简单、先进、多用途的面向对象编程语言

Introduction Manual

C#入门手册

极客学院出版

前言

C# 是一门简单、先进、多用途的面向对象编程语言，它由微软 .NET 研究员 Anders Hejlsberg 和他的团队研发完成。本指南由浅入深的向读者讲解 C# 编程语言的基础及 C# 语言的高级使用。

适用人群

本指南旨在帮助那些对 C# 语言感兴趣的初学者。

学习前提

C# 编程语言基础部分和 C/C++ 语言相似，如果你对 C/C++ 语言了解，那么你学习本指南将会比较轻松。

学习编程没有捷径，动手操作是最快的最有效的方法。

版本信息

书中演示代码基于以下版本： C# 5.0

目录

1

C# 基础篇

概述

C# 是一个现代的，通用的，面向对象的编程语言，由微软（Microsoft）开发并获得欧洲计算机制造商协会（ECMA）和国际标准化组织（ISO）认可。

C# 由 Anders Hejlsberg 和他的团队在 .Net 的框架开发期间开发。

C# 是专为公共语言基础结构（CLI）设计的，包括可执行代码和运行环境，允许在不同的计算机系统和体系结构上使用各种高级语言。

下面列出了 C# 成为一种广泛应用的专业语言的原因：

- 它是一种现代的、通用的编程语言。
- 它是面向对象的。
- 它是面向组件的。
- 它是容易学习的。
- 它是一种结构化语言。
- 它产生高效的程序。
- 它可以在多种计算机平台上编译。
- 它是 .Net 框架的一部分。

强大的编程功能

C# 的架构十分接近于传统高级语言 C 和 C++，是一门面向对象的编程语言。它与 Java 非常相似，有许多强大的编程功能，因此得到世界范围内广大程序员的亲睐。

下面列出 C# 一些重要的功能：

- 布尔条件（Boolean Conditions）
- 自动垃圾回收（Automatic Garbage Collection）
- 标准库（Standard Library）
- 组件版本（Assembly Versioning）
- 属性（Properties）和事件（Events）

? 委托（Delegates）和事件管理（Events Management）

? 易于使用的泛型（Generics）

? 索引器（Indexers）

? 条件编译（Conditional Compilation）

? 简单的多线程（Multithreading）

? LINQ 和 Lambda 表达式

? 集成 Windows

环境配置

在这一章中，我们将讨论创建 C# 编程所需要的工具。我们已经提到过 C# 是 .Net 框架的一部分， 且用于编写 .Net 应用程序。因此，在讨论运行一个 C# 程序的可用工具之前，让我们先了解一下 C# 与 .Net 框架之间的关系。

.Net 框架

.Net 框架是一个革命性的平台，能帮您编写出下面类型的应用程序：

- Windows 应用程序
- Web 应用程序
- Web 服务

.Net 框架应用程是多平台的应用程序。框架的设计方式使它适用于下列各种语言：C#、C++、Visual Basic、Jscript、COBOL 等等。所有这些语言可以访问框架，并且彼此之间可以互相交互。

.Net 框架由一个巨大的代码库组成，用于 C# 等客户端语言。下面列出一些 .Net 框架的组件：

- 公共语言运行库（Common Language Runtime – CLR）
- .Net 框架类库（.Net Framework Class Library）
- 公共语言规范（Common Language Specification）
- 通用类型系统（Common Type System）
- 元数据（Metadata）和组件（Assemblies）
- Windows 窗体（Windows Forms）
- ASP.Net 和 ASP.Net AJAX
- ADO.Net
- Windows 工作流基础（Windows Workflow Foundation – WF）
- Windows 显示基础（Windows Presentation Foundation）
- Windows 通信基础（Windows Communication Foundation – WCF）
- LINQ

如需了解每个组件的功能，请参阅 ASP.Net – Introduction ，更详细的信息，请参阅微软（Microsoft）的文档。

C# 的集成开发环境 (IDE)

微软提供了下列用于 C# 编程的开发工具：

- Visual Studio 2010 (VS)
- Visual C# 2010 Express (VCE)
- Visual Web Developer

后面两个是免费使用的，可从微软官方网址下载。使用这些工具，您可以编写各种 C# 程序， 从简单的命令行应用程序到更复杂的应用程序。您也可以使用基本的文本编辑器编写 C# 源代码文件，比如 Notepad，并使用命令行编译器（.NET 框架的一部分），编译代码成组件。

Visual C# Express 和 Visual Web Developer Express 版本是 Visual Studio 的定制版本，且具有相同的外观和感观。它们保留 Visual Studio 的大部分功能。 在本教程中，我们使用的是 Visual C# 2010 Express。

您可以从 Microsoft Visual Studio 上进行下载。它会自动安装在您的机器上。请注意，您在完成速成版的安装时需要提供一个可用的网络连接。

在 Linux 或 Mac OS 上编写 C# 程序

虽然 .NET 框架是运行在 Windows 操作系统上，但是也有一些运行于其它操作系统上的版本可供选择。 Mono 是 .NET 框架的一个开源版本， 它包含了一个 C# 编译器，且可运行于多种操作系统上，比如各种版本对 Linux 和 Mac OS 的支持。如需了解更多详情，请访问 Go Mono 。

Mono 的目的不仅仅是跨平台地运行微软 .NET 应用程序，而且也为 Linux 开发者提供了更好的开发工具。 Mono 可运行在多种操作系统上，包括 Android、BSD、iOS、Linux、OS X、Windows、Solaris 和 UNIX。

程序结构

在我们学习 C# 编程语言的基础构件块之前，让我们先看一下 C# 的最小的程序结构， 以便作为接下来章节的参考。

创建 Hello World 实例

一个 C# 程序主要包括以下部分：

- 命名空间声明
- 一个类
- 类方法
- 类属性
- 一个 Main 方法
- 语句和表达式
- 注释

让我们看一个可以打印出 "Hello World" 的简单的代码：

```
using System;
namespace HelloWorldApplication
{
  class HelloWorld
  {
    static void Main(string[] args)
    {
      /* 我的第一个 C# 程序*/
      Console.WriteLine("Hello World");
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Hello World
```

让我们看一下上面给出程序的各个部分：

- 程序的第一行 using System;-using 关键字用于在程序中包含 System 命名空间。 一个程序一般有多个 using 语句。
- 下一行是 namespace 声明。一个 namespace 是一系列的类。*HelloWorldApplication* 命名空间包含了类 *HelloWorld* 。
- 下一行是 class 声明。类 *HelloWorld* 包含了程序所使用的数据和方法的声明。类一般包含多个方法。方法定义了类的行为。 在这里，*HelloWorld* 类只有一个 Main 方法。
- 下一行定义了 Main 方法，是所有 C# 程序的**入口**。Main 方法说明当类执行时，它将做什么动作。
- 下一行 /*...*/ 将会被编译器忽略，且它会在程序中添加**注释** 。
- Main 方法通过语句 Console.WriteLine("Hello World"); 指定了它的行为。 *WriteLine* 是一个定义在 *System* 命名空间中的 *Console* 类的一个方法。该语句会在屏幕上显示消息 "Hello, World!" 。
- 最后一行 Console.ReadKey(); 是针对 VS.NET 用户的。这使得程序会等待一个用户按键的动作，防止程序在 Visual Studio .NET 启动时屏幕会快速运行并关闭。

以下几点值得注意：

- C# 是大小写敏感的。
- 所有的语句和表达式必须以分号（;）结尾。
- 程序的执行从 Main 方法开始。
- 与 Java 不同的是，文件名可以不同于类的名称。

编译执行 C# 程序

如果您使用 Visual Studio.Net 编译和执行 C# 程序，请按下面的步骤进行：

- 启动 Visual Studio。
- 在菜单栏上，选择 File -> New -> Project。
- 从模板中选择 Visual C#，然后选择 Windows。
- 选择 Console Application。
- 为您的项目制定一个名称，然后点击 OK 按钮。
- 新项目会出现在解决方案资源管理器中。
- 在代码编辑器中编写代码。
- 点击 Run 按钮或者按下 F5 键来运行程序。会出现一个命令提示符窗口，显示 Hello World。

您也可以使用命令行代替 Visual Studio IDE 来编译 C# 程序：

- 打开一个文本编辑器，添加上面提到的代码。
- 保存文件为 helloworld.cs。
- 打开命令提示符工具，定位到文件所保存的目录。
- 键入 csc helloworld.cs 并按下回车键来编译代码。
- 如果代码没有错误，命令提示符会进入下一行，并生成 helloworld.exe 可执行文件。
- 接下来，键入 helloworld 来执行程序。
- 您将看到 "Hello World" 打印在屏幕上。

基本语法

C# 是一种面向对象的编程语言。在面向对象的程序设计方法中，程序由各种相互作用的对象组成。一个对象采取的动作称为方法。 相同种类的对象通常具有相同的属性，或者说，是在相同的类中。

例如，以 Rectangle（矩形）对象为例。它具有 length 和 width 属性。 根据设计，它可能需要接受这些属性值、计算面积和显示细节的方法。

让我们来看看一个 Rectangle（矩形）类的实现，并借此讨论 C# 的基本语法：

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      // member variables
      double length;
      double width;
      public void Acceptdetails()
      {
         length = 4.5;
         width = 3.5;
      }
      public double GetArea()
      {
         return length * width;
      }
      public void Display()
      {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }

   class ExecuteRectangle
   {
      static void Main(string[] args)
      {
         Rectangle r = new Rectangle();
         r.Acceptdetails();
         r.Display();
         Console.ReadLine();
      }
   }
}
```

编译执行上述代码，得到如下结果：

```
Length: 4.5
Width: 3.5
Area: 15.75
```

using 关键字

在任何 C# 程序中的第一条语句都是：

```
using System;
```

using 关键字用于在程序中包含命名空间。一个程序可以包含多个 using 语句。

class 关键字

class 关键字用于声明一个类。

C# 中的注释

注释用于解释代码。编译器会忽略注释的部分。在 C# 程序中，多行注释以 /* 开始，并以字符 */ 终止，如下所示：

```
/* This program demonstrates
The basic syntax of C# programming
Language */
```

单行注释是用 '//' 符号表示。例如：

```
}//end class Rectangle
```

成员变量

变量是一个类的属性或数据成员，用于存储数据。在上面的程序中，*Rectangle* 类有两个成员变量，名为 *length* 和 *width* 。

成员函数

函数是一系列执行特定任务的语句。类的成员函数是在类内声明的。我们举例的类 Rectangle 包含了三个成员函数： *AcceptDetails*、*GetArea* 和 *Display* 。

实例化一个类

在上面的程序中，类 *ExecuteRectangle* 是一个包含 *Main()* 方法和实例化 *Rectangle* 类的类。

标识符

标识符是用来识别类、变量、函数或任何其它用户定义的项目。在 C# 中，类的命名必须遵循如下基本规则：

- 标识符必须以字母开头，后面可以跟一系列的字母、数字（0 - 9）或下划线（_）。标识符中的第一个字符不能是数字。
- 标识符必须不包含任何嵌入的空格或符号，比如 `?-+!@#%^&*()[]{}.;:"'/\` 。但是，可以使用下划线（_）。
- 标识符不能是 C# 关键字。

C# 关键字

关键字是 C# 编译器预定义的保留字。这些关键字不能用作标识符，但是，如果您想使用这些关键字作为标识符，可以在关键字前面加上 @ 字符作为前缀。

在 C# 中，有些标识符在代码的上下文中有特殊的意义，如 get 和 set，这些被称为上下文关键字。

下表列出了 C# 中的保留关键字（Reserved Keywords）和上下文关键字（Contextual Keywords）：

保留关键字						
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	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint

保留关键字						
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	while					
上下文关键字						
add	alias	ascending	descending	dynamic	from	get
global	group	into	join	let	orderby	partial (type)
partial(method)	remove	select	set			

数据类型

在 C# 中，变量分为以下几种类型：

? 值类型（Value types）

? 引用类型（Reference types）

? 指针类型（Pointer types）

值类型

值类型变量可以直接分配给其一个值。它们是从类 System.ValueType 中派生的。

值类型直接包含数据。比如 int、char、float，它们分别存储数字、字母和浮点数。当您声明一个 int 类型的变量时，系统将会分配内存来存储它的值。

下表列出了 C# 2010 中可用的值类型：

类型	描述	范围	默认值
bool	布尔值	True 或 False	False
byte	8 位无符号整数	0 到 255	0
char	16 位 Unicode 字符	U +0000 到 U +ffff	'\0'
decimal	128 位精确的十进制值，28−29 有效位数	(−7.9 x 1028 到 7.9 x 1028) / 100 到 28	0.0M
double	64 位双精度浮点型	(+/−) 5.0 x 10−324 到 (+/−) 1.7 x 10308	0.0D
float	32 位单精度浮点型	−3.4 x 1038 到 + 3.4 x 1038	0.0F
int	32 位有符号整数类型	−2 , 147 , 483 , 648 到 2 , 147 , 483 , 647	0
long	64 位有符号整数类型	−923 , 372 , 036 , 854 , 775 , 808 到 9 , 223 , 372 , 036 , 854 , 775 , 807	0L
sbyte	8 位有符号整数类型	−128 到 127	0
short	16 位有符号整数类型	−32 , 768 到 32 , 767	0
uint	32 位无符号整数类型	0 到 4 , 294 , 967 , 295	0
ulong	64 位无符号整数类型	0 到 18 , 446 , 744 , 073 ,709 ,551 , 615	0
ushort	16 位无符号整数类型	0 到 65 , 535	0

如需得到一个类型或一个变量在特定平台上的准确大小，可以使用 sizeof 方法。表达式 *sizeof(type)* 产生以字节为单位存储对象或类型的存储尺寸。下面举例获取任何机器上 *int* 类型的存储大小：

```
namespace DataTypeApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Size of int: {0}", sizeof(int));
      Console.ReadLine();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Size of int: 4
```

引用类型

引用类型不包含存储在变量中的实际数据，但它们包含对变量的引用。

换句话说，它们指向的是一个内存位置。使用多重变量时，引用类型可以指向一个内存位置。 如果内存位置的数据是由多重变量之中的一个改变的，其他变量会自动相应这种值的变化。 例如，**内置的**引用类型有：object、dynamic 和 string。

对象类型

对象类型是 C# 通用类型系统（CTS）中所有数据类型基类。 Object 是 System.Object 类的别名。 对象类型可以被分配任何其他类型（值类型、引用类型、预定义类型或用户自定义类型）的值。但是，在分配值之前，需要先进行类型转换。

当一个值类型转换为对象类型时，则被称为**装箱**；另一方面，当一个对象类型转换为值类型时，则被称为**拆箱**。

```
object obj;
obj = 100; // 这是装箱
```

动态类型

您可以在动态数据类型变量中存储任何类型的值。这些变量的类型检查是在运行时进行的。

声明动态类型的语法：

```
dynamic <variable_name> = value;
```

例如：

```
dynamic d = 20;
```

动态类型与对象类型相似，但是对象类型变量的类型检查是在编译时进行的，而动态类型变量的类型检查是在运行时进行的。

字符串类型

字符串 类型允许您给变量分配任何字符串值。字符串类型是 System.String 类的别名。它是从对象类型派生的。字符串类型的值可以通过两种形式进行分配：引号和 @ 引号。

例如：

```
String str = "Tutorials Point";
```

一个 @ 引号字符串：

```
@"Tutorials Point";
```

用户自定义引用类型有：class、interface 或 delegate。我们将在以后的章节中讨论这些类型。

指针类型

指针类型变量存储另一种类型的内存地址。C# 中的指针与 C 或 C++ 中的指针有相同的功能。

声明指针类型的语法：

```
type* identifier;
```

例如：

```
char* cptr;
int* iptr;
```

我们将在章节"不安全的代码"中讨论指针类型。

类型转换

类型转换是把数据从一种类型转换为另一种类型。在 C# 中，类型转换有两种形式：

- **隐式类型转换** 这些转换是 C# 默认的以安全方式进行的转换。例如，从小的整数类型转换为大的整数类型，从派生类转换为基类。
- **显式类型转换** 这些转换是通过用户使用预定义的函数显示完成的。显式转换需要强制转换运算符。

下面的实例显示了一个显式的类型转换：

```
namespace TypeConversionApplication
{
   class ExplicitConversion
   {
      static void Main(string[] args)
      {
         double d = 5673.74;
         int i;

         // 强制转换 double 为 int
         i = (int)d;
         Console.WriteLine(i);
         Console.ReadKey();

      }
   }
}
```

编译执行上述代码，得到如下结果：

```
5673
```

C# 类型转换方法

C# 提供了下列内置的类型转换方法：

序号	方法与描述
1	ToBoolean 如果可能的话，把类型转换为布尔型。
2	ToByte 把类型转换为字节类型。

序号	方法与描述
3	ToChar 如果可能的话，把类型转换为单个 Unicode 字符类型。
4	ToDateTime 把类型（整数或字符串类型）转换为日期-时间结构。
5	ToDecimal 把浮点型或整数类型转换为十进制类型。
6	ToDouble 把类型转换为双精度浮点型。
7	ToInt16 把类型转换为 16 位整数类型。
8	ToInt32 把类型转换为 32 位整数类型。
9	ToInt64 把类型转换为 64 位整数类型。
10	ToSbyte 把类型转换为有符号字节类型。
11	ToSingle 把类型转换为小浮点数类型。
12	ToString 把类型转换为字符串类型。
13	ToType 把类型转换为指定类型。
14	ToUInt16 把类型转换为 16 位无符号整数类型。
15	ToUInt32 把类型转换为 32 位无符号整数类型。
16	ToUInt64 把类型转换为 64 位无符号整数类型。

下面的实例把不同值类型变量转换为字符串类型变量：

```
namespace TypeConversionApplication
{
   class StringConversion
   {
      static void Main(string[] args)
      {
         int i = 75;
         float f = 53.005f;
         double d = 2345.7652;
         bool b = true;

         Console.WriteLine(i.ToString());
```

```
            Console.WriteLine(f.ToString());
            Console.WriteLine(d.ToString());
            Console.WriteLine(b.ToString());
            Console.ReadKey();

        }
    }
}
```

编译执行上述代码，得到如下结果：

```
75
53.005
2345.7652
True
```

变量

一个变量只不过是一个供程序操作的存储区的名字。在 C# 中，每个变量都有一个特定的类型，类型决定了变量的内存大小、布局、可以存储在内存中的值的范围以及可以对变量进行的一系列操作。

C# 中提供的基本的值类型大致可以分为以下几类：

类型	举例
整数类型	sbyte、byte、short、ushort、int、uint、long、ulong 和 char
浮点型	float 和 double
十进制类型	decimal
布尔类型	true 或 false 值，指定的值
空类型	可为空值的数据类型

C# 允许定义其他值类型的变量，比如 enum，也允许定义引用类型变量，比如 class。这些我们将在以后的章节中进行讨论。

变量定义

C# 中变量定义的语法：

```
<data_type> <variable_list>;
```

在这里，data_type 必须是一个有效的 C# 数据类型，可以是 char、int、float、double 或其他用户自定义的数据类型。variable_list 可以由一个或多个用逗号分隔的标识符名称组成。

一些有效的变量定义如下所示：

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

您可以在定义一个变量时对其进行初始化：

```
int i = 100;
```

变量初始化

变量通过等号后的一个常量表达式进行初始化（也就是赋值）。初始化的一般语法为：

```
variable_name = value;
```

变量可以在声明时被初始化。初始化由等号后的一个常量表达式完成，如下所示：

```
<data_type> <variable_name> = value;
```

一些实例：

```
int d = 3, f = 5;    /* 初始化 d 和 f. */
byte z = 22;         /* 初始化 z. */
double pi = 3.14159; /* 声明 pi 的近似值 */
char x = 'x';        /* 变量 x 的值为 'x' */
```

适时地初始化变量是一个良好的编程习惯，否则有时程序会产生意想不到的结果。

请看下面的实例，使用了各种类型的变量：

```
namespace VariableDefinition
{
   class Program
   {
      static void Main(string[] args)
      {
         short a;
         int b ;
         double c;

         /* 实际初始化 */
         a = 10;
         b = 20;
         c = a + b;
         Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
         Console.ReadLine();
      }
   }
}
```

编译执行上述代码，得到如下结果：

```
a = 10, b = 20, c = 30
```

接受来自用户的值

System 命名空间中的 Console 类提供了一个函数 ReadLine()，用于接收来自用户的输入，并把它存储到一个变量中。 例如：

```
int num;
num = Convert.ToInt32(Console.ReadLine());
```

函数 Convert.ToInt32() 把用户输入的数据转换为 int 数据类型，因为函数 Console.ReadLine() 只接受字符串格式的数据。

C# 中的 Lvalues 和 Rvalues

在 C# 中的两种表达式：

- lvalue：lvalue 表达式可以出现在赋值语句的左边或右边。
- rvalue：rvalue 表达式可以出现在赋值语句的右边，不能出现在赋值语句的左边。

变量是 lvalue 的，所以可以出现在赋值语句的左边。数值是 rvalue 的，因此不能被赋值，也不能出现在赋值语句的左边。下面是一个有效的语句：

```
int g = 20;
```

下面是一个无效的语句，会在编译时产生错误：

```
10 = 20;
```

常量和文字

常量是固定值，程序执行期间不会改变。这些固定值也被称为文字。常量可以是任何基本数据类型，如整数常量，浮点常量，字符常量或者字符串常量，还有枚举常量。

常量可以被当作常规的变量，只是它们的值在定义后不能被修改。

整数常量

整数常量可以是十进制、八进制、或十六进制的常量。前缀指定基或基数：0x 或 0X 表示十六进制，0 表示八进制，没有前缀则表示十进制。

整数常量也可以有后缀，可以是 U 和 L 的组合，其中 U 和 L 分别表示 unsigned 和 long。后缀可以是大写或小写，多个后缀以任意顺序进行组合。

这里是一些整数常量的例子：

```
212         /* 合法 */
215u        /* 合法 */
0xFeeL      /* 合法 */
078         /* 非法: 8 不是一个八进制数字 */
032UU       /* 非法: 不能重复后缀 */
```

以下是各种类型的整数常量的实例：

```
85         /* 十进制 */
0213       /* 八进制 */
0x4b       /* 十六进制 */
30         /* int */
30u        /* 无符号 int */
30l        /* long */
30ul       /* 无符号 long */
```

浮点常量

一个浮点常量是由整数部分，小数点，小数部分和指数部分组成。您可以使用小数形式或指数形式来表示浮点常量。

这里是一些浮点常量的例子：

```
3.14159      /* 合法 */
314159E-5L   /* 合法 */
510E         /* 非法: 不完全指数 */
210f         /* 非法: 没有小数或指数 */
.e55         /* 非法: 缺少整数或小数 */
```

使用小数形式表示时，必须包含小数点、指数或同时包含两者。使用指数形式表示时，必须包含整数部分、小数部分或同时包含两者。有符号的指数是用 e 或 E 表示的。

字符常量

字符常量是括在单引号里，例如 'x'，并可以存储在一个简单的字符类型变量中。一个字符常量可以是一个普通字符（例如 'x'）、一个转义序列（例如 '\t'）或一个通用字符（例如 '\u02C0'）。

在 C# 中有一些特定的字符，当它们的前面带有反斜杠时有特殊的意义，可用于表示换行符（\n）或制表符（\t）。在这里，列出一些转义序列码：

转义序列	含义
\	\ 字符
\'	' 字符
\"	" 字符
\?	? 字符
\a	Alet 或 bell
\b	退格键
\f	换页符
\n	换行符
\r	回车
\t	水平制表符
\v	垂直制表符
\ooo	一到三位的八进制数
\xhh...	一个或多个数字的十六进制数

下面的实例说明了一些转义字符序列：

```
using System;
namespace EscapeChar
{
  class Program
  {
    static void Main(string[] args)
    {
```

```
        Console.WriteLine("Hello\tWorld\n\n");
        Console.ReadLine();
      }
   }
}
```

编译执行上述代码，得到如下结果：

```
Hello   World
```

字符串常量

字符串常量是括在双引号""里，或者是括在@""里。字符串常量包含的字符类似于字符常量，可以是普通字符、转义序列和通用字符。

使用字符串常量时，可以把一个很长的行拆成多个行，可以使用空格分隔各个部分。

这里是一些字符串常量的例子。下面所列的各种形式表示相同的字符串。

```
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
@"hello dear"
```

定义常量

常量是使用 const 关键字来定义的。定义一个常量的语法如下：

```
const <data_type> <constant_name> = value;
```

下面的代码演示了如何在程序中定义和使用常量：

```
using System;
namespace DeclaringConstants
{
   class Program
   {
      static void Main(string[] args)
      {
         const double pi = 3.14159;  // 常量声明
         double r;
         Console.WriteLine("Enter Radius: ");
```

```
            r = Convert.ToDouble(Console.ReadLine());
            double areaCircle = pi * r * r;
            Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
            Console.ReadLine();
        }
    }
}
```

编译执行上述代码，得到如下结果：

```
Enter Radius:
3
Radius: 3, Area: 28.27431
```

运算符

运算符是一种告诉编译器执行特定的数字或逻辑操作的符号。C# 中有丰富的内置运算符，分类如下：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 其它运算符

本教程将逐一讲解算运算符、关系运算符、逻辑运算符、位运算符、赋值运算符和其他运算符。

算术运算符

下表列出了 C# 支持的所有算术运算符。假设变量 A 的值为10，变量 B 的值为20，则：

算术运算符实例

运算符	描述	实例
+	两个操作数相加	A + B = 30
−	两个操作数相减(第一个减去第二个)	A − B = −10
*	两个操作数相乘	A * B = 200
/	分子除以分母	B / A = 2
%	取模运算符，整除后的余数	B % A = 0
++	自增运算符，整数值增加1	A++ = 11
−−	自减运算符，整数值减少1	A−− = 9

关系运算符

下表列出了 C# 支持的所有关系运算符。假设变量 A 的值为10，变量 B 的值为 20，则：

关系运算符实例

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真	(A == B) 不为真

运算符	描述	实例
!=	检查两个操作数的值是否相等，如果不相等则条件为真	(A != B) 为真
>	检查左边的操作数的值是否大于右操作数的值，如果是则条件为真	(A > B) 不为真
<	检查左边的操作数的值是否小于右操作数的值，如果是则条件为真	(A < B) 为真
>=	检查左边的操作数的值是否大于或等于右操作数的值，如果是则条件为真	(A >= B) 不为真
<=	检查左边的操作数的值是否小于或等于右操作数的值，如果是则条件为真	(A <= B) 为真

逻辑运算符

下表列出了 C# 所支持的所有逻辑运算符。假设变量 A 为布尔值 true，变量 B 为布尔值 false，则：

逻辑运算符实例

运算符	描述	实例
&&	称为逻辑与操作，如果两个操作数都非零，则条件为真	(A && B) 结果为 false
||	称为逻辑或操作，如果两个操作数中有任意一个非零，则条件为真	(A || B) 结果为 true
!	称为逻辑非运算符，用来逆转操作数的逻辑状态。如果条件为真则逻非运算符将使其为假	

位运算符

位运算符作用于位，并逐位执行操作。&、| 和 ^ 的真值表如下：

p	p	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假设，如果 A=60，B=13，现以二进制格式表示如下：

A = 0011 1100

B = 0000 1101

A & B = 0000 1100

A | B = 0011 1101

A ^ B = 0011 0001

~A = 1100 0011

下表列出了 C# 支持的位运算符。假设变量 A 的值为60，变量 B 的值为13 则：

位运算符示例

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(A & B) 将得到 12. 即 0000 1100
|	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A | B) 将得到 61, 即 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(A ^ B) 将得到 49, 即 0011 0001
~	二进制补码运算符是一元运算符，具有“翻转”的位效果。	(~A) 将得到 -61, 即 1100 0011
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240, 即 1111 0000
>>	二进制右移运算符。左操作数的值由右移动右操作数指定的位数。	A >> 2 将得到 15, 即 0000 1111

赋值运算符

下表列出了 C# 支持的赋值运算符：

赋值运算符实例

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给到左边操作数	C = A + B 把 A + B 的值赋给 C
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
*=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C *= A 相当于 C = C * A
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
<<=	左移且赋值运算符	C <<= 2 相当于 C = C << 2

运算符	描述	实例
>>=	右移且赋值运算符	C >>= 2 相当于 C = C >> 2
&=	按位与且赋值运算符	C &= 2 相当于 C = C & 2
^=	按位异或且赋值运算符	C ^= 2 相当于 C = C ^ 2
|=	按位或且赋值运算符	C |= 2 相当于 C = C | 2

其它运算符

下表列出了 C# 支持的其他一些重要的运算符，包括 sizeof，typeof 运算和? :。

其它运算符实例

运算符	描述	示例
sizeof()	返回一个数据类型的大小	sizeof(int)，将返回 4
typeof()	返回一个类的类型	typeof(StreamReader)
&	返回一个变量的地址	&a; 将给出变量的实际地址
*	指针的变量	*a; 将指向一个变量
? :	条件表达式	如果条件为真，那么 ? 值为 X : 否则 ? 值为 Y
is	判断一个对象是否是特定的类型	If(Ford is Car) // 判断 Ford 对象是否属于 Car 类型
as	转换，如果转换失败则引发异常	Object obj = new StringReader("Hello");StringReader r = obj as StringReader;

C# 运算符优先级

运算符优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符比加减运算符的优先级高：

例如 X = 7 + 3 * 2；这里，x 被赋值为 13，而不是 20，因为运算符 * 的优先级高于 +，所以这里首先进行 3*2 运算，然后再加上 7。

下表按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格上面，具有较低优先级的运算符出现在表格下面。在表达式中，较高优先级的运算符会优先被运算。

运算符优先级实例

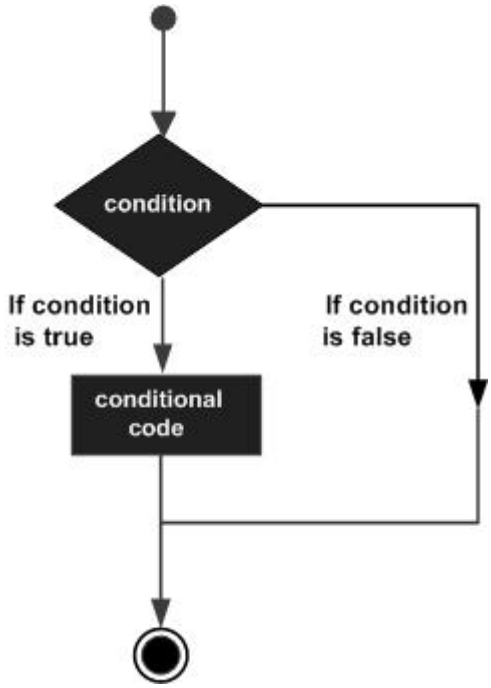
分类	运算符	结合性
后缀	() [] -> . ++ - -	从左到右

分类	运算符	结合性
一元	+ − ! ~ ++ − − (type)* & sizeof	从右到左
乘法	* / %	从左到右
相加	+ −	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
按位与	&	从左到右
按位异或	| ^	从左到右
按位或	|	从左到右
逻辑 AND	&&	从左到右
逻辑 OR	||	从左到右
条件	?:	从右到左
赋值	= += −= *= /= %=>>= <<= &= ^= |=	从右到左
逗号	,	从左到右

判断

判断结构需要程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。

下面是大多数编程语言中典型判断结构的一般形式：



图片 1.1 image

C# 提供了以下类型的判断语句。点击链接查看每个语句的详细信息。

语句	描述
if 语句	一个if 语句由一个布尔表达式后跟一个或多个语句组成。
if...else 语句	一个if 语句后跟一个可选的 else 语句，else 语句在布尔表达式为假时执行。
嵌套 if 语句	您可以在一个 if 或 else if 语句内使用另一个 if 或 else if 语句。
switch 语句	一个 switch 语句允许测试一个变量等于多个值时的情况。
嵌套 switch 语句	您可以在一个 switch 语句内使用另一个 switch 语句。

?：运算符：

我们已经在前面的章节中介绍了**条件运算符 ?：**，可以用来代替 if...else 语句。它的一般形式如下：

```
Exp1 ? Exp2 : Exp3;
```

其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和放置。

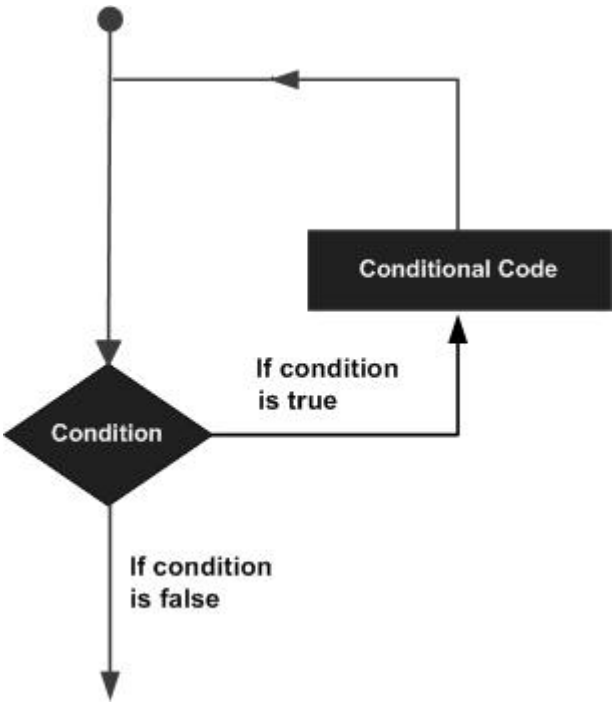
? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个?表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个?表达式的值。

循环

有的情况下，可能需要多次执行同一块代码。一般情况下，语句是顺序执行的：函数中的第一个语句先执行，然后是第二个语句，以此类推。

编程语言提供了允许更为复杂的执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的一般形式：



图片 1.2 image

C# 提供了以下几种类型的循环类型。点击链接查看每个类型的详细信息。

循环类型	描述
while 循环	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
for 循环	多次执行一个语句序列，简化管理循环变量的代码。
do...while 循环	除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
嵌套循环	您可以在 do...while 循环内使用一个或多个循环。

循环控制语句

循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围内创建的自动对象都会被销毁。

C# 提供了以下控制语句。点击链接查看每个语句的详细信息。

控制语句	描述
break 语句	终止 loop 或 switch 语句，程序流将继续执行紧接着 loop 或 switch 的下一条语句。
continue 语句	引起循环跳过主体的剩余部分，立即重新开始测试条件。

无限循环

如果条件永远不为假，则循环将变为无限循环。for 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。

举例

```
using System;
namespace Loops
{
  class Program
  {
    static void Main(string[] args)
    {
      for (; ; )
      {
        Console.WriteLine("Hey! I am Trapped");
      }
    }
  }
}
```

当条件表达式不存在时，它被假定为真。您也可以设置一个初始值和增量表达式，但一般情况下，程序员偏向于使用 for(;;) 结构来表示一个无限循环。

封装

封装被定为义为“把一个或多个项目封闭在一个物理的或者逻辑的包中”。在面向对象程序设计方法论中，封装是为了防止对实现细节的访问。

抽象和封装是面向对象程序设计的相关特性。抽象允许相关信息可视化，封装使程序员*实现所需级别的抽象*。

封装使用**访问修饰符**来实现。一个**访问修饰符**定义了一个类成员的范围和可见性。C# 支持的访问修饰符如下所示：

- Public
- Private
- Protected
- Internal
- Protected internal

Public 访问修饰符

Public 访问修饰符允许一个类将其成员变和成员函数暴露给其他的函数和对象。任何公有成员可以被外部的类访问。

下面的例子说明了这点：

```
using System;
namespace RectangleApplication
{
  class Rectangle
  {
    //成员变量
    public double length;
    public double width;

    public double GetArea()
    {
      return length * width;
    }
    public void Display()
    {
```

```
      Console.WriteLine("长度: {0}", length);
      Console.WriteLine("宽度: {0}", width);
      Console.WriteLine("面积: {0}", GetArea());
    }
  }//end class Rectangle

  class ExecuteRectangle
  {
    static void Main(string[] args)
    {
      Rectangle r = new Rectangle();
      r.length = 4.5;
      r.width = 3.5;
      r.Display();
      Console.ReadLine();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
长度: 4.5
宽度: 3.5
面积: 15.75
```

在上面的例子中，成员变量 length 和 width 被声明为 public，所以它们可以被函数 Main() 使用 Rectangle 类的实例 r 访问。

成员函数 *Display()* 和 *GetArea()* 也可以不通过类的实例直接直接访问这些变量。

成员函数 *Display()* 也被声明为 public，所以从它也能被 *Main()* 使用 Rectangle 类的实例 r 访问。

Private 访问修饰符

Private 访问修饰符允许一个类将其成员变量和成员函数对其他的函数和对象进行隐藏。只有同一个类中的函数可以访问它的私有成员。即使是类的实例也不能访问它的私有成员。

下面的例子说明了这点：

```
using System;
namespace RectangleApplication
{
  class Rectangle
  {
```

```
    //成员变量
    private double length;
    private double width;

    public void Acceptdetails()
    {
      Console.WriteLine("请输入长度: ");
      length = Convert.ToDouble(Console.ReadLine());
      Console.WriteLine("请输入宽度: ");
      width = Convert.ToDouble(Console.ReadLine());
    }
    public double GetArea()
    {
      return length * width;
    }
    public void Display()
    {
      Console.WriteLine("长度: {0}", length);
      Console.WriteLine("宽度: {0}", width);
      Console.WriteLine("面积: {0}", GetArea());
    }
  }//end class Rectangle

  class ExecuteRectangle
  {
    static void Main(string[] args)
    {
      Rectangle r = new Rectangle();
      r.Acceptdetails();
      r.Display();
      Console.ReadLine();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
请输入长度:
4.4
请输入宽度:
3.3
长度: 4.4
宽度: 3.3
面积: 14.52
```

在上面的例子中，成员变量 length 和 width 被声明为 private，所以它们不能被函数 Main() 访问。成员函数 *AcceptDetails()* 和 *Display()* 可以访问这些变量。由于成员函数 *AcceptDetails()* 和 *Display()* 被声明为 public，所以它们可以被 *Main()* 函数使用 Rectangle 类的实例 r 访问。

Protected 访问修饰符

Protected 访问修饰符允许子类访问它的基类的成员变量和成员函数。这种方式有助于实现继承。我们将在继承的章节详细讨论这个问题。

Internal 访问修饰符

Internal 访问修饰符允许一个类将其成员变量和成员函数暴露给当前程序中的其他函数和对象。换句话说，带有 Internal 访问修饰符的任何成员可以被定义在该成员所定义的应用程序内的任何类或方法访问。

下面的例子说明了这点：

```
using System;
namespace RectangleApplication
{
  class Rectangle
  {
    //成员变量
    internal double length;
    internal double width;

    double GetArea()
    {
      return length * width;
    }
    public void Display()
    {
      Console.WriteLine("长度: {0}", length);
      Console.WriteLine("宽度: {0}", width);
      Console.WriteLine("面积: {0}", GetArea());
    }
  }//end class Rectangle

  class ExecuteRectangle
  {
    static void Main(string[] args)
    {
```

```
        Rectangle r = new Rectangle();
        r.length = 4.5;
        r.width = 3.5;
        r.Display();
        Console.ReadLine();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
长度: 4.5
宽度: 3.5
面积: 15.75
```

在上面的例子中，请注意成员函数 *GetArea()* 声明的时候不带有任何访问修饰符。如果没有指定访问修饰符，则使用类成员的默认访问修饰符，即为 private。

Protected internal 访问修饰符

Protected internal 访问修饰符允许一个类将其成员变量和成员函数对同一应用程内的子类以外的其他的类对象和函数进行隐藏。这也被用于实现继承。

方法

方法是一组在一起执行任务的语句。每个 C# 程序都至少有一个含有方法的类，名为 Main。
若要使用方法，您需要：

- 定义一个方法
- 调用方法

在 C# 中定义方法

当你定义一个方法时，你基本上要声明其结构的组成元素。在 C# 中定义方法的语法如下所示：

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
  Method Body
}
```

以下是方法中的各种元素：

- **访问说明符**：它用于从一个类中确定一个变量或方法的可见性。
- **返回类型**：方法可能会返回一个值。返回类型是方法返回值的数据类型。如果该方法不返回任何值，那么返回类型是 void。
- **方法名称**：方法名称是唯一的标识符，并区分大小写。它不能与在类中声明的任何其他标识符相同。
- **参数列表**：括号括起来，使用参数从方法中传递和接收数据。参数列表是指类型、顺序和方法的参数数目。参数是可选的；方法可能包含任何参数。
- **方法主体**：它包含的一组指令完成所需要的活动所需。

示例
下面的代码段显示了一个函数 FindMax，从两个整数值中，返回其中较大的一个。它具有公共访问说明符，所以它可以通过使用类的外部例子来访问。

```
class NumberManipulator
{
  public int FindMax(int num1, int num2)
  {
    /* local variable declaration */
    int result;
```

```
    if (num1 > num2)
      result = num1;
    else
      result = num2;

    return result;
  }
  ...
}
```

在 C# 中调用方法

你可以使用方法的名称来调用方法。下面的示例说明了这一点：

```
using System;
namespace CalculatorApplication
{
  class NumberManipulator
  {
    public int FindMax(int num1, int num2)
    {
      /* local variable declaration */
      int result;

      if (num1 > num2)
        result = num1;
      else
        result = num2;
      return result;
    }
    static void Main(string[] args)
    {
      /* local variable definition */
      int a = 100;
      int b = 200;
      int ret;
      NumberManipulator n = new NumberManipulator();

      //calling the FindMax method
      ret = n.FindMax(a, b);
      Console.WriteLine("Max value is : {0}", ret );
      Console.ReadLine();
    }
```

```
  }
}
```

编译执行上述代码，得到如下结果：

```
Max value is : 200
```

你也可以通过使用类的实例来从其他类中调用公开方法。

例如，FindMax 它属于 NumberManipulator 类中的方法，你可以从另一个类测试中调用它。

```
using System;
namespace CalculatorApplication
{
  class NumberManipulator
  {
    public int FindMax(int num1, int num2)
    {
      /* local variable declaration */
      int result;

      if(num1 > num2)
        result = num1;
      else
        result = num2;

      return result;
    }
  }

  class Test
  {
    static void Main(string[] args)
    {
      /* local variable definition */
      int a = 100;
      int b = 200;
      int ret;
      NumberManipulator n = new NumberManipulator();

      //calling the FindMax method
      ret = n.FindMax(a, b);
      Console.WriteLine("Max value is : {0}", ret );
      Console.ReadLine();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Max value is : 200
```

递归方法调用

有一种方法可以调用本身。这就是所谓的递归。下面是使用递归函数计算一个给定数字阶乘的示例：

```
using System;
namespace CalculatorApplication
{
  class NumberManipulator
  {
    public int factorial(int num)
    {
      /* local variable declaration */
      int result;
      if (num == 1)
      {
        return 1;
      }
      else
      {
        result = factorial(num - 1) * num;
        return result;
      }
    }

    static void Main(string[] args)
    {
      NumberManipulator n = new NumberManipulator();
      //calling the factorial method
      Console.WriteLine("Factorial of 6 is : {0}", n.factorial(6));
      Console.WriteLine("Factorial of 7 is : {0}", n.factorial(7));
      Console.WriteLine("Factorial of 8 is : {0}", n.factorial(8));
      Console.ReadLine();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Factorial of 6 is: 720
Factorial of 7 is: 5040
Factorial of 8 is: 40320
```

将参数传递给方法

当调用带参数的方法时，您需要将参数传递给该方法。有三种方法，可以将参数传递给方法：

方法	描述
值参数	此方法将参数的实际值复制到该函数的形参。在这种情况下，对该参数在函数内部所做的更改没有对参数产生影响。
引用参数	此方法将对实参的内存位置的引用复制到形参。这意味着对参数所做的更改会影响参数本身。
输出参数	这种方法有助于返回多个值。

可空类型

C # 提供了一个特殊的数据类型，**可空类型**，可以在其中指定正常范围值，以及空 (null) 值。

例如，在一个可空 变量中，你可以从 −2,147,483,648 到 2,147,483,647 或空值中存储任意值。

同样，你可以指定 true，false 或 null 的 Nullable 变量。声明一个**可空类型** (Nullable) 的语法如下：

```
< data_type> ? <variable_name> = null;
```

下面的例子演示了使用可空数据类型：

```
using System;
namespace CalculatorApplication
{
  class NullablesAtShow
  {
    static void Main(string[] args)
    {
      int? num1 = null;
      int? num2 = 45;
      double? num3 = new double?();
      double? num4 = 3.14157;

      bool? boolval = new bool?();

      // display the values

      Console.WriteLine("Nullables at Show: {0}, {1}, {2}, {3}", num1, num2, num3, num4);
      Console.WriteLine("A Nullable boolean value: {0}", boolval);
      Console.ReadLine();
    }
  }
}
```

编译运行上述代码，得到以下结果：

```
Nullables at Show: , 45, , 3.14157
A Nullable boolean value:
```

空合并运算符(??)

空合并运算符是用于空值类型和引用类型。它是用于一个操作数转换为另一种可为空的(或不为空)值类型的操作数，其中，隐式转换是可能的类型。

如果第一个操作数的值为 null，则该运算符返回第二个操作数的值，否则返回第一个操作数的值。下面的例子说明了这一点：

```
using System;
namespace CalculatorApplication
{
  class NullablesAtShow
  {
    static void Main(string[] args)
    {
      double? num1 = null;
      double? num2 = 3.14157;
      double num3;
      num3 = num1 ?? 5.34;
      Console.WriteLine(" Value of num3: {0}", num3);
      num3 = num2 ?? 5.34;
      Console.WriteLine(" Value of num3: {0}", num3);
      Console.ReadLine();
    }
  }
}
```

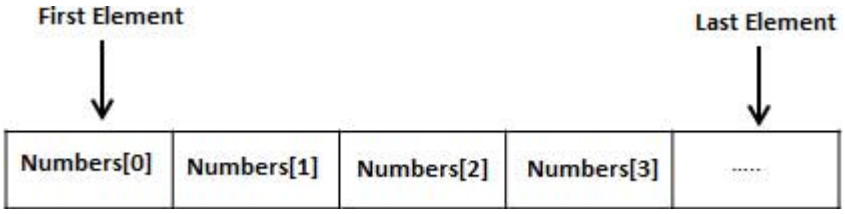
编译运行上述代码，得到以下结果：

```
Value of num3: 5.34
Value of num3: 3.14157
```

数组

数组存储一个大小固定的顺序集合中相同类型的元素。数组用于存储数据的集合，但我们通常认为数组是一个存储在连续的内存位置的相同类型的集合。

相反，声明单个变量，如 number0, number1, ..., 和 number99，声明一个数组变量，如 numbers[0], numbers[1],…, 和 numbers[99] 表示单个变量。在数组的特定元素由一个索引进行访问。
所有数组都由连续的内存位置构成。最低的地址对应于第一元素，最高地址为最后一个元素地址。



图片 1.3 image

声明数组

要在 C# 中声明数组，可以使用下面的语法：

```
datatype[] arrayName;
```

这里，

- *datatype 用于指定要被存储在数组中的元素的类型*
- *[ ] 指定数组的大小*
- *arrayName 指定数组的名称*

例如，

```
double[] balance;
```

初始化数组

声明没有在存储器初始化的数组。当数组变量初始化时，您可以赋值给数组。
数组是引用类型，所以需要使用 new 关键字来创建数组的一个实例。
例如，

```
double[] balance = new double[10];
```

赋值数组

通过使用索引号，可以将值指派给单独的数组元素，比如：

```
double[] balance = new double[10];
balance[0] = 4500.0;
```

你可以在声明数组的同时给它赋值，如下：

```
double[] balance = { 2340.0, 4523.69, 3421.0};
```

你也可以创建和初始化一个数组，如下：

```
int [] marks = new int[5]  { 99,  98, 92, 97, 95};
```

你也可以省略数组的长度，如下：

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
```

你可以将一个数组变量赋给另一个目标。这种情况，两个数组都指向同一内存地址。

```
int [] marks = new int[]  { 99,  98, 92, 97, 95};
int[] score = marks;
```

当你创建一个数组时，C# 编译器初始化每个数组元素为数组类型的默认值。对于 int 数组的所有元素都初始化为 0。

访问数组元素

一个元素由索引数组名访问。这是通过放置在数组名后面的方括号里的元素索引完成的。例如：

```
double salary = balance[9];
```

以下是一个例子，将使用所有上述三个概念即声明，分配和访问数组：

```
using System;
namespace ArrayApplication
{
  class MyArray
  {
    static void Main(string[] args)
    {
      int []  n = new int[10]; /* n is an array of 10 integers */
      int i,j;
```

```
        /* initialize elements of array n */
        for ( i = 0; i < 10; i++ )
        {
          n[ i ] = i + 100;
        }

        /* output each array element's value */
        for (j = 0; j < 10; j++ )
        {
          Console.WriteLine("Element[{0}] = {1}", j, n[j]);
        }
        Console.ReadKey();
      }
   }
}
```

编译运行上述代码，得到以下结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

使用 foreach 循环

在前面的例子中，我们已经使用了一个 for 循环用于访问每个数组元素。还可以使用 foreach 语句来遍历数组。

```
using System;
namespace ArrayApplication
{
  class MyArray
  {
    static void Main(string[] args)
    {
      int []  n = new int[10]; /* n is an array of 10 integers */

      /* initialize elements of array n */
      for ( int i = 0; i < 10; i++ )
      {
        n[i] = i + 100;
      }

      /* output each array element's value */
      foreach (int j in n )
      {
        int i = j-100;
        Console.WriteLine("Element[{0}] = {1}", i, j);
        i++;
      }
```

```
        Console.ReadKey();
    }
  }
}
```

编译运行上述代码，得到以下结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

数组详解

数组在 C# 中是很重要的，应该需要很多更详细的解释。下列有关数组的几个重要概念，C# 程序员应当清楚：

概念	描述
**多维数组 **	C# 支持多维数组。多维数组的最简单的形式是二维数组
**锯齿状数组 **	C# 支持多维数组，这是数组的数组
通过数组到函数	可以通过指定数组的名称没有索引传递给函数的指针数组
参数数组	这是用于使未知数量的参数传到函数
Array 类	定义在系统命名空间中，它是基类所有的数组，并使用数组提供了各种属性和方法

字符串

在 C# 中，可以使用字符串作为字符数组，但更常见的做法是使用 string 关键字来声明一个字符串变量。该 string 关键字是 System.String 类的别名。

创建一个 String 对象

可以使用下列方法之一字符串对象：

- 通过指定一个字符串给一个字符串变量
- 通过使用 String 类的构造函数
- 通过使用字符串连接运算符(+)
- 通过检索属性或调用返回一个字符串的方法
- 通过调用格式化方法的值或对象转换成它的字符串表示

下面的例子说明了这一点：

```
using System;
namespace StringApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      //from string literal and string concatenation
      string fname, lname;
      fname = "Rowan";
      lname = "Atkinson";

      string fullname = fname + lname;
      Console.WriteLine("Full Name: {0}", fullname);

      //by using string constructor
      char[] letters = { 'H', 'e', 'l', 'l','o' };
      string greetings = new string(letters);
      Console.WriteLine("Greetings: {0}", greetings);

      //methods returning string
      string[] sarray = { "Hello", "From", "Tutorials", "Point" };
      string message = String.Join(" ", sarray);
      Console.WriteLine("Message: {0}", message);

      //formatting method to convert a value
      DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
      string chat = String.Format("Message sent at {0:t} on {0:D}", waiting);
      Console.WriteLine("Message: {0}", chat);
    }
```

```
  }
}
```

编译执行上述代码，得到如下结果：

```
Full Name: Rowan Atkinson
Greetings: Hello
Message: Hello From Tutorials Point
Message: Message sent at 5:58 PM on Wednesday, October 10, 2012
```

String 类的属性

String 类有以下两个属性：

序号	属性名称和描述
1	Chars 获取在当前字符串对象中的指定位置的字符对象
2	Length 获取字符在当前字符串对象的数目

String 类的方法

String 类有许多方法，以帮助使用 String 对象。下表提供了一些最常用的方法：

序号	属性名称和描述
1	public static int Compare(string strA,string strB) 比较两个指定的字符串对象，并返回一个整数，指示其在排序顺序相对位置
2	public static int Compare(string strA,string strB,bool ignoreCase) 比较两个指定的字符串对象，并返回一个整数，指示其在排序顺序相对位置。但是，它忽略情况下，如果布尔参数为true
3	public static string Concat(string str0,string str1) 连接两个字符串对象
4	public static string Concat(string str0,string str1,string str2) 拼接三个字符串对象
5	public static string Concat(string str0,string str1,string str2,string str3) 符连接四个字符串对象
6	public bool Contains(string value) 返回一个值，指示指定的字符串对象是否发生此字符串中
7	public static string Copy(string str) 创建具有相同的值作为指定字符串的新String对象
8	public void CopyTo(int sourceIndex,char[] destination,int destinationIndex,int count) 复制从字符串对象到指定位置Unicode字符数组的指定位置指定的字符数
9	public bool EndsWith(string value) 确定字符串对象的末尾是否与指定的字符串匹配

序号	属性名称和描述
10	public bool Equals(string value) 确定当前字符串对象，并指定字符串对象是否具有相同的值
11	public static bool Equals(string a,string b) 确定两个指定的String对象是否具有相同的值
12	public static string Format(string format,Object arg0) 替换指定的字符串指定对象的字符串表示在一个或多个格式项
13	public int IndexOf(char value) 返回当前字符串指定Unicode字符中第一次出现从零开始的索引
14	public int IndexOf(string value) 返回在这种情况下指定字符串中第一次出现从零开始的索引
15	public int IndexOf(char value,int startIndex) 返回此字符串指定Unicode字符中第一次出现从零开始的索引，搜索开始在指定的字符位置
16	public int IndexOf(string value,int startIndex) 返回在这种情况下指定字符串中第一次出现的从零开始的索引，搜索开始在指定的字符位置
17	public int IndexOfAny(char[] anyOf) 返回Unicode字符指定数组中第一次出现的任何字符的这个实例从零开始的索引
18	public int IndexOfAny(char[] anyOf,int startIndex) 返回Unicode字符指定数组，开始搜索从指定字符位置中第一次出现的任何字符的这个实例从零开始的索引
19	public string Insert(int startIndex,string value) 返回在指定的字符串被插入在当前字符串对象指定索引位置一个新的字符串
20	public static bool IsNullOrEmpty(string value) 指示指定的字符串是否为空或空字符串
21	public static string Join(string separator,params string[] value) 连接字符串数组中的所有元素，使用每个元件之间指定的分隔
22	public static string Join(string separator,string[] value,int startIndex,int count) 连接字符串数组的指定元素，利用每一个元素之间指定分隔符
23	public int LastIndexOf(char value) 返回当前字符串对象中指定的Unicode字符的最后出现从零开始的索引位置
24	public int LastIndexOf(string value) 返回当前字符串对象中的指定字符串最后一次出现的从零开始的索引位置
25	public string Remove(int startIndex) 删除在当前实例中的所有字符，开始在指定的位置，并继续通过最后位置，并返回字符串
26	public string Remove(int startIndex,int count) 删除在当前字符串的字符开始的指定位置的指定数量，并返回字符串
27	public string Replace(char oldChar,char newChar) 替换与指定的Unicode字符当前字符串对象指定的Unicode字符的所有匹配，并返回新的字符串
28	public string Replace(string oldValue,string newValue) 替换用指定的字符串当前字符串对象指定的字符串的所有匹配，并返回新的字符串
29	public string[] Split(params char[] separator) 返回一个字符串数组，其中包含的子字符串在当前字符串对象，由指定的Unicode字符数组的元素分隔

序号	属性名称和描述
30	public string[] Split(char[] separator,int count) 返回一个字符串数组，其中包含的子字符串在当前字符串对象，由指定的Unicode字符数组的元素分隔。整型参数指定的子串返回最大数量
31	public bool StartsWith(string value) 确定此字符串实例的开头是否与指定的字符串匹配
32	public char[] ToCharArray() 返回一个Unicode字符数组，在当前字符串对象中的所有字符
33	public char[] ToCharArray(int startIndex,int length) 返回一个Unicode字符数组，在当前字符串对象中的所有字符，从指定的索引开始，并到指定的长度
34	public string ToLower() 返回此字符串的一个副本转换为小写
35	public string ToUpper() 返回此字符串的一个副本转换为大写
36	public string Trim() 从当前String对象去除所有开头和结尾的空白字符

方法上述名单并不是详尽的信息，请访问MSDN库的方法和String类的构造函数的完整列表。

示例:

下面的例子说明了一些上面提到的方法:

比较字符串:

```
using System;
namespace StringApplication
{
  class StringProg
  {
    static void Main(string[] args)
    {
      string str1 = "This is test";
      string str2 = "This is text";

      if (String.Compare(str1, str2) == 0)
      {
        Console.WriteLine(str1 + " and " + str2 +  " are equal.");
      }
      else
      {
        Console.WriteLine(str1 + " and " + str2 + " are not equal.");
      }
      Console.ReadKey() ;
    }
```

```
  }
}
```

编译执行上述代码，得到如下结果：

```
This is test and This is text are not equal.
```

String包含字符串：

```
using System;
namespace StringApplication
{
  class StringProg
  {
    static void Main(string[] args)
    {
      string str = "This is test";
      if (str.Contains("test"))
      {
        Console.WriteLine("The sequence 'test' was found.");
      }
      Console.ReadKey() ;
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
The sequence 'test' was found.
```

获取一个子字符串：

```
using System;
namespace StringApplication
{
  class StringProg
  {
    static void Main(string[] args)
    {
      string str = "Last night I dreamt of San Pedro";
      Console.WriteLine(str);
      string substr = str.Substring(23);
      Console.WriteLine(substr);
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
San Pedro
```

连接字符串：

```
using System;
namespace StringApplication
{
  class StringProg
  {
    static void Main(string[] args)
    {
      string[] starray = new string[]{"Down the way nights are dark",
      "And the sun shines daily on the mountain top",
      "I took a trip on a sailing ship",
      "And when I reached Jamaica",
      "I made a stop"};

      string str = String.Join("\n", starray);
      Console.WriteLine(str);
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Down the way nights are dark
And the sun shines daily on the mountain top
I took a trip on a sailing ship
And when I reached Jamaica
I made a stop
```

结构体

在 C# 中，结构体是一种值数据类型。包含数据成员和方法成员。 struct 关键字是用于创建一个结构体。

结构体是用来代表一个记录。假设你想追踪一个图书馆的书。你可能想追踪每本书的属性如下：

- 标题
- 作者
- 类别
- 书号

定义一个结构体

定义一个结构体，你必须要声明这个结构体。结构体声明定义了一种新的数据类型，这个数据类型为你的程序包含了一个以上的成员变量。

例如，你可以声明一个书的结构如下：

```
struct Books
{
  public string title;
  public string author;
  public string subject;
  public int book_id;
};
```

下面的程序显示了结构体的用法：

```
using System;
struct Books
{
  public string title;
  public string author;
  public string subject;
  public int book_id;
};

public class testStructure
{
  public static void Main(string[] args)
```

```
  {

    Books Book1;   /* 将 Book1 声明为 Book 类型  */
    Books Book2;   /* 将 Book2 声明为 Book 类型 */

    /* book 1 specification */
    Book1.title = "C Programming";
    Book1.author = "Nuha Ali";
    Book1.subject = "C Programming Tutorial";
    Book1.book_id = 6495407;

    /* book 2 详细数据 */
    Book2.title = "Telecom Billing";
    Book2.author = "Zara Ali";
    Book2.subject =  "Telecom Billing Tutorial";
    Book2.book_id = 6495700;

    /* 打印 Book1 信息 */
    Console.WriteLine( "Book 1 title : {0}", Book1.title);
    Console.WriteLine("Book 1 author : {0}", Book1.author);
    Console.WriteLine("Book 1 subject : {0}", Book1.subject);
    Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);

    /* 打印 Book2 信息 */
    Console.WriteLine("Book 2 title : {0}", Book2.title);
    Console.WriteLine("Book 2 author : {0}", Book2.author);
    Console.WriteLine("Book 2 subject : {0}", Book2.subject);
    Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);

    Console.ReadKey();

  }
}
```

编译执行上述代码，得到如下结果：

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

结构体的特征

你已经使用了一个名为 Books 的简单结构体。C# 中的结构体与传统的 C 或者 C++ 有明显的不同。 C# 中的结构体有以下特征：

- 结构体可以有方法，域，属性，索引器，操作方法，和事件。
- 结构体可以定义构造函数，但是不能构造析构函数。尽管如此，你还是不能定义一个结构体的默认构造函数。默认构造函数是自动定义的，且不能被改变。
- 与类不同，结构体不能继承其他的结构体或这其他的类。
- 结构体不能用于作为其他结构或者类的基。
- 结构体可以实现一个或多个接口。
- 结构体成员不能被指定为抽象的，虚拟的，或者保护的对象。
- 使用 New 运算符创建结构体对象时，将创建该结构体对象，并且调用适当的构造函数。与类不同的是，结构体的实例化可以不使用 New 运算符。
- 如果不使用 New 操作符，那么在初始化所有字段之前，字段将保持未赋值状态，且对象不可用。

类和结构体

类和结构体有以下几个主要的区别：

- 类是引用类型，结构体是值类型
- 结构体不支持继承
- 结构体不能有默认构造函数

针对上述讨论，让我们重写前面的例子：

```
using System;
struct Books
{
  private string title;
  private string author;
  private string subject;
  private int book_id;
  public void getValues(string t, string a, string s, int id)
  {
    title = t;
```

```
    author = a;
    subject = s;
    book_id = id;
  }
  public void display()
  {
    Console.WriteLine("Title : {0}", title);
    Console.WriteLine("Author : {0}", author);
    Console.WriteLine("Subject : {0}", subject);
    Console.WriteLine("Book_id :{0}", book_id);
  }

};

public class testStructure
{
  public static void Main(string[] args)
  {

    Books Book1 = new Books();    /* 将 Book1 声明为 Book 类型  */
    Books Book2 = new Books();    /* 将 Book2 声明为 Book 类型 */

    /* book 1 详细信息 */
    Book1.getValues("C Programming",
    "Nuha Ali", "C Programming Tutorial",6495407);

    /* book 2 详细信息 */
    Book2.getValues("Telecom Billing",
    "Zara Ali", "Telecom Billing Tutorial", 6495700);

    /* 打印 Book1 信息 */
    Book1.display();

    /* 打印 Book2 信息 */
    Book2.display();

    Console.ReadKey();

  }
}
```

编译执行上述代码，得到如下结果：

```
Title : C Programming
Author : Nuha Ali
Subject : C Programming Tutorial
```

```
Book_id : 6495407
Title : Telecom Billing
Author : Zara Ali
Subject : Telecom Billing Tutorial
Book_id : 6495700
```

枚举

枚举是一组命名的整型常量。枚举类型使用 enum 关键字声明。

C# 枚举是值的数据类型。换句话说，枚举包含它自己的值，不能继承或被继承。

声明枚举变量

用于声明枚举的一般语法：

```
enum <enum_name>
{
   enumeration list
};
```

这里

- enum_name 指定枚举类型名称。
- enumeration list 是一个逗号分隔的标识符的列表。

每个枚举列表中的符号表示整数值，比它前面的符号一个更大。
缺省情况下，第一枚举符号的值是 0。

例如：

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

示例

下面的例子演示了使用枚举变量：

```
using System;
namespace EnumApplication
{
  class EnumProgram
  {
    enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

    static void Main(string[] args)
    {
```

```
        int WeekdayStart = (int)Days.Mon;
        int WeekdayEnd = (int)Days.Fri;
        Console.WriteLine("Monday: {0}", WeekdayStart);
        Console.WriteLine("Friday: {0}", WeekdayEnd);
        Console.ReadKey();
    }
  }
}
```

编译和执行上述代码，得到如下结果：

```
Monday: 1
Friday: 5
```

类

当你定义一个类，你实际上定义的是一个数据类型的蓝图。实际上你并没有定义任何数据，但是它定义了类的名字意味着什么。也就是说，一个类的对象由一些可以在该类上进行的操作构成。对象是类的实例。构成类的方法和变量被称为类的成员。

定义一个类

一个类定义以关键字 class 开始，其后跟的是类的名称；类的主体部分体由一对花括号括起来。以下是一个类定义的一般形式：

```
 class  class_name
{
  // 成员变量
    variable1;
    variable2;
  ...
    variableN;
  // 成员变量
    method1(parameter_list)
  {
     // 方法主体
  }
    method2(parameter_list)
  {
     // 方法主体
  }
  ...
    methodN(parameter_list)
  {
     // 方法主体
  }
}
```

笔记：

- 访问说明符的访问成员的规则与类本身的访问规则相同。如果没有说明，则默认访问的类的类型为 internal 。对成员的默认访问类型是 private 。
- 数据类型指定了变量的类型，如果有返回值的话，返回指定方法的数据类型。

？ 访问类的成员，可以使用 "." 点运算符。

？ 点运算符连接的成员的名字和对象的名称。

下面的例子说明了到目前为止对于概念的讨论：

```
using System;
namespace BoxApplication
{
   class Box
   {
      public double length;  // box 的长度
      public double breadth; // box 的宽度
      public double height;  // box 的高度
   }
   class Boxtester
   {
       static void Main(string[] args)
      {
         Box Box1 = new Box();   // 声明 box1 为 box 类型
         Box Box2 = new Box();   // 声明 box2 为 box 类型
         double volume = 0.0;    // 在这里存放 box 的体积

         // box 1 详细数据
         Box1.height = 5.0;
         Box1.length = 6.0;
         Box1.breadth = 7.0;

         // box 2 详细数据
         Box2.height = 10.0;
         Box2.length = 12.0;
         Box2.breadth = 13.0;

         // box 1 的体积
         volume = Box1.height * Box1.length * Box1.breadth;
         Console.WriteLine("Volume of Box1 : {0}",  volume);

         // box 2 的体积
         volume = Box2.height * Box2.length * Box2.breadth;
         Console.WriteLine("Volume of Box2 : {0}", volume);
         Console.ReadKey();
      }
   }
}
```

编译执行上述代码，得到如下结果：

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

成员函数和封装

一个类的成员函数有其自己定义的或其原型写在类体中。它可以操作该类的任何成员对象，并且可以访问一个类的所有成员。

成员变量是一个对象的属性（从设计的角度来看），他们都是私有（private）的以便实施封装。这些变量只能被 public 成员函数访问。

让我们把上述那些概念来 set 和 get 一个类中不同的类成员的值：

```
using System;
namespace BoxApplication
{
  class Box
  {
    private double length;   // box 的长度
    private double breadth;  // box 的宽度
    private double height;   // box 的高度
    public void setLength( double len )
    {
      length = len;
    }

    public void setBreadth( double bre )
    {
      breadth = bre;
    }

    public void setHeight( double hei )
    {
      height = hei;
    }
    public double getVolume()
    {
      return length * breadth * height;
    }
  }
  class Boxtester
```

```
  {
    static void Main(string[] args)
    {
      Box Box1 = new Box();   // 将 Box1 声明为 Box 类型
      Box Box2 = new Box();
      double volume;

      // 将 Box2 声明为 Box 类型
      // box 1 详细数据
      Box1.setLength(6.0);
      Box1.setBreadth(7.0);
      Box1.setHeight(5.0);

      // box 2 详细数据
      Box2.setLength(12.0);
      Box2.setBreadth(13.0);
      Box2.setHeight(10.0);

      // box 1 的体积
      volume = Box1.getVolume();
      Console.WriteLine("Volume of Box1 : {0}" ,volume);

      // box 2 的体积
      volume = Box2.getVolume();
      Console.WriteLine("Volume of Box2 : {0}", volume);

      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

C# 构造函数

一个类的**构造函数**（constructor）是类的一种特殊的成员函数，当我们创建一个类的新的对象时执行该函数。

构造函数与类具有相同的名称，但它没有任何返回类型。下面的例子说明了构造函数的概念：

```
using System;
namespace LineApplication
```

```
{
  class Line
  {
    private double length;   // 线段长度
    public Line()
    {
      Console.WriteLine("Object is being created");
    }

    public void setLength( double len )
    {
      length = len;
    }

    public double getLength()
    {
      return length;
    }

    static void Main(string[] args)
    {
      Line line = new Line();

      // 设置线段长度
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Object is being created
Length of line : 6
```

默认构造函数（default constructor）没有任何的参数，但是如果你需要，构造函数是可以有参数的。这种构造函数被称为**参数化的构造函数**（parameterized constructors）。这种技术有助于你在一个对象被创建时指定它的初始值，如下述示例：

```
using System;
namespace LineApplication
{
  class Line
  {
```

```
    private double length;   // 线段长度
    public Line(double len)  //参数化构造函数
    {
      Console.WriteLine("Object is being created, length = {0}", len);
      length = len;
    }

    public void setLength( double len )
    {
      length = len;
    }
    public double getLength()
    {
      return length;
    }

    static void Main(string[] args)
    {
      Line line = new Line(10.0);
      Console.WriteLine("Length of line : {0}", line.getLength());

      // 设置线段长度
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Object is being created
Length of line : 6
Object is being deleted
```

C# 析构函数

析构函数（destructor）是类的一种特殊的成员函数，当类的对象在超出作用域时被执行的一种成员函数。一个**析构函数**的名称是在其类名称前加上一个前缀字符（~），它既不能返回一个值，也不能带有任何参数。

析构函数对退出程序前释放内存资源时非常有用。析构函数不可以被继承或重载。

下面的示例解释了析构函数的概念：

```
using System;
namespace LineApplication
{
  class Line
  {
    private double length;   // 线段长度
    public Line()  // 构造函数
    {
      Console.WriteLine("Object is being created");
    }
    ~Line() //析构函数
    {
      Console.WriteLine("Object is being deleted");
    }

    public void setLength( double len )
    {
      length = len;
    }

    public double getLength()
    {
      return length;
    }

    static void Main(string[] args)
    {
      Line line = new Line();

      // 设置线段长度
      line.setLength(6.0);
      Console.WriteLine("Length of line : {0}", line.getLength());
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Object is being created
Length of line : 6
Object is being deleted
```

C# 类的静态成员

我们可以使用 static 关键字将类成员定义为静态的。当我们声明一个类的静态成员时，意味着无论有多少类的对象被创建，只有一个副本的静态成员。

关键字 static 意味着一个类的成员只有一个实例存在。静态变量被用于定义常数，因为他们的值可以通过调用不创建实例的类而被检索出来。静态变量可以在成员函数或者类的定义以外的地方初始化。你也可以在类的定义中初始化静态变量。

下面的示例论证了**静态变量(static variables)**的使用：

```
using System;
namespace StaticVarApplication
{
  class StaticVar
  {
    public static int num;
    public void count()
    {
      num++;
    }
    public int getNum()
    {
      return num;
    }
  }
  class StaticTester
  {
    static void Main(string[] args)
    {
      StaticVar s1 = new StaticVar();
      StaticVar s2 = new StaticVar();
      s1.count();
      s1.count();
      s1.count();
      s2.count();
      s2.count();
      s2.count();
      Console.WriteLine("Variable num for s1: {0}", s1.getNum());
      Console.WriteLine("Variable num for s2: {0}", s2.getNum());
      Console.ReadKey();
    }
```

```
  }
}
```

编译执行上述代码，得到如下结果：

```
Variable num for s1: 6
Variable num for s2: 6
```

你也可以声明 static 的**成员函数**。此类函数只能访问静态变量。静态函数的存在甚至先于创建对象。下面的示例论证了**静态函数**（static functions）的用法：

```
using System;
namespace StaticVarApplication
{
  class StaticVar
  {
    public static int num;
    public void count()
    {
      num++;
    }
    public static int getNum()
    {
      return num;
    }
  }
  class StaticTester
  {
    static void Main(string[] args)
    {
      StaticVar s = new StaticVar();
      s.count();
      s.count();
      s.count();
      Console.WriteLine("Variable num: {0}", StaticVar.getNum());
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Variable num: 3
```

继承

面向对象程序设计中最重要的一个概念就是继承（inheritance）。继承允许我们在另一个类中定义一个新的类，这使得它更容易创建和维护一个应用程序。这也提供了一个机会来重用代码的功能，加快实现时间。

创建一个类的时候，不是要写全新的数据成员和成员函数，程序员可以指定新的类继承一个已经存在的类的成员。已有的类称为**基类（base class）**，新的类称为**派生类（derived class）**。

继承的思想实现了 IS-A 的关系。例如，哺乳动物是（IS-A）动物，狗是（IS-A）哺乳动物，因此狗是（IS-A）一个动物等。

基类和派生类

一个类可以从多个类或接口被派生，这意味着它可以从多个基类或接口继承数据和函数。

用 C# 创建派生类的语法如下：

```
<acess-specifier> class <base_class>
{
  ...
}
class <derived_class> : <base_class>
{
  ...
}
```

比如基类为 Shape，其派生类为 Rectangle：

```
using System;
namespace InheritanceApplication
{
  class Shape
  {
    public void setWidth(int w)
    {
      width = w;
    }
    public void setHeight(int h)
    {
      height = h;
    }
```

```
    protected int width;
    protected int height;
  }

  // 派生类
  class Rectangle: Shape
  {
    public int getArea()
    {
      return (width * height);
    }
  }

  class RectangleTester
  {
    static void Main(string[] args)
    {
      Rectangle Rect = new Rectangle();

      Rect.setWidth(5);
      Rect.setHeight(7);

      // 打印对象的面积
      Console.WriteLine("Total area: {0}",  Rect.getArea());
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Total area: 35
```

初始化基类

派生类继承基类的成员变量和成员方法。因此，父类对象应该是先于子类被创建。你可以在初始化列表中说明父类的初始化。

下面的程序论证了上述方法：

```
using System;
namespace RectangleApplication
{
  class Rectangle
```

```
{
  //成员变量
  protected double length;
  protected double width;
  public Rectangle(double l, double w)
  {
    length = l;
    width = w;
  }

  public double GetArea()
  {
    return length * width;
  }

  public void Display()
  {
    Console.WriteLine("Length: {0}", length);
    Console.WriteLine("Width: {0}", width);
    Console.WriteLine("Area: {0}", GetArea());
  }
}// Rectangle 类结束

class Tabletop : Rectangle
{
  private double cost;
  public Tabletop(double l, double w) : base(l, w)
  { }
  public double GetCost()
  {
    double cost;
    cost = GetArea() * 70;
    return cost;
  }
  public void Display()
  {
    base.Display();
    Console.WriteLine("Cost: {0}", GetCost());
  }
}
class ExecuteRectangle
{
  static void Main(string[] args)
  {
    Tabletop t = new Tabletop(4.5, 7.5);
```

```
      t.Display();
      Console.ReadLine();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Length: 4.5
Width: 7.5
Area: 33.75
Cost: 2362.5
```

C# 中的多重继承

C# 不支持多重继承。但是你可以使用接口来实现多重继承。下面的程序实现了该功能：

```
using System;
namespace InheritanceApplication
{
  class Shape
  {
    public void setWidth(int w)
    {
      width = w;
    }
    public void setHeight(int h)
    {
      height = h;
    }
    protected int width;
    protected int height;
  }

  // 基类 PaintCost
  public interface PaintCost
  {
    int getCost(int area);

  }

  // 派生类
  class Rectangle : Shape, PaintCost
  {
```

```
    public int getArea()
    {
      return (width * height);
    }
    public int getCost(int area)
    {
      return area * 70;
    }
  }
  class RectangleTester
  {
    static void Main(string[] args)
    {
      Rectangle Rect = new Rectangle();
      int area;
      Rect.setWidth(5);
      Rect.setHeight(7);
      area = Rect.getArea();

      //打印对象面积
      Console.WriteLine("Total area: {0}",  Rect.getArea());
      Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Total area: 35
Total paint cost: $2450
```

多态性

多态性（polymorphism）这个词意味着有多种形式。在面向对象的编程范式中，多态性往往表现为“一个接口，多个函数”。

多态性可以是静态的，也可以是动态的。在 **静态多态**（static polymorphism）**性** 中，一个函数的响应是在编译时确定。**动态多态性**（ dynamic polymorphism ）中，其函数响应是在运行时决定。

静态多态性

在编译时将一个函数与一个对象连接起来的机制被称为早期绑定机制。它也被称为静态绑定。C # 提供了两种技术实现静态多态性。他们是：

- 函数重载
- 运算符重载

我们将在下一章讨论运算符重载。

函数重载

你可以在同一范围对同一函数名有多重定义。该函数的定义必须用不同的类型或通过参数列表中的参数的数量进行区分。不可以只用不同的返回类型区分不同的重载函数声明。

下面的示例显示使用函数 print() 打印不同的数据类型：

```
using System;
namespace PolymorphismApplication
{
  class Printdata
  {
    void print(int i)
    {
      Console.WriteLine("Printing int: {0}", i );
    }

    void print(double f)
    {
      Console.WriteLine("Printing float: {0}" , f);
```

```
    }

    void print(string s)
    {
      Console.WriteLine("Printing string: {0}", s);
    }
    static void Main(string[] args)
    {
      Printdata p = new Printdata();

      // 调用 print 函数打印整型
      p.print(5);

      // 调用 print 函数打印浮点型
      p.print(500.263);

      // 调用 print 函数打印字符型
      p.print("Hello C++");
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

动态多态性

C# 允许你创建一个抽象类，被用于提供部分类的接口实现。执行完成时，派生类继承它。**抽象类**（Abstract classes）包含抽象方法，这是由派生类来实现的。派生类具有更具体化，专业化的功能。

以下是关于抽象类的规则：

- 你不能创建抽象类的实例
- 你不能在抽象类的外部声明抽象方法
- 当一个类声明为 **密封的**（sealed），它不能被继承，抽象类被不能声明为密封的。

下面的程序演示了一个抽象类：

```
using System;
namespace PolymorphismApplication
{
  abstract class Shape
  {
    public abstract int area();
  }
  class Rectangle:  Shape
  {
    private int length;
    private int width;
    public Rectangle( int a=0, int b=0)
    {
      length = a;
      width = b;
    }
    public override int area ()
    {
      Console.WriteLine("Rectangle class area :");
      return (width * length);
    }
  }

  class RectangleTester
  {
    static void Main(string[] args)
    {
      Rectangle r = new Rectangle(10, 7);
      double a = r.area();
      Console.WriteLine("Area: {0}",a);
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Rectangle class area :
Area: 70
```

当你在一个类中有定义函数，并且希望它可以在一个被继承的类中实现功能时，你可以使用**虚函数**（virtual functions）。虚函数可以在不同的被继承的类中实现，并且将在程序运行时调用此类函数。

动态多样性是通过**抽象类**和**虚函数**实现的。

下列程序证实了上述说法：

```
using System;
namespace PolymorphismApplication
{
  class Shape
  {
    protected int width, height;
    public Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    public virtual int area()
    {
      Console.WriteLine("Parent class area :");
      return 0;
    }
  }
  class Rectangle: Shape
  {
    public Rectangle( int a=0, int b=0): base(a, b)
    {

    }
    public override int area ()
    {
      Console.WriteLine("Rectangle class area :");
      return (width * height);
    }
  }
  class Triangle: Shape
  {
    public Triangle(int a = 0, int b = 0): base(a, b)
    {

    }
    public override int area()
    {
      Console.WriteLine("Triangle class area :");
      return (width * height / 2);
    }
  }
  class Caller
  {
    public void CallArea(Shape sh)
    {
```

```
        int a;
        a = sh.area();
        Console.WriteLine("Area: {0}", a);
      }
    }
    class Tester
    {

      static void Main(string[] args)
      {
        Caller c = new Caller();
        Rectangle r = new Rectangle(10, 7);
        Triangle t = new Triangle(10, 5);
        c.CallArea(r);
        c.CallArea(t);
        Console.ReadKey();
      }
    }
}
```

编译执行上述代码，得到如下结果：

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```

运算符重载

你可以重新定义或重载大部分 C# 可用的内置操作符。因此，程序员也可以使用用户定义类型的操作符。重载操作符是特殊关键字 operator 其后跟被定义的名字的符号。像其他函数一样，重载操作符也有返回类型和参数列表。

例如，浏览如下函数：

```
public static Box operator+ (Box b, Box c)
{
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

上述函数实现了用户定义的 Box 类中加运算符，它增加了两个 Box 对象，并返回这个结果 Box 对象。

实现运算符重载

下列程序显示了完整地实现：

```
using System;
namespace OperatorOvlApplication
{
   class Box
   {
      private double length;   // box 的长度
      private double breadth;  // box 的宽度
      private double height;   // box 的高度

      public double getVolume()
      {
         return length * breadth * height;
      }

      public void setLength( double len )
      {
         length = len;
      }
```

```
   public void setBreadth( double bre )
   {
      breadth = bre;
   }

   public void setHeight( double hei )
   {
      height = hei;
   }

   // 重载 + operator 来增加两个 Box 对象
   public static Box operator+ (Box b, Box c)
   {
      Box box = new Box();
      box.length = b.length + c.length;
      box.breadth = b.breadth + c.breadth;
      box.height = b.height + c.height;
      return box;
   }

}

class Tester
{
   static void Main(string[] args)
   {
      Box Box1 = new Box();    // 声明 box1 为 box 类型
      Box Box2 = new Box();    // 声明 box2 为 box 类型
      Box Box3 = new Box();   // 声明 box3 为 box 类型
      double volume = 0.0;    // 在这里存放 box 的体积

      // box 1 详细数据
      Box1.setLength(6.0);
      Box1.setBreadth(7.0);
      Box1.setHeight(5.0);

      // box 2 详细数据
      Box2.setLength(12.0);
      Box2.setBreadth(13.0);
      Box2.setHeight(10.0);

      // box 1 的体积
      volume = Box1.getVolume();
      Console.WriteLine("Volume of Box1 : {0}", volume);
```

```
        //  box 2 的体积
        volume = Box2.getVolume();
        Console.WriteLine("Volume of Box2 : {0}", volume);

        // 将两个对象相加如下：
        Box3 = Box1 + Box2;

        // box 3 的体积
        volume = Box3.getVolume();
        Console.WriteLine("Volume of Box3 : {0}", volume);
        Console.ReadKey();
      }
    }
}
```

编译执行上述代码，得到如下结果：

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

可重载与不可重载的运算符

下表列出了 C# 中运算符的可重载能力：

操作符	描述
+, −, !, ~, ++, −−	这些一元运算符使用一个操作数，可以被重载
+, −, *, /, %	这些二元运算符使用一个操作数，可以被重载
==, !=, <, >, <=, >=	这些比较运算符，可以被重载
&&, ||	这些条件逻辑运算符不可被直接重载
+=, −=, *=, /=, %=	赋值运算符不能重载
=, ., ?:, ->, new, is, sizeof, typeof	这些运算符不能被重载

示例

针对上述讨论，让我们扩展上面的例子，重载更多的操作符：

```
using System;
namespace OperatorOvlApplication
{
  class Box
```

```
{
   private double length;   // box 的长度
   private double breadth;  // box 的宽度
   private double height;   // box 的高度

   public double getVolume()
   {
      return length * breadth * height;
   }

   public void setLength( double len )
   {
      length = len;
   }

   public void setBreadth( double bre )
   {
      breadth = bre;
   }


   public void setHeight( double hei )
   {
      height = hei;
   }

   // 重载 + operator 来增加两个 Box 对象
   public static Box operator+ (Box b, Box c)
   {
      Box box = new Box();
      box.length = b.length + c.length;
      box.breadth = b.breadth + c.breadth;
      box.height = b.height + c.height;
      return box;
   }

   public static bool operator == (Box lhs, Box rhs)
   {
      bool status = false;
      if (lhs.length == rhs.length && lhs.height == rhs.height && lhs.breadth == rhs.breadth)
      {
         status = true;
      }
      return status;
   }
```

```
public static bool operator !=(Box lhs, Box rhs)
{
  bool status = false;
  if (lhs.length != rhs.length || lhs.height != rhs.height || lhs.breadth != rhs.breadth)
  {
    status = true;
  }
  return status;
}

public static bool operator <(Box lhs, Box rhs)
{
  bool status = false;
  if (lhs.length < rhs.length && lhs.height < rhs.height && lhs.breadth < rhs.breadth)
  {
    status = true;
  }
  return status;
}

public static bool operator >(Box lhs, Box rhs)
{
  bool status = false;
  if (lhs.length > rhs.length && lhs.height > rhs.height && lhs.breadth > rhs.breadth)
  {
    status = true;
  }
  return status;
}

public static bool operator <=(Box lhs, Box rhs)
{
  bool status = false;
  if (lhs.length <= rhs.length && lhs.height <= rhs.height && lhs.breadth <= rhs.breadth)
  {
    status = true;
  }
  return status;
}

public static bool operator >=(Box lhs, Box rhs)
{
  bool status = false;
  if (lhs.length >= rhs.length && lhs.height >= rhs.height && lhs.breadth >= rhs.breadth)
```

```
    {
      status = true;
    }
    return status;
  }
  public override string ToString()
  {
    return String.Format("({0}, {1}, {2})", length, breadth, height);
  }
}

class Tester
{
  static void Main(string[] args)
  {
    Box Box1 = new Box();   // 声明 box1 为 box 类型
    Box Box2 = new Box();   // 声明 box2 为 box 类型
    Box Box3 = new Box();   // 声明 box3 为 box 类型
    Box Box4 = new Box();   // 声明 box4 为 box 类型
    double volume = 0.0;   // 这里存放 box 的体积

    // box 1 详细数据
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);

    // box 2 详细数据
    Box2.setLength(12.0);
    Box2.setBreadth(13.0);
    Box2.setHeight(10.0);

    //使用重载的 ToString() 显示 Boxes
    Console.WriteLine("Box 1: {0}", Box1.ToString());
    Console.WriteLine("Box 2: {0}", Box2.ToString());

    // box 1 的体积
    volume = Box1.getVolume();
    Console.WriteLine("Volume of Box1 : {0}", volume);

    // box 2 的体积
    volume = Box2.getVolume();
    Console.WriteLine("Volume of Box2 : {0}", volume);

    // 将两个对象相加如下：
    Box3 = Box1 + Box2;
```

```
        Console.WriteLine("Box 3: {0}", Box3.ToString());

        // box 3 的体积
        volume = Box3.getVolume();
        Console.WriteLine("Volume of Box3 : {0}", volume);

        //比较 boxes
        if (Box1 > Box2)
          Console.WriteLine("Box1 is greater than Box2");
        else
          Console.WriteLine("Box1 is  greater than Box2");

        if (Box1 < Box2)
          Console.WriteLine("Box1 is less than Box2");
        else
          Console.WriteLine("Box1 is not less than Box2");

        if (Box1 >= Box2)
          Console.WriteLine("Box1 is greater or equal to Box2");
        else
          Console.WriteLine("Box1 is not greater or equal to Box2");

        if (Box1 <= Box2)
          Console.WriteLine("Box1 is less or equal to Box2");
        else
          Console.WriteLine("Box1 is not less or equal to Box2");

        if (Box1 != Box2)
          Console.WriteLine("Box1 is not equal to Box2");
        else
          Console.WriteLine("Box1 is not greater or equal to Box2");
        Box4 = Box3;

        if (Box3 == Box4)
          Console.WriteLine("Box3 is equal to Box4");
        else
          Console.WriteLine("Box3 is not equal to Box4");

        Console.ReadKey();
      }
    }
}
```

编译执行上述代码，得到如下结果：

```
Box 1: (6, 7, 5)
Box 2: (12, 13, 10)
Volume of Box1 : 210
Volume of Box2 : 1560
Box 3: (18, 20, 15)
Volume of Box3 : 5400
Box1 is not greater than Box2
Box1 is less than Box2
Box1 is not greater or equal to Box2
Box1 is less or equal to Box2
Box1 is not equal to Box2
Box3 is equal to Box4
```

接口

一个接口定义为一种句法的合同，所有类继承接口应遵循。这个接口定义了部分的句法合同“**是什么（what）**”和派生类定义了部分的句法合同“**怎么做（how）**”。

接口定义的属性，方法和事件，是接口的成员。接口只包含成员的声明。它是派生类定义的成员的责任。它提供一个派生类可以采用的标准的结构。

抽象类在一定程度上服务于同一个目的，然而，它们主要用于基类的方法和派生类中实现的功能。

接口的声明

接口使用关键字 interface 声明。它类似于类的声明。接口声明缺省为 public 类型。以下是一个接口声明的例子：

```
public interface ITransactions
{
  // 接口成员
  void showTransaction();
  double getAmount();
}
```

示例

下面的示例演示上述接口的实现：

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System;

namespace InterfaceApplication
{
  public interface ITransactions
  {
    // 接口成员
    void showTransaction();
    double getAmount();
  }
```

```
  public class Transaction : ITransactions
  {
    private string tCode;
    private string date;
    private double amount;
    public Transaction()
    {
      tCode = " ";
      date = " ";
      amount = 0.0;
    }

    public Transaction(string c, string d, double a)
    {
      tCode = c;
      date = d;
      amount = a;
    }

    public double getAmount()
    {
      return amount;
    }

    public void showTransaction()
    {
      Console.WriteLine("Transaction: {0}", tCode);
      Console.WriteLine("Date: {0}", date);
      Console.WriteLine("Amount: {0}", getAmount());
    }
  }
  class Tester
  {
    static void Main(string[] args)
    {
      Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
      Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
      t1.showTransaction();
      t2.showTransaction();
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
```

命名空间

命名空间（namespace）专为提供一种来保留一套独立名字与其他命名区分开来的方式。一个命名空间中声明的类的名字与在另一个命名空间中声明的相同的类名并不会发生冲突。

命名空间的定义

命名空间的定义以关键字 namespace 开始，其后跟命名空间的名称：

```
namespace namespace_name
{
  // 代码声明
}
```

调用的函数或变量的命名空间启用版本，在命名空间名称如下：

```
namespace_name.item_name;
```

下面的程序演示了命名空间的使用：

```
using System;
namespace first_space
{
  class namespace_cl
  {
    public void func()
    {
      Console.WriteLine("Inside first_space");
    }
  }
}

namespace second_space
{
  class namespace_cl
  {
    public void func()
    {
      Console.WriteLine("Inside second_space");
    }
  }
}
```

```
class TestClass
{
  static void Main(string[] args)
  {
    first_space.namespace_cl fc = new first_space.namespace_cl();
    second_space.namespace_cl sc = new second_space.namespace_cl();
    fc.func();
    sc.func();
    Console.ReadKey();
  }
}
```

编译执行上述代码，得到如下结果：

```
Inside first_space
Inside second_space
```

关键字 using

关键词 using 指出了该程序是在使用给定的命名空间的名称。例如，我们在程序中使用的是系统命名空间。其中有 Console 类的定义。我们只需要写：

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

我们还可以写完全限定名称：

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

你也可以使用 using 指令避免还要在前面加上 namespace 。这个指令会告诉编译器后面的代码使用的是在指定的命名空间中的名字。命名空间是因此包含下面的代码：

让我们重写前面的示例，使用 using 指令：

```
using System;
using first_space;
using second_space;

namespace first_space
{
  class abc
  {
    public void func()
    {
```

```
        Console.WriteLine("Inside first_space");
      }
    }
}

namespace second_space
{
  class efg
  {
    public void func()
    {
      Console.WriteLine("Inside second_space");
    }
  }
}

class TestClass
{
  static void Main(string[] args)
  {
    abc fc = new abc();
    efg sc = new efg();
    fc.func();
    sc.func();
    Console.ReadKey();
  }
}
```

编译执行上述代码，得到如下结果：

```
Inside first_space
Inside second_space
```

嵌套命名空间

你可以在一个命名空间中定义另一个命名空间，方法如下：

```
namespace namespace_name1
{
  // 代码声明
  namespace namespace_name2
  {
   //代码声明
```

```
  }
}
```

你可以使用点运算符“.”来访问嵌套命名空间中的成员

```
using System;
using first_space;
using first_space.second_space;

namespace first_space
{
  class abc
  {
    public void func()
    {
      Console.WriteLine("Inside first_space");
    }
  }
  namespace second_space
  {
    class efg
    {
      public void func()
      {
        Console.WriteLine("Inside second_space");
      }
    }
  }
}

class TestClass
{
  static void Main(string[] args)
  {
    abc fc = new abc();
    efg sc = new efg();
    fc.func();
    sc.func();
    Console.ReadKey();
  }
}
```

编译执行上述代码，得到如下结果：

```
Inside first_space
Inside second_space
```

预处理指令

预处理指令是一种给编译器的指令，用来在实际的编译开始之前预处理一些信息。

所有的预处理指令都以 # 开始，并且在一行预处理指令中，只有空白字符可以出现在指令之前。预处理指令没有声明，所以他们不需要以分号（；）结尾。

C# 编译器不具有独立的预处理机制；然而，指令执行的时候就像是只有这一条一样。在 C# 中，预处理指令被用来帮助条件编译。不像 C 或 C++ 的指令，他们不能创建宏。一个预处理指令必须是这一行代码中的唯一的指令。

C# 中的预处理指令

下面的表格中列出了 C# 中可用的预处理指令：

预处理指令	描述
#define	定义了一串字符，称为符号。
#undef	可以取消定义的符号。
#if	测试一个或多个表达式的结果是否为真。
#else	用于创建复合条件指令，和 #if 一起使用。
#elif	用于创建复合条件指令。
#endif	指出条件指令的结尾。
#line	可以修改编译器的行号，选择性修改输出错误和警告的文件名
#error	从代码的特定位置生成误差
#warning	从代码的特定位置生成一级预警
#region	当你使用 Visual Studio 代码编译器时，你可以展开或折叠一部分代码块
#engregion	它标志着 #region 块的结束

#define 指令

`#define` 预处理指令是用来创建符号常量的。

应用 #define 可以定义一个符号，这个符号会作为一个表达式传递给 #if 指令，这个判断会得到 ture 的 结果。语法如下：

```
#define symbol
```

下面的程序说明了这一点：

```
#define PI
using System;
namespace PreprocessorDAppl
{
    class Program
    {
      static void Main(string[] args)
      {
        #if (PI)
        Console.WriteLine("PI is defined");
        #else
        Console.WriteLine("PI is not defined");
        #endif
        Console.ReadKey();
      }
    }
}
```

编译执行上述代码，得到如下结果：

```
PI is defined
```

条件指令

你可以使用 #if 指令创建一个条件指令。条件指令可以用来判断一个或多个符号是否为真。如果他们的结果为真，编译器就会执行 #if 和下一条指令间的所有代码。

条件指令的语法如下：

```
#if symbol [operator symbol]...
```

当你想测试的符号是 “symbol”这个名字的时候。你也可以使用 ture 和 false 或者提前使用反运算符操作这个符号。

operator symbol（运算符符号）是一种用于符号求值的运算符。运算符可以是下列之一：

- == (相等)
- != (不相等)
- && (与)
- || (或)

你也可以通过括号使用组符号和组运算符。条件指令用于编译代码生成 debug 或者是编译特定配置时。一个条件指令以 #if 开头并且必须明确的以 #endif 指令结束。

下面的程序示范了条件指令的使用方法：

```
#define DEBUG
#define VC_V10
using System;
public class TestClass
{
    public static void Main()
    {
      #if (DEBUG && !VC_V10)
        Console.WriteLine("DEBUG is defined");
      #elif (!DEBUG && VC_V10)
        Console.WriteLine("VC_V10 is defined");
      #elif (DEBUG && VC_V10)
        Console.WriteLine("DEBUG and VC_V10 are defined");
      #else
        Console.WriteLine("DEBUG and VC_V10 are not defined");
      #endif
      Console.ReadKey();
    }
}
```

编译执行上述代码，得到如下结果：

```
DEBUG and VC_V10 are defined
```

正则表达式

正则表达式是一种可以和输入文本相匹配的表达式。.Net framework 提供了一个正则表达式引擎让这种匹配成为可能。一个表达式可以由一个或多个字符，运算符，或结构体组成。

构建正则表达式的定义

有很多种类的字符，运算符，结构体可以定义正则表达式。

- 转义字符
- 字符类
- 集合
- 分组构造
- 限定符
- 回溯引用构造
- 可替换结构
- 替换
- 混合结构

Regex 正则表达式类

Regex 类用于表示一个正则表达式。它有下列常用的方法：

Sr.No	方法
1	public bool IsMatch(string input) 指出输入的字符串中是否含有特定的正则表达式。
2	public bool IsMatch(string input, int startat) 指出输入的字符串中是否含有特定的正则表达式，该函数的 startat 变量指出了字符串开始查找的位置。
3	public static bool IsMatch(string input, string pattern) 指出特定的表达式是否和输入的字符串匹配。
4	public MatchCollection Matches(string input) 在所有出现的正则表达式中搜索特定的输入字符

Sr.No	方法
5	public string Replace(string input, string replacement) 在一个特定的输入字符中，用特定的字符串替换所有满足某个表达式的字符串。
6	public string[] Split(string input) 将一个输入字符拆分成一组子字符串，从一个由正则表达式指出的位置上开始。

有关属性和方法的完成列表，请参见微软的 C# 文档。

示例 1

下列的例子中找出了以 s 开头的单词：

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
    class Program
    {
      private static void showMatch(string text, string expr)
      {
        Console.WriteLine("The Expression: " + expr);
        MatchCollection mc = Regex.Matches(text, expr);
        foreach (Match m in mc)
        {
          Console.WriteLine(m);
        }
      }

      static void Main(string[] args)
      {
        string str = "A Thousand Splendid Suns";

        Console.WriteLine("Matching words that start with 'S': ");
        showMatch(str, @"\bS\S*");
        Console.ReadKey();
      }
    }
}
```

编译执行上述代码，得到如下结果：

```
Matching words that start with 'S':
```

```
The Expression: \bS\S*
Splendid
Suns
```

示例 2

下面的例子中找出了以 m 开头以 e 结尾的单词

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
    class Program
    {
      private static void showMatch(string text, string expr)
      {
        Console.WriteLine("The Expression: " + expr);
        MatchCollection mc = Regex.Matches(text, expr);
        foreach (Match m in mc)
        {
          Console.WriteLine(m);
        }
      }
      static void Main(string[] args)
      {
        string str = "make maze and manage to measure it";

        Console.WriteLine("Matching words start with 'm' and ends with 'e':");
        showMatch(str, @"\bm\S*e\b");
        Console.ReadKey();
      }
    }
}
```

编译执行上述代码，得到如下结果：

```
Matching words start with 'm' and ends with 'e':
The Expression: \bm\S*e\b
make
maze
```

```
manage
measure
```

示例 3

这个例子替换了额外的空白符：

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
    class Program
    {
      static void Main(string[] args)
      {
        string input = "Hello   World   ";
        string pattern = "\\s+";
        string replacement = " ";
        Regex rgx = new Regex(pattern);
        string result = rgx.Replace(input, replacement);

        Console.WriteLine("Original String: {0}", input);
        Console.WriteLine("Replacement String: {0}", result);
        Console.ReadKey();
      }
    }
}
```

编译执行上述代码，得到如下结果：

```
Original String: Hello World
Replacement String: Hello World
```

异常处理

异常是程序执行过程中产生的问题。C# 异常是对程序运行过程中出现的额外情况的一种反馈，例如除数为零时。

异常提供了一种将控制权从程序的一个部分转移到另一个部分的方式。C# 异常处理有四个关键词：try，catch，finally，throw。

- try：try 块标识代码块的哪些特定的异常将被激活。它的后面是一个或多个 catch 块。
- catch：一个用于捕获异常的程序段，将 catch 放在你希望能处理这个异常的地方。“catch”这个关键字标志着异常的捕获。
- finally：finally 保证了无论是否有异常抛出，此代码段中的程序都会被执行。例如，如果你打开了一个文件，那么不管是否发生了异常，文件都需要关闭。
- throw：当出现问题时，程序会抛出异常。这项工作是通过使用 throw 来实现的。

语法

假设一个代码块产生了一个异常，通过使用 try 和 catch 的组合可以捕获这个异常。一个 try/catch 代码块需要放置在可能会产生异常的代码段周围。try/catch 代码段就像是保护代码，它的使用语法如下：

```
try
{
    // statements causing exception
}
catch( ExceptionName e1 )
{
     // error handling code
}
catch( ExceptionName e2 )
{
     // error handling code
}
catch( ExceptionName eN )
{
     // error handling code
}
finally
{
```

```
    // statements to be executed
}
```

当你的 try 语句块可能会抛出多种异常时，你可以列出多种的 catch 语句，以便捕获不同种类的异常。

C#中的异常类

C# 异常由类表示。在 C# 中的异常类主要是直接或间接地来源于 System.Exception 类。有些从 System.Exception 类派生的异常类，它们是 System.ApplicationException 和 System.SystemException 类。

System.ApplicationException 类支持由应用程序生成的异常。因此，由程序员定义的异常应该源于这个类。

System.SystemException 类是所有预定义的系统异常的基类。

下表提供了一些从 Sytem.SystemException 类派生的预定义的异常类：

Exception类	描述
System.IO.IOException	处理 I/O 错误
System.IndexOutOfRangeException	处理的方法是指当一个数组索引超出范围的错误产生
System.ArrayTypeMismatchException	处理时，类型不匹配的数组类型产生的错误
System.NullReferenceException	处理从取消引用一个空对象产生的错误
System.DivideByZeroException	处理来自将一个除零而产生的错误
System.InvalidCastException	处理类型转换过程中产生的错误
System.OutOfMemoryException	处理来自可用内存不足产生的错误
System.StackOverflowException	处理从堆栈溢出产生的错误

处理异常

C# 为在 try catch 语句块中处理异常提供了一种结构化的解决方案。这种方法可以使核心代码段和异常处理部分分离开。

这些异常处理代码段是通过使用 try，catch，finally 关键字实现的。下面是一个除数为零的异常处理情况：

```
using System;
namespace ErrorHandlingApplication
{
    class DivNumbers
    {
      int result;
      DivNumbers()
```

```
        {
          result = 0;
        }
        public void division(int num1, int num2)
        {
          try
          {
            result = num1 / num2;
          }
          catch (DivideByZeroException e)
          {
            Console.WriteLine("Exception caught: {0}", e);
          }
          finally
          {
            .WriteLine("Result: {0}", result);
          }
        }
        static void Main(string[] args)
        {
             DivNumbers d = new DivNumbers();
          d.division(25, 0);
          Console.ReadKey();
        }
      }
}
```

编译执行上述代码，得到如下结果：

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ...
Result: 0
```

创建自定义异常

你也可以定义你自己的异常。自定义异常类继承自 ApplicationException 类。示范如下：

```
using System;
namespace UserDefinedException
{
    class TestTemperature
    {
      static void Main(string[] args)
      {
```

```
            Temperature temp = new Temperature();
            try
            {
              temp.showTemp();
            }
            catch(TempIsZeroException e)
            {
              Console.WriteLine("TempIsZeroException: {0}", e.Message);
            }
            Console.ReadKey();
        }
    }
}

public class TempIsZeroException: ApplicationException
{
    public TempIsZeroException(string message): base(message)
    {
    }
}

public class Temperature
{
    int temperature = 0;
    public void showTemp()
    {
      if(temperature == 0)
      {
        throw (new TempIsZeroException("Zero Temperature found"));
      }
      else
      {
        Console.WriteLine("Temperature: {0}", temperature);
      }
    }
}
```

编译执行上述代码，得到如下结果：

```
TempIsZeroException: Zero Temperature found
```

抛出对象

如果某个对象是直接或间接地继承自 System.Exception 类，你可以抛出这个对象。你可以在 catch 语句块中用 throw 语句抛出这个对象：

```
Catch(Exception e)
{
    ...
    Throw e
}
```

文件 I/O

文件是存储在磁盘具有特定名称和目录路径的数据的集合。当一个文件被打开阅读或书写时，就变成了流。

流基本上是通过通信路径中的字节顺序。主要有两个流：输入流和输出流。输入流用于从文件系统中读取数据，输出流用于向文件中写数据。

I/O 类

System.IO 的命名空间有多种类，这些类被用于执行大量和文件有关的操作，例如创建和删除文件，读写文件，关闭文件等等。

下面的表格中列出了一些 System.IO 命名空间中常用的非抽象类：

I/O 类	描述
BinaryReader	从二进制流读取原始数据
BinaryWriter	以二进制形式写入原始数据
BufferedStream	字节流的临时存储
Directory	用于操作目录结构
DirectoryInfo	用于创建复合条件指令。
DriveInfo	用于执行目录操作
File	用于操作文件
FileInfo	用于执行文件操作
FileStream	用于读写文件中任意位置的内容
MemoryStream	用于随机存取存储器中存储的流数据
Path	用于执行有关路径信息的操作
StreamReader	用于从字节流中读取字符
StreamWriter	用于向流中写字符
StringReader	用于读取字符串数组
StringWriter	用于写入字符串数组

FileStream 类

System.IO 命名空间中的 FileStream 类有助于读取，写入和关闭文件。这个类派生自抽象类流。

你需要创建一个 FileStream 对象用于创建一个新的文件或打开一个已存在的文件。创建 FileStream 对象的语法如下：

```
FileStream <object_name> = new FileStream( <file_name>, <FileMode Enumerator>, <FileAccess Enumerator>, <File
```

例如，创建一个 FileStream 对象F，读取一个名为 sample.txt 的文件的方法如下：

```
FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read, FileShare.Read);
```

参数	描述
FileMode	fileMode 枚举定义了各种方法来打开文件。fileMode 枚举的成员是： Append: 打开一个已有的文件，并将光标放置在文件的末尾。如果文件不存在，则创建文件。 Create: 它创建一个新的文件 CreateNew: 指定操作系统应创建一个新的文件。如果文件已存在，则抛出异常。 Open: 它会打开一个现有的文件 OpenOrCreate: 指定操作系统应打开一个已有的文件。如果文件不存在，则用指定的名称创建一个新的文件打开。 Truncate: 打开一个已有的文件，文件一旦打开，就将被截断为零字节大小。
FileAccess	FileAccess 枚举成员有：Read，ReadWrite 和 Write。
FileShare	FileShare 枚举有以下成员： Inheritable: 允许文件句柄可由子进程继承。 None: 它拒绝共享当前文件 Read: 它允许打开文件进行读取 ReadWrite: 它允许打开文件进行读取和写入 Write: 它允许打开文件写入

示例

下面的程序示范了 FileStream 类：

```
using System;
using System.IO;

namespace FileIOApplication
{
    class Program
    {
      static void Main(string[] args)
      {
        FileStream F = new FileStream("test.dat", FileMode.OpenOrCreate, FileAccess.ReadWrite);
        for (int i = 1; i <= 20; i++)
        {
          F.WriteByte((byte)i);
        }
```

```
            F.Position = 0;
            for (int i = 0; i <= 20; i++)
            {
              Console.Write(F.ReadByte() + " ");
            }
            F.Close();
            Console.ReadKey();
          }
        }
    }
```

编译执行上述代码，得到如下结果：

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1
```

C# 中的高级文件操作

上面的实例演示了 C# 中简单的文件操作。但是，要充分利用 C# System.IO 类的强大功能，您需要知道这些类常用的属性和方法。

主题和描述
文本文件的读写 它涉及到文本文件的读写。StreamReader 和 StreamWriter 类有助于完成文本文件的读写。
二进制文件的读写 它涉及到二进制文件的读写。BinaryReader 和 BinaryWriter 类有助于完成二进制文件的读写。
Windows文件系统的操作 它让 C# 程序员能够浏览并定位 Windows 文件和目录。

2

C# 高级篇

特性

特性（Attribute）是用于在运行时传递程序中各种元素（比如类、方法、结构、枚举、组件等）的行为信息的声明性标签。您可以通过使用特性向程序添加声明性信息。一个声明性标签是通过放置在它所应用的元素前面的方括号（[ ]）来描述的。

特性（Attribute）用于添加元数据，如编译器指令和注释、描述、方法、类等其他信息。.Net 框架提供了两种类型的特性：预定义特性和自定义特性。

列举特性

列举特性的语法如下：

```
[attribute(positional_parameters, name_parameter = value, ...)]
element
```

特性的名称和值都是在方括号内规定的，放在这个特性应用的元素之前。表示位置的参数规定出特性的基本信息，名称参数规定了可选择的信息。

预定义特性

.Net Framework 提供了三种预定义的特性：

- AttributeUsage
- Conditional
- Obsolete

AttributeUsage

该特性描述了用户定义的特性类是如何使用的。它规定了某个特性应用的项目类型。

规定这种特性的语法如下：

```
[AttributeUsage(
    validon,
    AllowMultiple=allowmultiple,
```

```
    Inherited=inherited
)]
```

其中，

- 参数 validon 规定特性了能承载特性的语言元素。它是枚举器 AttributeTargets 的值的组合。默认值是 AttributeTargets.All。
- 参数 allowmultiple（可选的）为该特性的 AllowMultiple 属性提供了一个布尔值。如果为 true，则该特性是多用的。默认值是 false（单用的）。
- 参数 inherited（可选的）为该特性的 Inherited 属性提供一个布尔值。如果为 true，则该特性可被派生类继承。默认值是 false（不可继承）。

例如：

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]
```

Conditional

这个预定义特性标记了一个条件方法，其执行依赖于特定的预处理标识符。

它会引起方法调用的条件编译，取决于指定的值，比如 Debug 或 Trace。例如，当调试代码时显示变量的值。

规定该特性的语法如下：

```
[Conditional(
    conditionalSymbol
)]
```

例如：

```
[Conditional("DEBUG")]
```

下面的实例演示了该特性：

```
#define DEBUG
using System;
using System.Diagnostics;
```

```
public class Myclass
{
    [Conditional("DEBUG")]
    public static void Message(string msg)
    {
      Console.WriteLine(msg);
    }
}

class Test
{
    static void function1()
    {
      Myclass.Message("In Function 1.");
      function2();
    }
    static void function2()
    {
      Myclass.Message("In Function 2.");
    }

    public static void Main()
    {
      Myclass.Message("In Main function.");
      function1();
      Console.ReadKey();
    }
}
```

编译执行上述代码，得到如下结果：

```
In Main function
In Function 1
In Function 2
```

Obsolete

这个预定义特性标记了不应被使用的程序实体。它可以让您通知编译器丢弃某个特定的目标元素。例如，当一个新方法被用在一个类中，但是您仍然想要保持类中的旧方法，您可以通过显示一个应该使用新方法，而不是旧方法的消息，来把它标记为 obsolete（过时的）。

规定该特性的语法如下：

```
[Obsolete(
    message
)]
[Obsolete(
    message,
    iserror
)]
```

其中，

- 参数 message，是一个字符串，描述项目为什么过时的原因以及该替代使用什么。
- 参数 iserror，是一个布尔值。如果该值为 true，编译器应把该项目的使用当作一个错误。默认值是 false（编译器生成一个警告）。

下面的实例演示了该特性：

```
using System;

public class MyClass
{
    [Obsolete("Don't use OldMethod, use NewMethod instead", true)]
    static void OldMethod()
    {
      Console.WriteLine("It is the old method");
    }
    static void NewMethod()
    {
      Console.WriteLine("It is the new method");
    }
    public static void Main()
    {
      OldMethod();
    }
}
```

当你执行这个程序时，编译器会提示如下的错误：

```
Don't use OldMethod, use NewMethod instead
```

创建自定义的特性

.Net 框架允许创建自定义特性，用于存储声明性的信息，且可在运行时被检索。该信息根据设计标准和应用程序需要，可与任何目标元素相关。

创建并使用自定义特性包含四个步骤：

- 声明自定义特性
- 构建自定义特性
- 在目标程序元素上应用自定义特性
- 通过反射访问特性

最后一个步骤包含编写一个简单的程序来读取元数据以便查找各种符号。元数据是用于描述其他数据的数据和信息。该程序应使用反射来在运行时访问特性。我们将在下一章详细讨论这点。

声明自定义特性

一个新的自定义特性应派生自 System.Attribute 类。例如：

```
//一个自定义的特性BugFix被分配给类和类的成员
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
```

在上面的代码中，我们已经声明了一个名为 DeBugInfo 的自定义特性。

构建自定义特性

让我们构建一个名为 DeBugInfo 的自定义特性，该特性将存储调试程序获得的信息。它存储下面的信息：

- bug 的代码编号
- 辨认该 bug 的开发人员名字
- 最后一次审查该代码的日期
- 一个存储了开发人员标记的字符串消息

我们的 DeBugInfo 类将带有三个用于存储前三个信息的私有属性（property）和一个用于存储消息的公有属性（property）。所以 bug 编号、开发人员名字和审查日期将是 DeBugInfo 类的必需的定位（ positional）参数，消息将是一个可选的命名（named）参数。

每个特性必须至少有一个构造函数。必需的定位（ positional ）参数应通过构造函数传递。下面的代码演示了 DeBugInfo 类：

```
//一个自定义的特性BugFix被分配给类和类的成员
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
{
    private int bugNo;
    private string developer;
    private string lastReview;
    public string message;

    public DeBugInfo(int bg, string dev, string d)
    {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
    }

    public int BugNo
    {
      get
      {
        return bugNo;
      }
    }

    public string Developer
    {
      get
      {
        return developer;
      }
    }

    public string LastReview
    {
      get
      {
```

```
            return lastReview;
        }
    }

    public string Message
    {
        get
        {
            return message;
        }
        set
        {
            message = value;
        }
    }
}
```

应用自定义特性

通过将特性放置在目标之前来使用它：

```
[DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]
[DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]
class Rectangle
{
    //成员变量
    protected double length;
    protected double width;
    public Rectangle(double l, double w)
    {
        length = l;
        width = w;
    }
    [DeBugInfo(55, "Zara Ali", "19/10/2012", Message = "Return type mismatch")]

    public double GetArea()
    {
        return length * width;
    }
    [DeBugInfo(56, "Zara Ali", "19/10/2012")]

    public void Display()
    {
        Console.WriteLine("Length: {0}", length);
```

```
        Console.WriteLine("Width: {0}", width);
        Console.WriteLine("Area: {0}", GetArea());
    }
}
```

在下一章节中，我们会介绍如何使用 Reflection 类来检索特性信息。

反射

反射（Reflection）对象用于在运行时获取类型信息。该类位于 System.Reflection 命名空间中，可访问一个正在运行的程序的元数据。

System.Reflection 命名空间包含了允许您获取有关应用程序信息及向应用程序动态添加类型、值和对象的类。

反射的应用

反射（Reflection）有下列用途：

- 它允许在运行时查看属性（attribute）信息。
- 它允许审查集合中的各种类型，以及实例化这些类型。
- 它允许延迟绑定的方法和属性（property）。
- 它允许在运行时创建新类型，然后使用这些类型执行一些任务。

查看元数据

我们已经在上面的章节中提到过，使用反射（Reflection）可以查看属性（attribute）信息。

System.Reflection 类的 MemberInfo 对象需要被初始化，用于发现与类相关的属性（attribute）。为了做到这点，您可以定义目标类的一个对象，如下：

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

下面的程序示范了这点：

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : System.Attribute
{
      public readonly string Url;

      public string Topic   // Topic 是一个表示名字的参数
      {
        get
        {
```

```
            return topic;
        }
        set
        {
            topic = value;
        }
    }

    public HelpAttribute(string url)   // url 是一个表示位置的参数
    {
        this.Url = url;
    }
    private string topic;
}

[HelpAttribute("Information on the class MyClass")]
class MyClass
{
}
namespace AttributeAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Reflection.MemberInfo info = typeof(MyClass);
            object[] attributes = info.GetCustomAttributes(true);
            for (int i = 0; i < attributes.Length; i++)
            {
                System.Console.WriteLine(attributes[i]);
            }

            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会显示附加到类 MyClass 上的自定义属性：

```
HelpAttribute
```

示例

在本实例中，我们将使用在上一章中创建的 DeBugInfo 属性，并使用反射（Reflection）来读取 Rectangle 类中的元数据。

```
using System;
using System.Reflection;

namespace BugFixApplication
{
    //自定义特性BugFix分配给一个类和他的成员
    [AttributeUsage(AttributeTargets.Class |
    AttributeTargets.Constructor |
    AttributeTargets.Field |
    AttributeTargets.Method |
    AttributeTargets.Property,
    AllowMultiple = true)]

    public class DeBugInfo : System.Attribute
    {
      private int bugNo;
      private string developer;
      private string lastReview;
      public string message;

    public DeBugInfo(int bg, string dev, string d)
    {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
    }

    public int BugNo
    {
      get
      {
        return bugNo;
      }
    }

    public string Developer
    {
      get
```

```
      {
        return developer;
      }
    }

public string LastReview
{
  get
  {
    return lastReview;
  }
}

public string Message
{
  get
  {
    return message;
  }
  set
  {
    message = value;
  }
}
 }
 [DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]
 [DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]

 class Rectangle
 {
//成员变量
protected double length;
protected double width;
public Rectangle(double l, double w)
{
  length = l;
  width = w;
}
[DeBugInfo(55, "Zara Ali", "19/10/2012", Message = "Return type mismatch")]
public double GetArea()
{
  return length * width;
}
[DeBugInfo(56, "Zara Ali", "19/10/2012")]
public void Display()
```

```
{
  Console.WriteLine("Length: {0}", length);
  Console.WriteLine("Width: {0}", width);
  Console.WriteLine("Area: {0}", GetArea());
}
 }//Rectangle 类的结束

 class ExecuteRectangle
 {
static void Main(string[] args)
{
  Rectangle r = new Rectangle(4.5, 7.5);
  r.Display();
  Type type = typeof(Rectangle);

  //遍历 Rectangle 类的属性
  foreach (Object attributes in type.GetCustomAttributes(false))
  {
    DeBugInfo dbi = (DeBugInfo)attributes;
    if (null != dbi)
    {
      Console.WriteLine("Bug no: {0}", dbi.BugNo);
      Console.WriteLine("Developer: {0}", dbi.Developer);
      Console.WriteLine("Last Reviewed: {0}", dbi.LastReview);
      Console.WriteLine("Remarks: {0}", dbi.Message);
    }
  }

  //遍历方法属性
  foreach (MethodInfo m in type.GetMethods())
  {
    foreach (Attribute a in m.GetCustomAttributes(true))
    {
      DeBugInfo dbi = (DeBugInfo)a;
      if (null != dbi)
      {
        Console.WriteLine("Bug no: {0}, for Method: {1}", dbi.BugNo, m.Name);
        Console.WriteLine("Developer: {0}", dbi.Developer);
        Console.WriteLine("Last Reviewed: {0}", dbi.LastReview);
        Console.WriteLine("Remarks: {0}", dbi.Message);
      }
    }
  }

  Console.ReadLine();
```

```
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Length: 4.5
Width: 7.5
Area: 33.75
Bug No: 49
Developer: Nuha Ali
Last Reviewed: 10/10/2012
Remarks: Unused variable
Bug No: 45
Developer: Zara Ali
Last Reviewed: 12/8/2012
Remarks: Return type mismatch
Bug No: 55, for Method: GetArea
Developer: Zara Ali
Last Reviewed: 19/10/2012
Remarks: Return type mismatch
Bug No: 56, for Method: Display
Developer: Zara Ali
Last Reviewed: 19/10/2012
Remarks:
```

属性

属性是类、结构体和接口的命名成员。类或结构体中的成员变量或方法称为域。属性是域的扩展，且可使用相同的语法来访问。它们使用访问器让私有域的值可被读写或操作。

属性不会确定存储位置。相反，它们具有可读写或计算它们值的访问器。

例如，有一个名为 Student 的类，带有 age、name 和 code 的私有域。我们不能在类的范围以外直接访问这些域，但是我们可以拥有访问这些私有域的属性。

访问器

属性的访问器包含有助于获取（读取或计算）或设置（写入）属性的可执行语句。访问器声明可包含一个 get 访问器、一个 set 访问器，或者同时包含二者。例如：

```
// 为字符类型声明一个叫 Code 的属性：
public string Code
{
    get
    {
      return code;
    }
    set
    {
      code = value;
    }
}

// 为字符类型声明一个叫 Name 的属性：
public string Name
{
    get
    {
      return name;
    }
    set
    {
      name = value;
    }
}
```

```
// 为整形类型声明一个叫 Age 的属性:
public int Age
{
      get
      {
        return age;
      }
      set
      {
        age = value;
      }
}
```

示例

下面的程序说明了特性是如何使用的:

```
using System;
namespace tutorialspoint
{
   class Student
   {
  private string code = "N.A";
  private string name = "not known";
  private int age = 0;

  // 为字符类型声明一个叫 Code 的属性:
  public string Code
  {
     get
     {
       return code;
     }
     set
     {
       code = value;
     }
  }

  // 为字符类型声明一个叫 Name 的属性:
  public string Name
  {
     get
     {
```

```
      return name;
    }
    set
    {
      name = value;
    }
  }

  // 为整形类型声明一个叫 Age 的属性：
  public int Age
  {
    get
    {
      return age;
    }
    set
    {
      age = value;
    }
  }
  public override string ToString()
  {
    return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
  }
   }

   class ExampleDemo
   {
  public static void Main()
  {

    // 创建一个 Student 类的对象
    Student s = new Student();

    // 为 student 对象设置 code，name 和 age
    s.Code = "001";
    s.Name = "Zara";
    s.Age = 9;
    Console.WriteLine("Student Info: {0}", s);

    //为 age 加 1
    s.Age += 1;
    Console.WriteLine("Student Info: {0}", s);
    Console.ReadKey();
  }
```

```
    }
}
```

编译执行上述代码，得到如下结果：

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

抽象属性

抽象类可拥有抽象属性，这些属性应在派生类中被实现。下面的程序说明了这点：

```
using System;
namespace tutorialspoint
{
   public abstract class Person
   {
  public abstract string Name
  {
     get;
     set;
  }
  public abstract int Age
  {
     get;
     set;
  }
   }

   class Student : Person
   {
  private string code = "N.A";
  private string name = "N.A";
  private int age = 0;

  // 为字符类型声明一个叫 Code 的属性：
  public string Code
  {
     get
     {
        return code;
     }
     set
     {
```

```
      code = value;
   }
}

// 为字符类型声明一个叫 Name 的属性:
public override string Name
{
   get
   {
      return name;
   }
   set
   {
      name = value;
   }
}

// 为整形类型声明一个叫 Age 的属性:
public override int Age
{
   get
   {
      return age;
   }
   set
   {
      age = value;
   }
}
public override string ToString()
{
   return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
}
 }

 class ExampleDemo
 {
public static void Main()
{
   // 创建一个 Student 类的对象
   Student s = new Student();

   // 为 student 对象设置 code，name 和 age
   s.Code = "001";
   s.Name = "Zara";
```

```
        s.Age = 9;
        Console.WriteLine("Student Info:- {0}", s);

        //为age加1
        s.Age += 1;
        Console.WriteLine("Student Info:- {0}", s);
        Console.ReadKey();
    }
     }
}
```

编译执行上述代码，得到如下结果：

```
Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10
```

索引器

创建**索引器**可以使一个对象像数组一样被索引。为类定义索引器时，该类的行为类似于一个**虚拟数组**，使用数组访问运算符([ ])则可以对该类来进行访问。

句法规则

创建一个一维索引器的规则如下：

```
element-type this[int index]
{
  // get 访问器
  get
  {
    // 返回 index 指定的值
  }

  // set 访问器
  set
  {
    // 设置 index 指定的值
  }
}
```

使用索引器

索引器的声明在某种程度上类似于属性的声明，例如，使用 get 和 set 方法来定义一个索引器。不同的是，属性值的定义要求返回或设置一个特定的数据成员，而索引器的定义要求返回或设置的是某个对象实例的一个值，即索引器将实例数据切分成许多部分，然后通过一些方法去索引、获取或是设置每个部分。

定义属性需要提供属性名，而定义索引器需要提供一个指向对象实例的 this 关键字。

示例：

```
using System;
namespace IndexerApplication
{
  class IndexedNames
  {
```

```
private string[] namelist = new string[size];
static public int size = 10;
public IndexedNames()
{
  for (int i = 0; i < size; i++)
  namelist[i] = "N. A.";
}

public string this[int index]
{
  get
  {
    string tmp;

    if( index >= 0 && index <= size-1 )
    {
      tmp = namelist[index];
    }
    else
    {
      tmp = "";
    }

    return ( tmp );
  }
  set
  {
    if( index >= 0 && index <= size-1 )
    {
      namelist[index] = value;
    }
  }
}

static void Main(string[] args)
{
  IndexedNames names = new IndexedNames();
  names[0] = "Zara";
  names[1] = "Riz";
  names[2] = "Nuha";
  names[3] = "Asif";
  names[4] = "Davinder";
  names[5] = "Sunil";
  names[6] = "Rubic";
  for ( int i = 0; i < IndexedNames.size; i++ )
```

```
      {
        Console.WriteLine(names[i]);
      }

      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
```

重载索引器

索引器允许重载，即允许使用多种不同类型的参数来声明索引器。索引值可以是整数，但也可以是其他的数据类型，如字符型。

重载索引器示例：

```
using System;
namespace IndexerApplication
{
  class IndexedNames
  {
    private string[] namelist = new string[size];
    static public int size = 10;
    public IndexedNames()
    {
      for (int i = 0; i < size; i++)
      {
        namelist[i] = "N. A.";
      }
    }
```

```
public string this[int index]
{
  get
  {
    string tmp;

    if( index >= 0 && index <= size-1 )
    {
      tmp = namelist[index];
    }
    else
    {
      tmp = "";
    }

    return ( tmp );
  }
  set
  {
    if( index >= 0 && index <= size-1 )
    {
      namelist[index] = value;
    }
  }
}
public int this[string name]
{
  get
  {
    int index = 0;
    while(index < size)
    {
      if (namelist[index] == name)
      {
       return index;
      }
      index++;
    }
    return index;
  }

}

static void Main(string[] args)
{
```

```
        IndexedNames names = new IndexedNames();
        names[0] = "Zara";
        names[1] = "Riz";
        names[2] = "Nuha";
        names[3] = "Asif";
        names[4] = "Davinder";
        names[5] = "Sunil";
        names[6] = "Rubic";

        // 使用带有 int 参数的第一个索引器
        for (int i = 0; i < IndexedNames.size; i++)
        {
          Console.WriteLine(names[i]);
        }

        // 使用带有 string 参数的第二个索引器
        Console.WriteLine(names["Nuha"]);
        Console.ReadKey();
      }
    }
}
```

编译执行上述代码，得到如下结果：

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
2
```

委托

C# 中的委托类似于 C 或 C++ 中指向函数的指针。**委托**表示引用某个方法的引用类型变量，运行时可以更改引用对象。

特别地，委托可以用于处理事件或回调函数。并且，所有的委托类都是从 System.Delegate 类继承而来。

声明委托

声明委托时，需要定义能够被委托所引用的方法，任意委托可以引用与该委托拥有相同签名的方法。如：

```
public delegate int MyDelegate (string s);
```

上述委托可以用于引用任何一个以字符型为参数的方法，且返回值类型为整型。

声明委托的句法规则为：

```
delegate <return type> <delegate-name> <parameter list>
```

实例化委托

声明委托之后，必须使用 new 关键字和一个特定的方法来创建一个委托对象。创建时，传递到 new 语句的参数写法与方法调用相同，但是不带有参数，例如：

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

下述示例演示了委托的声明、实例化，此处的委托用于引用一个带有一个整型参数的方法，且该方法返回一个整型值。

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
{
   class TestDelegate
   {
      static int num = 10;
```

```
    public static int AddNum(int p)
    {
      num += p;
      return num;
    }

    public static int MultNum(int q)
    {
      num *= q;
      return num;
    }
    public static int getNum()
    {
      return num;
    }

    static void Main(string[] args)
    {
      // 创建委托实例
      NumberChanger nc1 = new NumberChanger(AddNum);
      NumberChanger nc2 = new NumberChanger(MultNum);

      // 使用委托对象调用方法
      nc1(25);
      Console.WriteLine("Value of Num: {0}", getNum());
      nc2(5);
      Console.WriteLine("Value of Num: {0}", getNum());
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Value of Num: 35
Value of Num: 175
```

委托的多播

委托对象可通过 "+" 运算符进行合并。一个合并委托可以调用它所合并的两个委托，但只有相同类型的委托可被合并。"-" 运算符则可用于从合并的委托中移除其中一个委托。

利用委托的这种特性，可以创建一个委托被调用时所涉及的方法的调用列表。这被称为委托的**多播**，也叫组播。下面的程序演示了委托的多播：

```
using System;

delegate int NumberChanger(int n);
namespace DelegateAppl
{
  class TestDelegate
  {
    static int num = 10;
    public static int AddNum(int p)
    {
      num += p;
      return num;
    }

    public static int MultNum(int q)
    {
      num *= q;
      return num;
    }

    public static int getNum()
    {
      return num;
    }

    static void Main(string[] args)
    {
      // 创建委托实例
      NumberChanger nc;
      NumberChanger nc1 = new NumberChanger(AddNum);
      NumberChanger nc2 = new NumberChanger(MultNum);
      nc = nc1;
      nc += nc2;

      // 调用多播
      nc(5);
      Console.WriteLine("Value of Num: {0}", getNum());
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Value of Num: 75
```

委托的使用

下面的示例演示了委托的作用，示例中的 *printString* 委托可用于引用带有一个字符串作为输入的方法，且不返回数据。

我们使用这个委托来调用两个方法，第一个方法将字符串输出到控制台，第二个方法将字符串输出到文件：

```
using System;
using System.IO;

namespace DelegateAppl
{
  class PrintString
  {
    static FileStream fs;
    static StreamWriter sw;

    // 委托声明
    public delegate void printString(string s);

    // 该方法打印到控制台
    public static void WriteToScreen(string str)
    {
      Console.WriteLine("The String is: {0}", str);
    }

    // 该方法打印到文件
    public static void WriteToFile(string s)
    {
      fs = new FileStream("c:\\message.txt",
      FileMode.Append, FileAccess.Write);
      sw = new StreamWriter(fs);
      sw.WriteLine(s);
      sw.Flush();
      sw.Close();
      fs.Close();
    }

    // 该方法把委托作为参数，并使用它调用方法
    // call the methods as required
    public static void sendString(printString ps)
    {
      ps("Hello World");
```

```
    }
    static void Main(string[] args)
    {
      printString ps1 = new printString(WriteToScreen);
      printString ps2 = new printString(WriteToFile);
      sendString(ps1);
      sendString(ps2);
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
The String is: Hello World
```

事件

事件指一个用户操作，如按键、点击、移动鼠标等，也可以是系统生成的通知。当事件发生时，应用需要对其作出相应的反应，如中断。另外，事件也用于内部进程通信。

通过事件使用委托

事件生成于类的声明中，通过使用同一个类或其他类中的委托与事件处理程序关联。包含事件的类用于发布事件，称为**发布器**类。其他接受该事件的类称为**订阅器**类。事件使用的是发布-订阅（publisher-subscriber）模型。

发布器是一个定义了事件和委托的对象，此外还定义了事件和委托之间的联系。一个发布器类的对象调用这个事件，同时通知其他的对象。

订阅器是一个接受事件并提供事件处理程序的对象。在发布器类中的委托调用订阅器类中的方法（或事件处理程序）。

声明事件

在类中声明一个事件，首先需要声明该事件对应的委托类型。如：

```
public delegate void BoilerLogHandler(string status);
```

其次为使用 event 关键字来声明这个事件：

```
//基于上述委托定义事件
public event BoilerLogHandler BoilerEventLog;
```

上述代码段定义了一个名为 *BoilerLogHandler* 的委托以及一个名为 *BoilerEventLog* 的事件，该事件生成时会自动调用委托。

示例 1

```
using System;
namespace SimpleEvent
{
  using System;
```

```
  public class EventTest
  {
    private int value;
    public delegate void NumManipulationHandler();
    public event NumManipulationHandler ChangeNum;
    protected virtual void OnNumChanged()
    {
      if (ChangeNum != null)
      {
        ChangeNum();
      }
      else
      {
        Console.WriteLine("Event fired!");
      }
    }

    public EventTest(int n )
    {
      SetValue(n);
    }

    public void SetValue(int n)
    {
      if (value != n)
      {
        value = n;
        OnNumChanged();
      }
    }
  }

  public class MainClass
  {
    public static void Main()
    {
      EventTest e = new EventTest(5);
      e.SetValue(7);
      e.SetValue(11);
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Event Fired!
Event Fired!
Event Fired!
```

示例 2

该示例为一个简单的应用程序，该程序用于热水锅炉系统故障排除。当维修工程师检查锅炉时，锅炉的温度、压力以及工程师所写的备注都会被自动记录到一个日志文件中。

```
using System;
using System.IO;

namespace BoilerEventAppl
{
  // boiler 类
  class Boiler
  {
    private int temp;
    private int pressure;
    public Boiler(int t, int p)
    {
      temp = t;
      pressure = p;
    }

    public int getTemp()
    {
      return temp;
    }

    public int getPressure()
    {
      return pressure;
    }
  }

  // 事件发布器
  class DelegateBoilerEvent
  {
    public delegate void BoilerLogHandler(string status);

    // 基于上述委托定义事件
    public event BoilerLogHandler BoilerEventLog;
```

```
  public void LogProcess()
  {
    string remarks = "O. K";
    Boiler b = new Boiler(100, 12);
    int t = b.getTemp();
    int p = b.getPressure();
    if(t > 150 || t < 80 || p < 12 || p > 15)
    {
      remarks = "Need Maintenance";
    }
    OnBoilerEventLog("Logging Info:\n");
    OnBoilerEventLog("Temparature " + t + "\nPressure: " + p);
    OnBoilerEventLog("\nMessage: " + remarks);
  }

  protected void OnBoilerEventLog(string message)
  {
    if (BoilerEventLog != null)
    {
      BoilerEventLog(message);
    }
  }
}

// 该类保留写入日志文件的条款
class BoilerInfoLogger
{
  FileStream fs;
  StreamWriter sw;
  public BoilerInfoLogger(string filename)
  {
    fs = new FileStream(filename, FileMode.Append, FileAccess.Write);
    sw = new StreamWriter(fs);
  }

  public void Logger(string info)
  {
    sw.WriteLine(info);
  }

  public void Close()
  {
    sw.Close();
    fs.Close();
  }
```

```
  }

  // 事件订阅器
  public class RecordBoilerInfo
  {
    static void Logger(string info)
    {
      Console.WriteLine(info);
    }//end of Logger

    static void Main(string[] args)
    {
      BoilerInfoLogger filelog = new BoilerInfoLogger("e:\\boiler.txt");
      DelegateBoilerEvent boilerEvent = new DelegateBoilerEvent();
      boilerEvent.BoilerEventLog += new
      DelegateBoilerEvent.BoilerLogHandler(Logger);
      boilerEvent.BoilerEventLog += new
      DelegateBoilerEvent.BoilerLogHandler(filelog.Logger);
      boilerEvent.LogProcess();
      Console.ReadLine();
      filelog.Close();
    }//end of main

  }//end of RecordBoilerInfo
}
```

编译执行上述代码，得到如下结果：

```
Logging info:

Temperature 100
Pressure 12

Message: O. K
```

集合

集合类专门用于数据存储和数据检索，并提供堆栈、队列、列表和哈希表的支持。目前，大多数集合类都实现了相同的接口。

集合类服务于不同的目的，如为元素动态分配内存，基于索引访问列表项等等，这些类所创建的是 Object 类的对象的集合。在 C# 中，Object 类是所有数据类型的基类。

各种集合类及其用法

下表为一些常用的以 System.Collection 为命名空间的集合类，点击相应链接，可查看详细说明。

类	描述及用法
动态数组	动态数组表示可被单独**索引**的对象的有序集合。 动态数组基本上可以替代数组，但与数组不同的是，通过**索引**，动态数组可以在指定的位置添加和移除项目，且会自动重新调整大小，同样允许在列表中进行动态内存分配、增加、搜索、排序各项。
哈希表	哈希表使用**键**来访问集合中的元素。 当需要通过键访问元素时，则使用哈希表，且一个有用的键值可以很方便地被识别。哈希表中的每一项都有一个**键/值**对。键用于访问集合中的项目。
排序列表	排序列表使用**键**和**索引**来访问列表中的项。 它是数组和哈希表的组合，包含一个可使用键或索引访问各项的列表。若使用索引来访问各项，则它为一个动态数组，若使用键来访问各项，则它为一个哈希表。集合中的各项总是按键值排序。
堆栈	堆栈表示的是一个**后进先出**的对象集合。 当需要对各项进行后进先出的访问时，则使用堆栈。在列表中添加一项，称为**推入**元素；从列表中移除一项时，称为**弹出**元素。
队列	队列表示的是一个**先进先出**的对象集合。 当需要对各项进行先进先出的访问时，则使用队列。在列表中添加一项，称为**入队**；从列表中移除一项，称为**出队**。
点阵列	点阵列表示的是一个使用值 1 和 0 来表示的**二进制数组**。 当需要存储位，但事先不知道位数时，则使用点阵列。通过**整型索引**，可以从点阵列集合中访问各项，该索引值从零开始。

泛型

泛型允许推迟类或方法中编程元素的数据类型规范的编写，直到实际在程序中使用它的时候再编写。换句话说，泛型允许编写一个可以与任何数据类型协作的类或方法。

你可以通过数据类型的替代参数来编写类或方法的规范。当编译器遇到类的构造函数或方法的函数调用时，它会生成代码来处理指定的数据类型。下面这个简单的示例将有助于理解这个概念：

```
using System;
using System.Collections.Generic;

namespace GenericApplication
{
  public class MyGenericArray<T>
  {
    private T[] array;
    public MyGenericArray(int size)
    {
      array = new T[size + 1];
    }

    public T getItem(int index)
    {
      return array[index];
    }

    public void setItem(int index, T value)
    {
      array[index] = value;
    }
  }

  class Tester
  {
    static void Main(string[] args)
    {

      // 声明一个整型数组
      MyGenericArray<int> intArray = new MyGenericArray<int>(5);

      // 设置值
      for (int c = 0; c < 5; c++)
```

```
        {
          intArray.setItem(c, c*5);
        }

        // 获取值
        for (int c = 0; c < 5; c++)
        {
          Console.Write(intArray.getItem(c) + " ");
        }

        Console.WriteLine();

        // 声明一个字符数组
        MyGenericArray<char> charArray = new MyGenericArray<char>(5);

        // 设置值
        for (int c = 0; c < 5; c++)
        {
          charArray.setItem(c, (char)(c+97));
        }

        // 获取值
        for (int c = 0; c< 5; c++)
        {
          Console.Write(charArray.getItem(c) + " ");
        }
        Console.WriteLine();

        Console.ReadKey();
      }
    }
}
```

编译执行上述代码，得到如下结果：

```
0 5 10 15 20
a b c d e
```

泛型的特性

泛型是一种可以增强程序功能的技术，表现在如下几个方面：

？ 它有助于最大限度地进行重用代码、确保类型的安全以及提高性能。

- 你可以创建泛型集合类。.NET 框架类库在 *System.Collections.Generic* 命名空间中包含了一些新的泛型集合类，你可以使用这些泛型集合类来替代 *System.Collections* 中的集合类。
- 你可以创建自己的泛型接口、泛型类、泛型方法、泛型事件和泛型委托。
- 你可以对泛型类进行约束，使其只访问具有特定数据类型的方法。
- 在运行时，通过使用反射方法可以获取泛型数据类型中所使用的类型信息。

泛型方法

在之前的例子中，我们使用过一个泛型类，我们还可以通过类型参数来声明泛型方法。下述示例很好地展示了这个概念：

```
using System;
using System.Collections.Generic;

namespace GenericMethodAppl
{
  class Program
  {
    static void Swap<T>(ref T lhs, ref T rhs)
    {
      T temp;
      temp = lhs;
      lhs = rhs;
      rhs = temp;
    }
    static void Main(string[] args)
    {
      int a, b;
      char c, d;
      a = 10;
      b = 20;
      c = 'I';
      d = 'V';

      // 显示交换之前的值
      Console.WriteLine("Int values before calling swap:");
      Console.WriteLine("a = {0}, b = {1}", a, b);
      Console.WriteLine("Char values before calling swap:");
      Console.WriteLine("c = {0}, d = {1}", c, d);

      // 调用 swap 进行交换
```

```
        Swap<int>(ref a, ref b);
        Swap<char>(ref c, ref d);

        // 显示交换之后的值
        Console.WriteLine("Int values after calling swap:");
        Console.WriteLine("a = {0}, b = {1}", a, b);
        Console.WriteLine("Char values after calling swap:");
        Console.WriteLine("c = {0}, d = {1}", c, d);

        Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:
a = 20, b = 10
Char values after calling swap:
c = V, d = I
```

泛型委托

你可以通过类型参数来定义一个泛型委托，如：

```
delegate T NumberChanger<T>(T n);
```

泛型委托示例：

```
using System;
using System.Collections.Generic;

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl
{
  class TestDelegate
  {
    static int num = 10;
    public static int AddNum(int p)
    {
      num += p;
```

```
        return num;
    }

    public static int MultNum(int q)
    {
        num *= q;
        return num;
    }
    public static int getNum()
    {
        return num;
    }

    static void Main(string[] args)
    {
        // 创建委托实例
        NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);
        NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);

        // 使用委托对象调用方法
        nc1(25);
        Console.WriteLine("Value of Num: {0}", getNum());
        nc2(5);
        Console.WriteLine("Value of Num: {0}", getNum());
        Console.ReadKey();
    }
  }
}
```

编译执行以上代码，得到如下结果：

```
Value of Num: 35
Value of Num: 175
```

匿名方法

先前的章节中提过，委托是用于引用与其具有相同签名的方法，即使用委托对象，就可以调用任何被该委托引用的方法。

匿名方法提供了一种将一段代码块作为委托参数的技术。顾名思义，匿名方法没有名字，只有方法主体。

你不需要为匿名方法指定返回类型，其返回类型直接由方法主体推断而来。

编写匿名方法

匿名方法通过使用 delegate 关键字创建委托实例来实现方法的声明，如：

```
delegate void NumberChanger(int n);
...
NumberChanger nc = delegate(int x)
{
  Console.WriteLine("Anonymous Method: {0}", x);
};
```

上述代码块中的 *Console.WriteLine("Anonymous Method: {0}", x);* 就是匿名方法的主体。

委托可以通过匿名方法调用，也可以通过命名方法调用，即，通过向委托对象来传递方法参数，如：

```
nc(10);
```

示例

```
using System;

delegate void NumberChanger(int n);
namespace DelegateAppl
{
  class TestDelegate
  {
    static int num = 10;
    public static void AddNum(int p)
    {
      num += p;
      Console.WriteLine("Named Method: {0}", num);
    }
```

```
    public static void MultNum(int q)
    {
      num *= q;
      Console.WriteLine("Named Method: {0}", num);
    }

    public static int getNum()
    {
      return num;
    }
    static void Main(string[] args)
    {
      //使用匿名方法创建委托实例
      NumberChanger nc = delegate(int x)
      {
        Console.WriteLine("Anonymous Method: {0}", x);
      };

      //使用匿名方法调用委托
      nc(10);

      //使用命名方法实例化委托
      nc =  new NumberChanger(AddNum);

      //使用命名方法调用委托
      nc(5);

      //使用另一个命名方法实例化委托
      nc =  new NumberChanger(MultNum);

      //使用另一个命名方法调用委托
      nc(2);
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Anonymous Method: 10
Named Method: 15
Named Method: 30
```

不安全代码

当代码段被 unsafe 修饰符标记时，C# 允许该代码段中的函数使用指针变量，故使用了**指针**变量的代码块又被称为**不安全代码**或非托管代码。

> 注意： 若要在 codingground 中执行本章的程序，请将 *Project >> Compile Options >> Compilation Command to* 中的编辑项设置为 mcs *.cs -out:main.exe -unsafe"

指针

指针是指其值为另一个变量的地址的变量，即内存位置的直接地址。如一般的变量或常量一样，在使用指针来存储其他变量的地址之前，必须先进行指针声明。

声明指针变量的一般形式为：

```
type *var-name;
```

以下为一些有效的指针声明示例：

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

下述示例为在 C# 中使用了 unsafe 修饰符时指针的使用：

```
using System;
namespace UnsafeCodeApplication
{
  class Program
  {
    static unsafe void Main(string[] args)
    {
      int var = 20;
      int* p = &var;
      Console.WriteLine("Data is: {0} ", var);
      Console.WriteLine("Address is: {0}", (int)p);
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Data is: 20
Address is: 99215364
```

除了将整个方法声明为不安全代码之外，还可以只将部分代码声明为不安全代码，下面章节中的的示例说明了这点。

使用指针检索数据值

使用 ToString() 方法可以检索存储在指针变量所引用位置的数据。例如：

```
using System;
namespace UnsafeCodeApplication
{
  class Program
  {
    public static void Main()
    {
      unsafe
      {
        int var = 20;
        int* p = &var;
        Console.WriteLine("Data is: {0} " , var);
        Console.WriteLine("Data is: {0} " , p->ToString());
        Console.WriteLine("Address is: {0} " , (int)p);
      }

      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果

```
Data is: 20
Data is: 20
Address is: 77128984
```

传递指针作为方法的参数

指针可以作为方法中的参数，例如：

```
using System;
namespace UnsafeCodeApplication
{
  class TestPointer
  {
    public unsafe void swap(int* p, int *q)
    {
      int temp = *p;
      *p = *q;
      *q = temp;
    }

    public unsafe static void Main()
    {
      TestPointer p = new TestPointer();
      int var1 = 10;
      int var2 = 20;
      int* x = &var1;
      int* y = &var2;

      Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
      p.swap(x, y);

      Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

使用指针访问数组元素

在 C# 中，数组名称与一个指向与数组数据具有相同数据类型的指针是不同的变量类型。例如 int *p 和 int[] 是不同的类型。你可以对指针变量 p 进行加操作，因为它在内存中是不固定的，反之，数组的地址在内存中是固定的，因而无法对其直接进行加操作。

故如同 C 或 C++，这里同样需要使用一个指针变量来访问数组数据，并且需要使用 fixed 关键字来固定指针，示例如下：

```
using System;
namespace UnsafeCodeApplication
{
  class TestPointer
  {
    public unsafe static void Main()
    {
      int[]  list = {10, 100, 200};
      fixed(int *ptr = list)

      /* let us have array address in pointer */
      for ( int i = 0; i < 3; i++)
      {
        Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
        Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
      }

      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

编译不安全代码

必须切换命令行编译器到指定的 /unsafe 命令行才能进行不安全代码的编译。

例如，编译一个包含不安全代码的名为 prog1.cs 的程序，需要在命令行中输入如下命令：

```
csc /unsafe prog1.cs
```

在 Visual Studio IDE 环境中编译不安全代码，需要在项目属性中启用相关设置。

步骤如下：

？ 双击资源管理器中的属性节点，打开**项目属性**。

? 点击 Build 标签页。

? 选择选项 "Allow unsafe code"。

多线程

线程的定义是程序的执行路径。每个线程都定义了一个独特的控制流，如果应用程序涉及到复杂且耗时的操作，那么设置不同的线程执行路径会非常有好处，因为每个线程会被指定于执行特定的工作。

线程实际上是**轻量级进程**。一个常见的使用线程的实例是现代操作系统中的并行编程。使用线程不仅有效地减少了 CPU 周期的浪费，同时还提高了应用程序的运行效率。

到目前为止我们所编写的程序都是以一个单线程作为应用程序的运行的，其运行过程均为单一的。但是，在这种情况下，应用程序在同一时间只能执行一个任务。为了使应用程序可以同时执行多个任务，需要将其被划分为多个更小的线程。

线程的声明周期

线程的生命周期开始于对象的 System.Threading.Thread 类创建时，结束于线程被终止或是完成执行时。 下列各项为线程在生命周期中的各种状态：

- **未启动状态**：线程实例已被创建但 Start 方法仍未被调用时的状态。
- **就绪状态**：线程已准备好运行，正在等待 CPU 周期时的状态。
- **不可运行状态**：下面的几种情况下线程是不可运行的：
  - 已经调用 Sleep 方法
  - 已经调用 Wait 方法
  - 通过 I/O 操作阻塞
- **死亡状态**：线程已完成执行或已终止的状态。

主线程

在 C# 中，System.Threading.Thread 类用于线程的工作。它允许创建并访问多线程应用程序中的单个线程。进程中第一个被执行的线程称为主线程。

当 C# 程序开始执行时，会自动创建主线程。使用 Thread 类创建的线程被主线程的子线程调用。通过 Thread 类的 CurrentThread 属性可以访问线程。

下面的程序演示了主线程的执行：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
  class MainThreadProgram
  {
    static void Main(string[] args)
    {
      Thread th = Thread.CurrentThread;
      th.Name = "MainThread";
      Console.WriteLine("This is {0}", th.Name);
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
This is MainThread
```

Thread 类的属性和方法

下表为 Thread 类一些常用的属性：

属性	描述
CurrentContext	获取线程当前正在执行的线程的上下文。
CurrentCulture	获取或设置当前线程的区域性
CurrentPrinciple	获取或设置线程的当前责任人（针对基于角色的安全性）
CurrentThread	获取当前正在运行的线程
CurrentUICulture	获取或设置资源管理器当前所使用的区域性，便于在运行时查找区域性特定的资源
ExecutionContext	获取一个 ExcutionContext 对象，该对象包含有关当前线程的各种上下文信息。
IsAlive	获取一个值，指示当前线程的执行状态。
IsBackground	获取或设置一个值，指示线程是否为后台线程。
IsThreadPoolThread	获取或设置一个值，指示线程是否属于托管线程池。
ManagedThreadId	获取当前托管线程的唯一标识符
Name	获取或设置线程的名称。
Priority	获取或设置一个值，指示线程的调度优先级
ThreadState	获取一个值，指示当前线程的状态。

下表为 Thread 类一些常用的方法：

序号	方法名和描述
1	public void Abort() 在调用此方法的线程上引发 ThreadAbortException，则触发终止此线程的操作。调用此方法通常会终止线程。
2	public static LocalDataStoreSlot AllocateDataSlot() 在所有的线程上分配未命名的数据槽。使用以 ThreadStaticAttribute 属性标记的字段，可获得更好的性能。
3	public static LocalDataStoreSlot AllocateNamedDataSlot(string name) 在所有线程上分配已命名的数据槽。使用以 ThreadStaticAttribute 属性标记的字段，可获得更好的性能。
4	public static void BeginCriticalRegion() 通知主机将要进入一个代码区域，若该代码区域内的线程终止或发生未经处理的异常，可能会危害应用程序域中的其他任务。
5	public static void BeginThreadAffinity() 通知主机托管代码将要执行依赖于当前物理操作系统线程的标识的指令。
6	public static void EndCriticalRegion() 通知主机将要进入一个代码区域，若该代码区域内的线程终止或发送未经处理的异常，仅会影响当前任务。
7	public static void EndThreadAffinity() 通知主机托管代码已执行完依赖于当前物理操作系统线程的标识的指令。
8	public static void FreeNamedDataSlot(string name) 消除进程中所有线程的名称与槽之间的关联。使用以 ThreadStaticAttribute 属性标记的字段，可获得更好的性能。
9	public static Object GetData(LocalDataStoreSlot slot) 在当前线程的当前域中从当前线程上指定的槽中检索值。使用以 ThreadStaticAttribute 属性标记的字段，可获得更好的性能。
10	public static AppDomain GetDomain() 返回当前线程正在其中运行的当前域。
11	public static AppDomain GetDomainID() 返回唯一的应用程序域标识符。
12	public static LocalDataStoreSlot GetNamedDataSlot(string name) 查找已命名的数据槽。使用以 ThreadStaticAttribute 属性标记的字段，可获得更好的性能。
13	public void Interrupt() 中断处于 WaitSleepJoin 线程状态的线程。
14	public void Join() 在继续执行标准的 COM 和 SendMessage 消息泵处理期间，阻塞调用线程，直到某个线程终止为止。此方法有不同的重载形式。
15	public static void MemoryBarrier() 按如下方式同步内存存取：执行当前线程的处理器在对指令进行重新排序时，不能采用先执行 MemoryBarrier 调用之后的内存存取，再执行 MemoryBarrier 调用之前的内存存取的方式。
16	public static void ResetAbort() 取消当前线程请求的 Abort 操作。
17	public static void SetData(LocalDataStoreSlot slot, Object data) 在指定槽中，设置当前正在运行中线程的当前域的数据。使用以 ThreadStaticAttribute 属性标记的字段，可获得更好的性能。

序号	方法名和描述
1 8	public void Start() 开始一个线程。
1 9	public static void Sleep(int millisecondsTimeout) 令线程暂停一段时间。
2 0	public static void SpinWait(int iterations) 令线程等待一段时间，时间长度由 iterations 参数定义。
2 1	public static byte VolatileRead(ref byte address)；public static double VolatileRead(ref double address)；public static int VolatileRead(ref int address)；public static Object VolatileRead(ref Object address) 读取字段的值。无论处理器的数目或处理器缓存的状态如何，该值都表示由计算机中任何一个处理器写入的最新值。此方法有不同的重载形式，此处仅给出部分例子。
2 2	public static void VolatileWrite(ref byte address, byte value)；public static void VolatileWrite(ref double address, double value)；public static void VolatileWrite(ref int address, int value)；public static void VolatileWrite(ref Object address, Object value) 立即写入一个值到字段中，使该值对计算机中的所有处理器都可见。此方法有不同的重载形式，此处仅给出部分例子。
2 3	public static bool Yield() 令调用线程执行已准备好在当前处理器上运行的另一个线程，由操作系统选择要执行的线程。

线程的创建

线程是通过扩展 Thread 类创建的，扩展而来的 Thread 类调用 Start() 方法即可开始子线程的执行。 示例：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
  class ThreadCreationProgram
  {
    public static void CallToChildThread()
    {
      Console.WriteLine("Child thread starts");
    }

    static void Main(string[] args)
    {
      ThreadStart childref = new ThreadStart(CallToChildThread);
      Console.WriteLine("In Main: Creating the Child thread");
      Thread childThread = new Thread(childref);
      childThread.Start();
      Console.ReadKey();
```

```
    }
  }
}
```

编译执行上述代码段，得到如下结果：

```
In Main: Creating the Child thread
Child thread starts
```

线程的管理

Thread 类提供了多种用于线程管理的方法。 下面的示例调用了 sleep() 方法来在一段特定时间内暂停线程：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
  class ThreadCreationProgram
  {
    public static void CallToChildThread()
    {
      Console.WriteLine("Child thread starts");

      // 令线程暂停 5000 毫秒
      int sleepfor = 5000;

      Console.WriteLine("Child Thread Paused for {0} seconds", sleepfor / 1000);
      Thread.Sleep(sleepfor);
      Console.WriteLine("Child thread resumes");
    }

    static void Main(string[] args)
    {
      ThreadStart childref = new ThreadStart(CallToChildThread);
      Console.WriteLine("In Main: Creating the Child thread");
      Thread childThread = new Thread(childref);
      childThread.Start();
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下代码段：

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

线程的销毁

使用 Abort() 方法可销毁线程。

在运行时，通过抛出 ThreadAbortException 来终止线程。这个异常无法被捕获，当且仅当具备 *finally* 块时，才将控制送到 *finally* 块中。

示例：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
  class ThreadCreationProgram
  {
    public static void CallToChildThread()
    {
      try
      {
        Console.WriteLine("Child thread starts");

        // 执行一些任务，如计十个数
        for (int counter = 0; counter <= 10; counter++)
        {
          Thread.Sleep(500);
          Console.WriteLine(counter);
        }

        Console.WriteLine("Child Thread Completed");
      }

      catch (ThreadAbortException e)
      {
        Console.WriteLine("Thread Abort Exception");
      }
      finally
      {
        Console.WriteLine("Couldn't catch the Thread Exception");
```

```
      }
    }

    static void Main(string[] args)
    {
      ThreadStart childref = new ThreadStart(CallToChildThread);
      Console.WriteLine("In Main: Creating the Child thread");
      Thread childThread = new Thread(childref);
      childThread.Start();

      // 将主线程停止一段时间
      Thread.Sleep(2000);

      // 中止子线程
      Console.WriteLine("In Main: Aborting the Child thread");

      childThread.Abort();
      Console.ReadKey();
    }
  }
}
```

编译执行上述代码，得到如下结果：

```
In Main: Creating the Child thread
Child thread starts
0
1
2
In Main: Aborting the Child thread
Thread Abort Exception
Couldn't catch the Thread Exception
```

极客学院
jikexueyuan.com
中国最大的IT职业在线教育平台

更多信息请访问
http://wiki.jikexueyuan.com/project/csharp/