是否成立，若为真则再次执行循环体，否则退出。因此，do-while 语句中的循环体，无论条件表达式是否满足都至少被执行一次。

思考：至少执行一次循环体的循环语句是什么？已知循环次数的循环语句是哪个？

4. foreach 语句

`foreach` 循环是 C#语言提供的一种新的循环结构，该语句主要用于循环访问数组或集合中的元素，其语法结构如下：

```
foreach (集合子项 in 集合)
{
    //对子项处理的语句块;
}
```

`foreach` 是对一个集合变量进行循环，依次取出集合中的一项对其操作，循环的执行次数为此集合中的子项的数目。使用此语句可以方便快捷地完成对集合对象的循环操作。

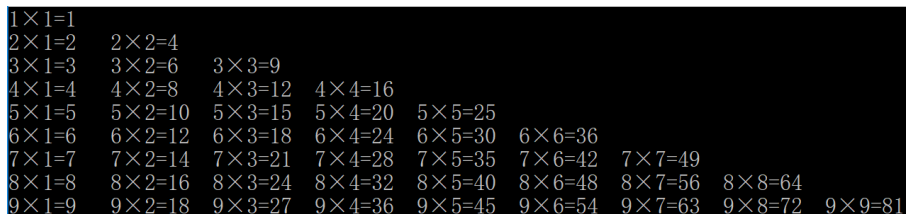
`foreach` 语句使用示例如下：

```
int[] myarr={11,22,33,44,55,66,77,88,99};
int sum=0;
foreach (int i in myarr)
sum+=i;
```

注意：上述代码首先将 1~10 这十个数存放至一维数组 `myarr` 中，之后通过 `foreach` 语句实现集合中 10 个元素相加。`foreach` 语句中的 `i` 表示数组 `myarr` 中的每一个元素，它不同于 `for` 以及 `while` 语句中的循环变量 `i`。

小贴士：循环语句可以相互嵌套，即一个循环体中可以嵌套另一个循环结构。`while`、`do-while` 和 `for` 循环语句都可以自身嵌套或相互嵌套。

【例 3.4】 创建控制台程序，输出九九乘法表，如图 3-4 所示。详细代码参见 ex3-4。



```
1×1=1
2×1=2  2×2=4
3×1=3  3×2=6  3×3=9
4×1=4  4×2=8  4×3=12  4×4=16
5×1=5  5×2=10  5×3=15  5×4=20  5×5=25
6×1=6  6×2=12  6×3=18  6×4=24  6×5=30  6×6=36
7×1=7  7×2=14  7×3=21  7×4=28  7×5=35  7×6=42  7×7=49
8×1=8  8×2=16  8×3=24  8×4=32  8×5=40  8×6=48  8×7=56  8×8=64
9×1=9  9×2=18  9×3=27  9×4=36  9×5=45  9×6=54  9×7=63  9×8=72  9×9=81
```

图 3-4 例 3.4 运行结果图

核心代码如下：

```
string s="";
for (int i = 1; i <= 9; i++)
{
    for (int j = 1; j <= i; j++)
    {
        s = s+i + "x" + j + "=" + (i * j); //得到每一个乘式
        if (i * j < 10)
            { s = s + "  "; } //每个乘式后面加空格分隔，以保证对齐
        else
```

```
        { s = s + " "; }  
    }  
    Console.WriteLine(s);    //输出乘法表中的每一行  
    s = "";  
}
```

3.4.4 跳转语句

跳转语句提供了程序之间跳转执行的功能。在某些条件和需求下，跳转语句可以使代码的编辑更加灵活、方便和有效。C#中的跳转语句包括 `break`、`continue`、`goto` 和 `return` 等。

1. break 语句

`break` 语句常常用于以下两种情况。

- (1) 用在 `switch` 条件语句中，可以终止其所在的 `case` 条件，并跳出当前 `switch` 语句。
- (2) 用在循环语句中，可以终止并跳出当前所在的循环语句，执行循环语句之后的代码。

`break` 语句使用示例如下：

计算 $1+2+3+\dots+i+\dots+200$ ，当和大于 3000 时结束求和操作，输出和小于 3000 的最大的 i 值。核心代码如下：

```
int sum = 0;  
for (int i = 1; i <= 200; i++)  
{  
    sum = sum + i;  
    if (sum > 3000)  
    { Console.WriteLine(i-1);  
      break; //结束整个循环  
    }  
}
```

2. continue 语句

`continue` 语句经常配合循环语句一起使用，其作用是终止当前循环，而继续下一次循环。与 `break` 语句不同的是，`continue` 语句不是跳出整个循环语句，只是终止当前的这一次循环，继而执行下一次的循环。

`continue` 语句使用示例：

计算 $3+6+9+\dots+198$ ，即对 1~200 范围内 3 的倍数的数求和。核心代码如下：

```
int sum = 0;  
for (int i = 1; i <= 200; i++)  
{  
    if (i % 3 == 0)  
        sum = sum + i;  
}
```

```
        else
            continue; //结束本次循环, 开始下一次循环
    }
    Console.WriteLine("1~200 范围内 3 的倍数的数之和为" + sum);
```

3. goto 语句

`goto` 语句将程序控制直接传递给标记语句, 可以嵌入到任何的代码块中, 通常用在 `switch` 语句或深嵌套循环中, 其作用是可以根据情况的不同, 执行指定的代码。

`goto` 语句使用示例:

```
int i = 0;
goto cc;
i = 9;
cc: Console.Write(i);
```

说明: 上述代码中 `cc` 为标签, 标签后面跟一个冒号。使用 `goto` 语句可以将程序控制权跳转至任意标签处。

4. return 语句

`return` 语句一般应用在方法中, 终止方法体中 `return` 语句行下面的代码, 并将控制返回给调用方法。`return` 语句还可以返回一个可选值, 如果没有返回值, 则调用方法为 `void` 类型。

`return` 语句使用示例:

```
double r, area;
r=3;
area=r*r*3.14;
return area;
```

说明: 上述代码中使用 `return` 语句以 `double` 型返回 `area` 变量的值。

3.5 C#面向对象基础

3.5.1 面向对象概述

早期的软件开发采用的是结构化的程序设计方法。程序设计人员把一个有待求解的问题按照自顶向下的原则进行分解, 以便形成一个个相对简单独立的子问题, 然后利用子程序或函数来解决这些子问题, 用子程序或函数之间的数据通信来模拟这些子问题间的联系, 最后把这些子程序或函数装配起来以形成解决问题的完整程序。

面对日趋复杂的应用系统, 结构化的程序设计方法逐渐暴露了一些弱点, 于是面向对象的程序设计方法应运而生。在面向对象的程序设计方法中, 程序设计人员不是完全按照过程对求解问题进行分解的, 而是按照面向对象的观点来描述并分解问题, 最后选择一种

支持面向对象方法的程序语言来解决问题。在这种方法中,设计人员直接用一种称之为“对象”的程序构件来描述客观问题中的“实体”,并用“对象”间的消息来模拟实体间的联系,用“类”来模拟这些实体的共性。

面向对象的程序设计方法将问题抽象为多个独立对象,以对象为基础来构建代码或者项目,通过调用对象方法来解决,成为当今软件开发方法的主流。面向对象的编程方法具有 4 个基本特征。

1. 抽象

抽象就是忽略一个主题中与当前目标无关的那些方面,以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题,而只是选择其中的一部分,暂时不用部分细节。例如为设计一个学生成绩管理系统而考查学生这个对象时,只要关心学生的班级、学号、成绩等,而不用去关心学生的身高、体重这些信息。抽象包括两个方面,一是过程抽象,二是数据抽象。过程抽象是指任何一个明确定义功能的操作都可被使用者当作单个的实体看待,尽管这个操作实际上可能由一系列更低级的操作来完成。数据抽象定义了数据类型和施加于该类型对象上的操作,并限定了对象的值只能通过调用这些操作来修改和观察。

2. 封装

封装是面向对象的特征之一,是对象和类概念的主要特性。封装是把过程和数据包裹起来,使对数据的访问只能通过已定义的界面来进行。面向对象计算始于这个基本概念,即现实世界可以被描绘成一系列完全自治、封装的对象,这些对象通过一个受保护的接口访问其他对象。一旦定义了一个对象的特性,则有必要决定这些特性的可见性,即哪些特性对外部世界是可见的,哪些特性用于表示内部状态。在这个阶段定义对象的接口,通常,应禁止直接访问一个对象的实际表示,而应通过操作接口访问对象,这称为信息隐藏。事实上,信息隐藏是用户对封装性的认识,封装则为信息隐藏提供支持。封装保证了模块具有较好的独立性,使得程序维护修改较为容易。对应用程序的修改仅限于类的内部,因而可以将应用程序修改带来的影响减少到最低限度。

3. 继承

继承是一种联结类的层次模型,并且允许和鼓励类的重用,它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生,这个过程称为类继承。新类继承了原始类的特性,新类称为原始类的派生类(子类),而原始类称为新类的基类(父类)。派生类可以从它的基类那里继承方法和实例变量,并且派生类可以修改或增加新的方法使之更适合特殊的需要。这也体现了大自然中一般与特殊的关系。继承性很好地解决了软件的可重用性问题。比如说,所有的 Windows 应用程序都有一个窗口,它们可以看作都是从一个窗口类派生出来的。但是有的应用程序用于文字处理,有的应用程序用于绘图,这是由于派生出了不同的子类,各个子类添加了不同的特性。

4. 多态性

多态性是指允许不同类的对象对同一消息作出响应。比如同样的加法,把两个时间加在一起和把两个整数加在一起肯定完全不同。又比如,同样的粘贴操作,在字处理程序和

绘图程序中有不同的效果。多态性语言具有灵活、抽象、行为共享、代码共享的优势，很好地解决了应用程序中的函数同名问题。

由于面向对象的程序设计采用抽象、继承、封装和多态等方法，因此在程序开发方面具有许多优点，如开发时间短，效率高，可靠性高，更强壮，更易于维护升级等。

3.5.2 类与对象

1. 类

类是具有相同属性和行为的同一类对象的抽象描述，是面向对象思想中最为核心的概念之一。同时，类也是 C# 中最强大的数据类型，它可以包含数据成员、函数成员（方法、属性、事件、索引器、构造函数和析构函数等）以及嵌套类型。类中可以定义字段和方法，字段用于描述对象的特征，方法用于描述对象的行为。在 C# 中，使用关键字 `class` 来定义类，其语法结构如下。

```
class 类名
{
    //添加字段、属性、方法、事件等的声明
}
```

类的主要成员是特性和行为，即属性和方法。

(1) 属性。属性封装了类的内部数据，类的特性可以通过成员变量体现出来。如果成员变量的修饰符是 `public`，则在创建类的实例时，就可以直接访问。如果修饰符是 `private`，该成员变量只能在类的内部访问，通常这种情况是为类中的方法服务的。属性是类中字段和方法的结合体，通过属性的定义，调用该类的时候，可以直接对该类的属性进行读写操作。属性的定义通过 `get` 和 `set` 关键字来实现，`get` 关键字用来定义读取该属性时的操作，而 `set` 关键字用来定义设置该属性时的操作。声明属性的语法结构如下所示。

```
public [数据类型] [属性名]
{
    get { //Property get code}
    set { //Property set code}
}
```

(2) 方法。通过方法可以封装一段功能完整的代码，这样有利于代码的复用性。例如，可以把计算圆形面积的代码封装到一个方法中，在调用时，只需要传递不同的半径参数即可。通过设置半径参数的数值，就可以获取不同的面积。创建方法的语法格式如下所示。

```
[作用域] 返回类型 方法名(参数列表)
{
    //方法体
}
```

类的定义示例如下：

```
public class Person    //定义 Person 类，public 为访问修饰符
{
    public string name;    //定义 string 类型的字段 name，描述 Person 类的姓名属性
    public int age;        //定义 int 类型的字段 age，描述 Person 类的年龄属性
    public void Speak()    //定义 Speak() 方法，描述 Person 类的行为
    { Console.WriteLine("大家好，我叫" + name + "，我今年" + age + "岁!"); }
}
```

2. 对象

对象是类实例化的产物，它把类的描述具体化。C#中使用 **new** 关键字创建类的对象，语法格式为：

类名 对象名称=new 类名();

例如：

```
Person p=new Person(); //创建 Person 类的实例对象 p
```

创建对象后，可通过对象访问该类对象所有的成员。

语法格式为：

对象名称.成员名称

例如：

```
p.name="张三"; p.age=18; p.Speak();
```

3.5.3 命名空间

命名空间（namespace）是用来组织对象类型的。通过命名空间，可以把相同功能的类型组织在一起，便于管理。与文件夹的概念不同，命名空间是一个逻辑组合，并不是物理组合。C#中包含有 80 多个命名空间，每个命名空间中又有上千个类。

通过 **using** 关键字可以引入命名空间，这样在调用该命名空间下的某个类时就无须在类名前加命名空间的名字了。

例如：

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

上述代码使用了 **System** 命名空间中定义的类 **Console**。如果在 VS.NET 代码编辑窗口中引入命名空间的位置加入了 **using System** 代码行，则表明已经引入了 **System** 命名空间，此时，调用 **WriteLine** 方法时可直接写 **Console.WriteLine("Hello World!");**即可，无须在 **Console** 前加上 **System**。

也可以使用 **namespace** 关键字自定义命名空间。命名空间是在对象类型之上的概念，所以即使是不同的类文件，也可以在同一个命名空间中定义。此外，在操作系统中，文件夹是可以嵌套的，也就是说一个文件夹里还有一个子文件夹。而命名空间也可以像文件夹那样嵌套，也就是说在一个命名空间内部可以再定义一个命名空间。例如：

```
namespace N1    //定义命名空间 N1
{
    namespace N2    //定义命名空间 N2
    {
        class C    //定义类 C
        {
            //类 C 的声明代码段
        }
    }
}
```

3.6 本章小结

C#语法是学习 ASP.NET 编程的基础，要想在代码编写过程中得心应手，必须首先熟练掌握该编程语言的语法基础。本章重点介绍了 C#编程语言的语法基础，让读者能够学习并掌握基本数据类型、常量和变量的定义、运算符和表达式以及 C#面向对象编程中的一些基本概念。本章的重点是在理解.NET 和 C#之间的关系的基础上，熟练掌握 C#语法知识。这些语法的理论知识虽然较为枯燥，但是大部分语言虽语法格式各不相同，但其编程中的程序设计思想却是相通的。所以，在实际的操作练习中，应该多注意不同语言在语法定义方面的异同点，灵活运用。

习题 3

1. 简述 C#的特点。
2. C#中变量名的命名规范有哪些？
3. 下列哪些可以作为 C#中用户自定义的标识符？
(1) Student (2) class (3) if (4) _3sum (5) 3ab
(6) x-yz (7) X + y (8) Hello! (9) 'Hoo' (10) sqr
4. C#中的数据类型包括 () 和 ()。
5. 在 C#统一类型系统中，所有类型都是直接或间接地从 () 继承。
6. 下列数据类型属于值类型的是 ()。
A. struct B. class C. interface D. delegate
7. 下列数据类型属于引用类型的是 ()。
A. bool B. char C. string D. enum
8. 下列数据哪些是变量？哪些是常量？是什么类型的常量？
(1) name (2) 'name' (3) false (4) '120' (5) 100
(6) 12.345 (7) num

9. 什么是装箱？什么是拆箱？
10. 用于引用命名空间的关键字是什么？定义命名空间的关键字是什么？
11. 如何交换两个变量 x 和 y 的值？请写下相应的赋值语句。
12. 编写控制台应用程序，输入圆的半径，计算并输出圆的周长和面积。
13. 编写控制台应用程序，输入长方形的长、宽、高，计算并输出其表面积和体积。
14. 编写控制台应用程序，计算并输出 100 以内所有奇数的和以及偶数的和。
15. 简述面向对象中类与对象的概念。