第 3 章

C# 语言基础

在本章节中，读者将学习到以下内容：

- 使用命名空间；
- 变量与常量；
- 声明程序入口点；
- 流程控制。

3.1 命名空间

实例 29 使用 namespace 关键字

【导语】

命名空间有两个作用：一是把各种类型按照用途进行分组，二是解决命名冲突。

第一个作用是将类型归类，例如在. NET 类库中，有一个 System. Security. Cryptography 命名空间，根据其命名，可以知道在该命名空间下面的类型与安全技术有关，并且包含用于加密或解密的 API。

对于第二个作用，假设用户在程序代码声明两个类型，它们的名字都是 P，虽然名字相同，但两个 P 类型的功能是完全不同的。为了解决同名冲突，就可以分别把两个 P 类型放在不同的命名空间下，例如第一个 P 类型放在 N1 命名空间下，全名称为 N1. P，再把第二个 P 类型放在 N2 命名空间下，全名称为 N2. P。这样 N1. P 与 N2. P 就不再发生命名冲突了。

定义命名空间使用 namespace 关键字，定义后就可以将类型放置在命名空间中。

【操作流程】

步骤 1：在 Visual Studio 开发环境中新建控制台应用程序项目。

步骤 2：新建项目后，会自动打开项目模板生成的 Program. cs 文件。从生成的代码中可以看到，默认的命名空间与项目名字相同，例如，用户给项目命名为 Demo，那么代码默认的命名空间同样为 Demo。如代码清单 3-1 所示。

代码清单 3-1 模板生成的命名空间

```
namespace Demo
{
    ...
}
```

在 Demo 命名空间下，有一个 Program 类（用 class 关键字声明），Program 类下面还有一个 Main 方法，它是整个程序的入口点，即应用程序会从 Main 方法开始执行，当退出 Main 方法后，程序也随之退出。完整结构如代码清单 3-2 所示。

代码清单 3-2 模板生成的完整程序结构

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

步骤 3：在项目生成的命名空间外（即命名空间的右大括号外）另起新行，使用 namespace 关键字声明一个 Test 命名空间。

```
namespace Test
{
}
```

注意：命名空间是一种容器，里面可以包含类型，属于代码块，因此在命名空间后面要加上一对大括号。

步骤 4：在定义好的 Test 命名空间两个大括号之间定义一个 Car 类。声明类使用 class 关键字，class 也是一种类型。

```
namespace Test
{
    public class Car
    {

    }
}
```

注意：类型内部可以包含类型成员，因此类定义之后也要附加一对大括号。

步骤 5：在 Car 类中再定义一个方法。

```
namespace Test
{
    public class Car
    {
        public void Run()
        {
            Console.WriteLine("开车啦。");
        }
    }
}
```

当调用 Run 方法时，会在控制台窗口输出文本信息。

步骤 6：回到 Program 类的 Main 方法，用以下代码替换默认生成的代码。

```
static void Main(string[] args)
{
    Test.Car c = new Test.Car();
    c.Run();
}
```

上面代码首先声明一个 Car 类型的变量 c，并且通过 new 关键字进行实例化，然后调用 Run 方法。

步骤 7：按 F6 快捷键生成解决方案。

步骤 8：打开“命令提示符”窗口，定位到项目文件目录下的\bin\Debug\netcoreapp<版本号>子目录下。

步骤 9：输入以下命令，执行应用程序。

```
dotnet <项目名称>.dll
```

步骤 10：如果看到输出文本“开车啦”，说明程序已经正确执行。

实例 30　嵌套命名空间

【导语】

命名空间下面不仅可以包含类型，还可以嵌套命名空间。即命名空间 A 下面可以包含命名空间 B，命名空间 B 下面还可以包含命名空间 C。

【操作流程】

步骤 1：新建控制台应用程序，命名为 Demo。

步骤 2：在生成的 Demo 命名空间之外，另声明一个命名空间，命名为 NTest。

```
namespace NTest
{

}
```

步骤 3：在 NTest 命名空间下再声明两个命名空间，分别命名为 NSub1、NSub2。

```
namespace NTest
{
    namespace NSub1
    {

    }

    namespace NSub2
    {

    }
}
```

步骤 4：在 NSub1 命名空间下声明一个类，命名为 WorkTask。

```
class WorkTask
{

}
```

步骤 5：在 NSub2 命名空间下，声明一个名为 Tool 的结构。

```
struct Tool
{

}
```

此时 NTest 命名空间的内部结构如代码清单 3-3 所示。

代码清单 3-3　NTest 命名空间的完整代码

```
namespace NTest
{
    namespace NSub1
    {
        class WorkTask
        {

        }
    }

    namespace NSub2
```

```
    {
        struct Tool
        {

        }
    }
}
```

步骤 6：回到 Program 类的 Main 方法，在方法体中分别使用刚才定义的两个类型来声明变量。

```
static void Main(string[] args)
{
    NTest.NSub1.WorkTask v1 = null;
    NTest.NSub2.Tool v2;
}
```

引用类型时，加上其所在的命名空间名字，每一层命名空间用半角句点分隔。

注意：嵌套命名空间主要是以“.”运算符分隔。在实际编写代码时，并不一定要求命名空间之间有嵌套格式，例如本实例中的代码结构也可以写为：

```
namespace NTest.NSub1
{
    class WorkTask
    {

    }
}
namespace NTest.NSub2
{
    struct Tool
    {

    }
}
```

实例 31　引入命名空间

【导语】

使用 using 指令可以在代码中引入命名空间。引入命名空间后，在代码中访问某个类型时就不必敲入命名空间的名字，使代码更简洁，可读性更高。

using 指令可以在以下两处使用：

(1) 代码文件顶部，在所有代码之前。此处所引入的命名空间，可以在整个代码文件中使用。不管当前代码文件中有多少个命名空间，有多少个类型，均可使用。

(2) 在某个命名空间内。此处只在当前命名空间内有效,在当前命名空间以外不可用。

本实例将以 System.Collections 命名空间下的类型进行演示。

【操作流程】

步骤 1:新建一个控制台应用程序项目。

步骤 2:此时会自动打开 Program.cs 文件。文档顶部默认已经引入了 System 命名空间,在第一行 using 指令后面,引入 System.Collections 命名空间。

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

步骤 3:在 Main 方法中实例化一个 ArrayList 对象,并向其中添加三个字符串实例。

```
ArrayList mylist = new ArrayList();
mylist.Add("Tom");
mylist.Add("Jim");
mylist.Add("Jack");
```

ArrayList 是一个容量可自动增长的数组类型,可以向其添加任意类型的元素。

如果没有使用 using 指令引入 System.Collections 命名空间,那么在访问 ArrayList 类的时候就必须写上完整的命名空间,例如:

```
System.Collections.ArrayList mylist = new System.Collections.ArrayList();
```

这样代码会变得冗长,而且阅读起来也不方便。尤其是在一个代码文件中多处使用同一个命名空间时,通过 using 指令在文档的顶部引入后,不必在代码中重复输入命名空间。

步骤 4:输出 ArrayList 对象中所有元素。

```
foreach(object o in mylist)
{
    Console.WriteLine(o);
}
```

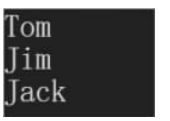

图 3-1 输出 ArrayList 实例中的元素

步骤 5:运行应用程序,输出结果如图 3-1 所示。

实例 32 在命名空间内部引入其他命名空间

【导语】

using 指令既可以在代码文档的顶部使用,也可以在某个命名空间内部使用,此时要注意所引入的命名空间只在当前命名空间中有效。

【操作流程】

步骤 1:新建一个控制台应用程序项目,并命名为 Demo。

步骤 2:在生成的 Program.cs 文件中,会创建默认的 Demo 命名空间。请读者手动把 Demo 命名空间外部的 using 指令代码(模板默认生成)删除。

步骤 3:在 Demo 命名空间内部加入以下 using 指令。

```
namespace Demo
{
    using System;
    using System.Collections.Generic;
...
}
```

步骤 4：实例化一个 List < int >对象，并添加 4 个元素。

```
List<int> list = new List<int>
{
    100,200,300
};
```

如果不引入 System.Collections.Generic 命名空间，那么在访问 List < T >类时就要写上完整的命名空间。

```
System.Collections.Generic.List<int> list = new System.Collections.Generic.List<int>
{
    100,200,300
};
```

注意：由于 System.Collections.Generic 命名空间是在 Demo 命名空间中引入的，所以在 Demo 命名空间以外的代码要访问 List < T >类就必须使用 System.Collections.Generic.List < T >，而不能直接使用 List < T >。

实例 33　使用全局命名空间

【导语】

应用程序项目隐藏着一个根命名空间，即全局命名空间。全局命名空间可以包含项目中的所有类型的访问范围，包括项目中所引用的其他组件。

由于根命名空间是隐式存在的，它没有明确的名称，所以访问它就必须使用 global 关键字。该关键字一般用来解决类型与命名空间的命名冲突问题，在以下实例中进行演示。

【操作流程】

步骤 1：新建控制台应用程序项目。

步骤 2：在 Program 类中声明一个嵌套的类，命名为 System。

```
class Program
{
    public class System { }
    static void Main(string[] args)
    {

    }
}
```

步骤 3：在 Main 方法中尝试实例化 System. Version 类。

```
System.Version v = new System.Version();
```

此时会发生错误，因为此处被识别为上面定义的 System 类，而且 System 类中没有嵌套的 Version 类，即编译器把 System. Version 识别为 System 类的嵌套类 Version。

步骤 4：如果要使用 System 命名空间下的 Version 类，就必须显式加上 global 关键字，通过全局命名空间强制指向 System 命名空间下的 Version 类。

```
global::System.Version v = new global::System.Version();
```

此时，编译器就能正确识别 System. Version 类。

实例 34 为引入的命名空间设置别名

【导语】

尽管命名空间在类型声明阶段解决了命名冲突的问题，然而该冲突在引入命名空间后依然会出现。例如，命名空间 A 下面有一个 Product 类，完整名称为 A. Product；命名空间 B 下面也有一个 Product 类，完整名称为 B. Product。如果将 A、B 两个命名空间同时引入，那么在代码中直接访问 Product 类会发生歧义，即编译器无法判断使用了哪个命名空间下的 Product 类。

如果命名空间的名称比较短（如上面举例中的 A、B），则访问类型时可以把命名空间写全，即在代码中使用 A. Product 或 B. Product；但是如果命名空间的名称很长，在代码中访问类型会显得冗长。例如把上面举例中的 A 命名空间改为 Company. Parts. WorkItems，把 B 命名空间改为 Company. Parts. CheckedItems，那么访问 Product 类时就要写上 Company. Parts. WorkItems. Product 或 Company. Parts. CheckedItems. Product。很明显，这样写出来的代码并不简洁。

要解决这个问题，可以在引入命名空间时分配一个别名。例如为上面的 Company. Parts. WorkItems 分配一个别名 W，这样访问 Product 类时就可以写上 W. Product。

【操作流程】

步骤 1：新建一个控制台应用程序项目。

步骤 2：在 Program. cs 文件中定义两个命名空间。

```
namespace Organization.Component.Extensions
{

}
namespace Organization.Component.MainParts
{

}
```

步骤 3：在以上两个命名空间内，分别声明一个 BackgroundWork 类。

```
namespace Organization.Component.Extensions
{
    public class BackgroundWork { }
}
namespace Organization.Component.MainParts
{
    public class BackgroundWork { }
}
```

步骤 4：在代码文件顶部使用 using 指令引入上面定义的两个命名空间，并为它们分配一个简短的别名。

```
using ext = Organization.Component.Extensions;
using mps = Organization.Component.MainParts;
```

步骤 5：在代码中访问 BackgroundWork 类时，就可以加上命名空间的别名。虽然多了个前缀，但由于别名比较简短，代码看起来依然很简洁。

```
ext.BackgroundWork bw1 = new ext.BackgroundWork();
mps.BackgroundWork bw2 = new mps.BackgroundWork();
```

实例 35　使用 using static 指令

【导语】

使用 using static 指令，可以像引入命名空间那样引入某个类型（该类型是静态类或者包含静态成员）。引入类型后，在代码中访问其静态成员时可以省略类型名称。

本例以 System 命名空间下比较有代表性的两个类来做演示。

【操作流程】

步骤 1：新建控制台应用程序项目。

步骤 2：在文件的顶部，使用 using static 指令引入 Console 和 Math 两个类。

```
using static System.Console;
using static System.Math;
```

步骤 3：此时访问 Console.WriteLine 方法可以不写 Console 类的名称。

```
WriteLine("Hello World!");
```

步骤 4：同样，访问 Math 类也不用写类名 Math。

```
WriteLine( $"5 的平方为:{Pow(5d, 2d)}");
WriteLine( $" - 650 的绝对值是:{Abs( - 650)}");
WriteLine( $"16,33 中最小的数是:{Min(16, 33)}");
```

其中 Abs、Pow 以及 Min 都是 Math 类公开的静态方法，如果不使用 using static 指令，

则上面三行代码就应写成

```
Console.WriteLine( $"5 的平方为:{Math.Pow(5d, 2d)}");
Console.WriteLine( $" - 650 的绝对值是:{Math.Abs( - 650)}");
Console.WriteLine( $"16,33 中最小的数是:{Math.Min(16, 33)}");
```

显然,使用了 using static 指令后,代码可以更简练。

3.2　变量与常量

实例 36　一次性声明多个变量

【导语】

变量的声明语法如下。

```
<类型> <变量名>
```

类型名称与变量名称之间要有空格。对于同一类型的多个变量,可以逐个声明,例如

```
int x;
int y;
int z;
```

其实,可以一次性声明多个类型相同的变量,即按照以下格式在一行代码中同时声明。

```
int x, y, z;
```

【操作流程】

步骤 1:新建控制台应用程序项目。

步骤 2:此时默认会打开 Program.cs 文件。

步骤 3:在 Main 方法中声明三个 string 类型的变量。

```
string a;
string b;
string c;
```

步骤 4:可以在一行代码中同时声明这三个变量(变量之间用半角逗号分隔)。

```
string a, b, c;
```

步骤 5:也可以在声明变量时进行赋值。

```
string d = null, e = "", f = "food";
```

实例 37　让编译器自动推断变量的类型

【导语】

在声明变量时,可以使用 var 关键字来描述类型,编译器会根据代码给变量的赋值来推

断其类型。因此,在使用 var 关键字声明变量后要马上给变量赋值,否则编译器无法推断变量的类型。

【操作流程】

步骤 1:新建一个控制台应用程序项目。

步骤 2:在 Main 方法中使用 var 关键字声明一个变量 abc,并为其赋值。

```
var abc = 3.141d;
```

数值 3.141 带有后缀 d,表示该数值为 double 类型(双精度浮点数),因此编译器推断出变量 abc 的类型为 System.Double。

步骤 3:可以使用以下代码来输出变量 abc 的类型。

```
Console.WriteLine( $ "变量 abc 的类型为:{abc.GetType().FullName}");
```

GetType 方法返回一个 Type 对象,该对象描述与类型有关的详细信息。应用程序运行之后,将输出以下文本。

```
变量 abc 的类型为:System.Double
```

注意:使用 var 关键字声明的变量必须初始化,即声明后必须马上赋值。因为编译器需要通过分配给变量的值来推断其类型,如果没有赋值,编译器就不知道变量是什么类型了。

实例 38 使用常量

【导语】

常量与变量相对,变量的"变"意味着在声明并初始化之后,可以在后续的代码中修改变量的值;而常量的"常"意味着一旦初始化之后,后续代码无法修改常量的值。

为了直观地看出变量与常量的区别,本实例将同时使用变量与常量。

【操作流程】

步骤 1:新建控制台应用程序项目。

步骤 2:声明变量 x 并初始化,然后在屏幕上输出一次。

```
int x = 10;
Console.WriteLine( $ "修改前,变量 x 的值:{x}");
```

步骤 3:把变量 x 的值改为 100,再输出一次。

```
x = 100;
Console.WriteLine( $ "修改后,变量 x 的值:{x}");
```

变量 x 在初始化时设定为 10,然后代码把它的值修改为 100,此时运行代码屏幕上会输出以下文本。

```
修改前,变量 x 的值:10
修改后,变量 x 的值:100
```

步骤 4：声明常量 Z,必须在声明后立即初始化。

```
const int Z = 500;
```

声明常量的方法与变量类似,只是要加上 const 关键字。

步骤 5：以下代码试图修改常量 Z 的值。

```
Z = 700;                    // 此行代码会报错
```

常量的值一旦初始化,是不能被修改的,所以上面这行代码会发生编译错误。

注意：按照习惯,常量的名称使用的是全大写的字母。但这只是习惯,并不是语法要求。

实例 39　获取变量的内存地址

【导语】

在 C++语言中,通过指针变量或者引用运算符(&),可以获得变量的内存地址。在 C#中,尽管有一些限制,仍然可以进行类似的处理。

指针操作在 C#语言中被认为是不安全的,因此,如果要使用指针变量,则必须将相关的代码写在 unsafe 代码块中,或者在方法的声明中加入 unsafe 修饰符。

例如,以下代码在 unsafe 代码块中声明指针类型的变量。

```
byte b = 255;
unsafe
{
    byte* pb = &b;
}
```

以上代码在方法声明中加入 unsafe 修饰符,表示该方法内部的代码中会出现不安全代码。

```
unsafe void DoSomething()
{
    float f = 0.0077f;
    float* pf = &f;
}
```

【操作流程】

步骤 1：新建一个控制台应用程序项目。

步骤 2：打开“解决方案资源管理器”窗口,右击项目名称,从菜单中选择“属性”命令。

步骤 3：此时会打开项目属性窗口,然后切换到“生成”选项卡。

步骤 4：在“常规”分组下,勾选“允许不安全代码”,如图 3-2 所示。

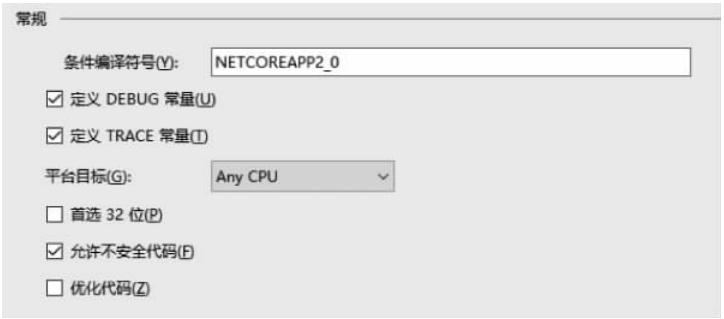

图 3-2 允许使用不安全代码

步骤 5：回到 Program.cs 文件，在 Main 方法上加上 unsafe 修饰符。

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

步骤 6：声明一个 int 类型的变量并初始化。

```
int val = 200;
```

步骤 7：声明一个指向 int 类型的指针，并且引用变量的地址。

```
int* p = &val;
```

步骤 8：为了能够获取并输出指针变量所包含的地址，还需要将其转换为 IntPtr 类型。

```
IntPtr ptr = (IntPtr)p;
Console.WriteLine($"变量的地址:{ptr.ToString("x")}");
```

指针类型并非从 Object 类派生，所以它没有 ToString 方法。要想在代码中输出指针指向的内存地址，需要将其转换为 IntPtr 类型。

实例 40 输出变量的名称

【导语】

在应用程序中输出变量的名称，实际就是获得变量名称的字符串表示形式，这需要用到 nameof 运算符。

【操作流程】

步骤 1：新建一个控制台应用程序项目。

步骤 2：在打开的 Program.cs 文件中找到 Main 方法，在方法体内部声明四个变量，并进行初始化。

```
string strvar = "hello";
int intvar = 3600;
```

```
var singlevar = 7.115f;
var longvar = 6560000L;
```

在代码中使用不带任何后缀的数值表示整型数值(32 位整数),带后缀 f 的表示单精度浮点数值,带 L 后缀的数值为长整型数值(64 位整数)。

步骤 3:将变量的名称与实值输出到屏幕。

```
Console.WriteLine( $ "变量 {nameof(strvar)} 的值为 {strvar}。");
Console.WriteLine( $ "变量 {nameof(intvar)} 的值为 {intvar}。");
Console.WriteLine( $ "变量 {nameof(singlevar)} 的值为 {singlevar}。");
Console.WriteLine( $ "变量 {nameof(longvar)} 的值为 {longvar}。");
```

步骤 4:按 F5 快捷键运行应用程序,屏幕输出如图 3-3 所示。

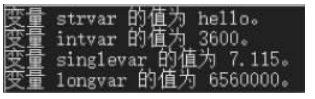

图 3-3 输出变量的名称

注意:nameof 运算符不仅可以获取变量/常量的名称,它还可以用于代码文档中的任何对象,例如可以获取命名空间名、类型名、类型成员名等。nameof 运算符能返回对象名称的字符串表示形式,一般可用于向用户输出变量名,或者某些需要以字符串形式提供对象名称的情况,例如通过反射技术动态查找类型的特定成员。

实例 41 为变量分配默认值

【导语】

在声明变量时可以为其分配一个初始值,也可以使用类型的默认值。例如,int 类型的默认值为 0,类(class)的默认值是 null。

要获取某个类型的默认值,建议使用 default 关键字,该关键字能自动返回指定类型的默认值。

【操作流程】

步骤 1:新建一个控制台应用程序项目。

步骤 2:声明一个 int 类型和一个 string 类型的变量,分别用 default 关键字分配默认值。

```
int v = default(int);
string s = default(string);
```

步骤 3:在屏幕上输出两个变量的值。

```
Console.WriteLine( $ "int 类型的默认值:{v}");
Console.WriteLine( $ "string 类型的默认值:{s ?? "null"}");
```

由于 string 类型的默认值是 null，但是 null 在屏幕中无法输出，所以这里用了一个?? 运算符，其意思是：如果字符串变量不为 null，就输出字符串内容；如果字符串为 null，就输出字符串“null”。

步骤 4：default 关键字，还有更简洁的写法，就是不必指定类型。例如上面声明变量的代码可以改为

```
int v = default;
string s = default;
```

步骤 5：此时需要更改项目使用的 C# 语言版本，版本号不低于 7.1。打开“解决方案资源管理器”窗口，在项目名称上右击，从菜单中选择“属性”，打开项目属性窗口。

步骤 6：切换到“生成”选项页，在页面底部找到并单击“高级”按钮。

步骤 7：语言版本选择 7.1 或以上，或者选择“最新次要版本(最新)”，然后单击“确定”按钮，如图 3-4 所示。

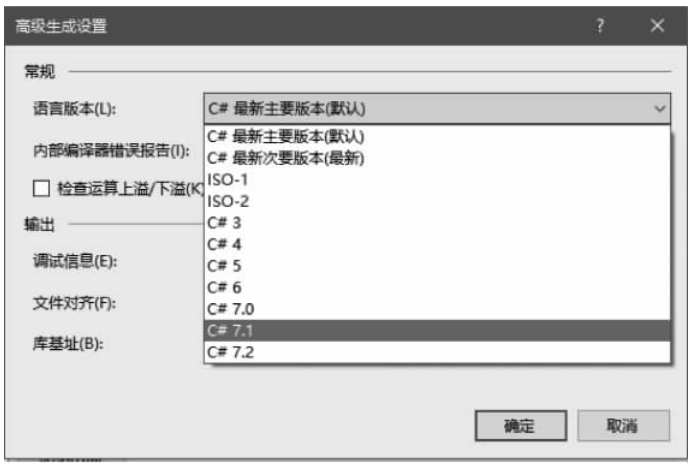

图 3-4　选择语言版本

注意：由于 default 关键字的这项增强功能是在 C# 7.1 中推出的，因此需要修改项目使用的语言版本。

3.3　程序入口点

实例 42　获取命令行参数

【导语】

程序入口点，即应用程序开始执行的位置，进入入口点后，代码会一直往下执行；当代

码退出入口点后，应用程序也会退出，整个应用程序生命周期结束。

程序入口点是一个静态方法，必须命名为 Main。Main 方法中一般会有一个字符串数组类型的参数，该参数用于接收传递给应用程序的命令行参数。

【操作流程】

步骤 1：新建控制台应用程序项目。

步骤 2：项目模板默认会生成 Program 类，并包含一个 Main 方法（即程序入口点）。

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

}
```

步骤 3：输入以下代码，在屏幕上输出命令行参数。

```
StringBuilder sbd = new StringBuilder();
sbd.AppendLine("接收到的命令行参数:");
foreach (string a in args)
{
    sbd.AppendFormat(" {0}", a);
}
Console.WriteLine(sbd);
```

应用程序接收到的命令行参数会传递给 Main 方法的参数 args，数组中每个元素都是一个参数。

步骤 4：要在调试时传递命令行参数，需要打开“解决方案资源管理器”窗口，然后右击项目名称，从菜单中执行“属性”命令，打开项目属性窗口。

步骤 5：在项目属性窗口中切换到“调试”选项卡。

步骤 6：在“应用程序参数”右侧的文本框中输入三个测试参数。

```
-a -b -c
```

参数之间用空格隔开，如图 3-5 所示。

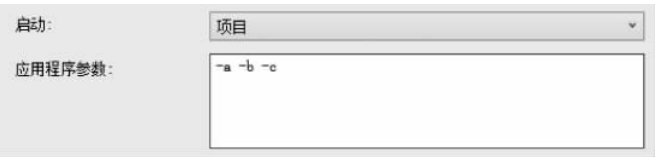

图 3-5　输入用于测试的参数

步骤 7：按 F5 快捷键运行应用程序，在控制台窗口中就能看到传递的命令行参数了，如图 3-6 所示。

步骤 8：如果使用 dotnet 命令直接执行应用程序，可以把需要传递的参数附加在.dll 文件名后面。

```
dotnet Demo.dll - hello - world
```

其中 Demo.dll 是项目编译后生成的程序文件。-hello 和-world 是命令行参数。

步骤 9：执行上述命令后，会输出如图 3-7 所示的内容。

图 3-6 输出的命令行参数

图 3-7 使用命令行执行程序

实例 43 处理多个入口点

【导语】

一个应用程序只能有一个入口点，但是在程序代码中是可以定义多个 Main 方法的。如果应用程序项目中包含多个 Main 方法，只能从中选择一个作为程序的入口点。

【操作流程】

步骤 1：新建控制台应用程序项目。

步骤 2：项目模板会生成一个 Program 类以及 Main 方法。

步骤 3：另定义一个 Test 类，并在其中也声明一个 Main 方法。

```
class Test
{
    static void Main(string[] args)
    {
        Console.WriteLine("第二个入口点。");
        Console.Read();
    }
}
```

注意：Main 方法必须是静态的(static)，但不要求是公共的(public)。

步骤 4：由于项目中包含了两个 Main 方法，此时运行项目会出现以下错误。

```
错误  CS0017  程序定义了多个入口点。使用 /main (指定包含入口点的类型)进行编译。
```

步骤 5：打开项目属性窗口，切换到“应用程序”选项卡，在“启动对象”下拉列表框中选择一个包含 Main 方法的类，如图 3-8 所示。此时应用程序就能正常运行了。

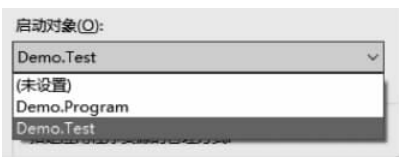

图 3-8 选择一个入口点

3.4　流程控制

实例 44　奇数还是偶数

【导语】

if 语句能够对代码的执行进行分支处理，其用法如下：

```
if ( <条件> )
{
     代码段 1
}
else
{
     代码段 2
}
```

其中“<条件>”是一个表达式，其结果为布尔类型（真或假）。如果“<条件>”成立（为真），就执行“代码段 1”，否则（为假）就执行“代码段 2”。

如果代码逻辑只有一个分支，else 子句可以省略，即：

```
if ( <条件> )
{
    代码段 1
}
```

当“<条件>”成立时，“代码段 1”被执行；如果条件不成立，直接跳过“代码段 1”。

执行本实例时由用户输入一个数值，然后程序判断该数值是奇数还是偶数，最后将结果输出到屏幕上，如图 3-9 所示。

图 3-9　判断数值的奇偶性

【操作流程】

步骤 1：新建一个控制台应用程序项目。

步骤 2：调用 Console 类的 ReadLine 方法读取用户从键盘输入的内容。

```
string input = Console.ReadLine();
```

其中，ReadLine 方法返回一个字符串实例，它表示用户输入的文本，用户可以一次性输入多个字符，并按 Enter 键确认。

步骤 3：得到用户输入的内容后，还需要对内容的有效性进行验证。因为用户有可能输入了非数字字符（例如输入了字母），而且本实例要求是大于 0 的整数，例如：

```
if(uint.TryParse(input, out uint number) && number > 0)
{
    …
}
```

```
else
{
    Console.WriteLine("你输入的内容无效。");
}
```

其中，TryParse 方法能够对字符串进行分析，验证其能不能转换为 uint 值(无符号整数)，如果能转换，就把转换后得到的值保存到变量 number 中，并且 TryParse 返回 true；如果无法转换，方法会返回 false。

此时 if 语句的判断条件由两个因素组成。首先，输入的文本必须是有效的整数；其次，该数值必须是大于 0 的。两个表达式用运算符 && 连接，表示只有当两个表达式同时成立时，if 语句的判断条件才会成立；如果其中有一个不成立，那么整个判断条件也不成立。

步骤 4：确保用户输入的数值有效后，就可以分析其奇偶性了。

```
if(uint.TryParse(input, out uint number) && number > 0)
{
    // 判断整数的奇偶性
    if((number % 2) == 0)
    {
        Console.WriteLine( $ "你输入的 {number} 是偶数。");
    }
    else
    {
        Console.WriteLine( $ "你输入的 {number} 是奇数。");
    }
}
else
{
    Console.WriteLine("你输入的内容无效。");
}
```

奇偶性的判断依据为：数值是否能被 2 整除。这里的处理方法是让 number 变量的值除以 2 并取其余数，如果余数为 0 说明可被 2 整除，即为偶数，否则是奇数。运算符%用于获取两个数相除后的余数。

注意：本实例使用了嵌套的 if 语句，即在 if 语句的代码块中又包含一层 if 语句，在一些复杂的逻辑处理中是允许使用多层嵌套的 if 语句的。

实例 45　使用 for 循环输出文本

【导语】

for 循环的用法如下：

```
for ( <变量初始值> ; <循环条件>; <修改变量> )
{
   代码片段
}
```

首先通过“<变量初始值>”表达式对变量(一般是整数值的类型,如 int、long 等)进行初始化(设定初值),然后启动循环,“代码片段”处的代码会被执行。执行完“代码片段”后会执行“<修改变量>”表达式,对变量的值进行修改(通常是加上 1),接着使用“<循环条件>”判断是否再需要进行循环。如果条件依然成立,则“代码片段”处的代码又会执行;如果“<循环条件>”不成立,就会跳出 for 循环。不断地循环往返,直到跳出 for 语句块。

【操作流程】

步骤 1:新建控制台应用程序项目。

步骤 2:运用 for 循环,在屏幕上输出 5 行文本:

```
for (int n = 1; n <= 5; n++)
{
    Console.WriteLine( $ "这是第 {n} 行文本。");
}
```

其中变量 n 初始化为 1,循环的条件是 n 要小于或等于 5,每次循环过后都会让 n 加上 1。例如,第一轮循环,n 的值为 1,输出“这是第 1 行文本”,然后回到 for 语句块起点,将 n 的值加 1,变为 2,2 小于 5,因此条件成立,继续循环,输出“这是第 2 行文本”;依此类推,直到 n 的值等于 6 时,循环条件不再成立,就会退出循环。运行效果如图 3-10 所示。

图 3-10　循环输出的文本

实例 46　生成由字符组成的图案

【导语】

本实例是在前面的实例基础上增强的。首先,通过循环产生以下图案:

```
*
**
***
****
*****
******
*******
********
```

把上面的图案中每一行进行反转,即

```
*            →            *
**           →           **
***          →          ***
****         →         ****
*****        →        *****
******       →       ******
*******      →      *******
********     →     ********
```

接着反转所有行，得到

```
******************
*******    *******
******      ******
*****        *****
****          ****
***            ***
**              **
*                *
```

最后把所有行再做一次反转，并拼接到上面各行后面，就能得到如下最终图案。

```
*****************
*******  *******
******     ******
*****       *****
****         ****
***           ***
**             **
*               *
*               *
**             **
***           ***
****         ****
*****       *****
******     ******
*******  *******
*****************
```

控制台最终输出效果如图 3-11 所示。

图 3-11　程序最后输出的图案

【操作流程】

步骤 1：新建控制台应用程序项目。

步骤 2：通过循环，产生图案的上半部分。

```
List<string> list1 = new List<string>();
for (int x = 1; x <= 8; x++)
{
    String s1 = "";
    int v = 0;
    while (v < x)
    {
        s1 += "*";
        v++;
    }
    // 其余的字符用空格补齐，使字符串总长度为 8
    s1 = s1.PadRight(8);
```

```
    // 将整行字符进行反转,并产生新的字符串
    string s2 = new string(s1.Reverse().ToArray());
    // 将两个字符串进行拼接
    list1.Add(s1 + s2);
}
```

步骤 3：将图案中的所有行反转再输出。

```
// 第一轮输出
list1.Reverse();
foreach (var item in list1)
{
    Console.WriteLine(item);
}
```

步骤 4：将图案中的所有行再一次反转,继续输出。

```
// 第二轮输出
list1.Reverse();
foreach (var item in list1)
{
    Console.WriteLine(item);
}
```

实例 47 死循环的处理方法

【导语】

所谓死循环,就是永不休止的循环。循环体内部的代码会永久性地执行,产生的原因在于循环条件永远成立,使得循环体无法退出。在实际编程中,要避免出现死循环,一旦遇到死循环,后续的代码将无法被执行。

如果出于代码逻辑考虑,确实要设定永久成立的循环条件(例如,3<5,因为 3 确实小于 5,这样的条件永久成立),那么也要在适合的时候从循环体内部退出循环。要在循环体内部退出循环,可以使用 break 语句,例如:

```
while ( 3<5 )
{
    if ( … )
{
        break;
}
}
```

【操作流程】

步骤 1：新建一个控制台应用程序项目。

步骤 2：在 Main 方法中设置一个死循环。

```
while (9 == 9)
{
    Console.WriteLine($"{DateTime.Now:T} 正在执行循环……");
}
```

由于 9 等于 9 是永远成立的，所以上面的 while 循环代码会永远地执行，除非强制退出应用程序。为了能更好地看到 Console.WriteLine 方法被无限次调用，上面代码中刻意在输出的文本前面加上当前时间。

步骤 3：按 F5 快捷键执行程序，会看到如图 3-12 所示的输出。

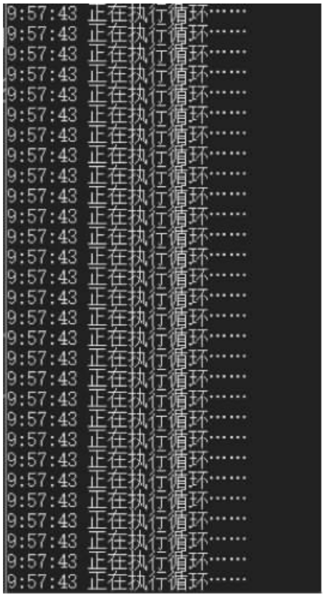

图 3-12 应用程序在执行死循环

步骤 4：要让循环终止，此时只能强制关闭应用程序。

实例 48 退出循环的方法

【导语】

前面提到过，使用 break 语句可以退出循环，在死循环内部更需要这样做。本实例将实现：在死循环执行过程中，只要用户按下 E 键，就可以退出循环。

【操作流程】

步骤 1：新建控制台应用程序项目。

步骤 2：在 Main 方法内设定一个死循环。

```
while (true)
{
    …
}
```

循环条件始终是 true，表明这个条件是永远成立的，因此这是一个死循环。

步骤 3：在死循环内部，加退出循环的条件。

```
while (true)
{
    Console.WriteLine("请按 E 键退出。");
    if(Console.ReadKey(true).Key == ConsoleKey.E)
    {
        break;
    }
}
```

当用户按下 E 键后，使用 break 语句退出循环。

实例 49 输出 20 以内能被 3 整除的正整数

【导语】

break 语句会直接退出整个循环；而 continue 语句则会跳过本轮循环，重新回到循环代码的顶部执行下一轮循环，循环并不会退出。

本实例将执行一个从 1 到 20 的循环，如果当前数值能被 3 整除就将其输出，否则就跳过本次循环。例如，当前数值为 5，不能被 3 整除，则直接跳过，不再往下执行，而是继续下一轮循环；然后当前数值变为 6，可以被 3 整除，于是就输出数值 6。

【操作流程】

步骤 1：新建控制台应用程序项目。

步骤 2：使用 for 循环对 1 到 20 的正整数进行试验，找出能被 3 整除的数。

```
for(int i = 1; i <= 20; i++)
{
    if ((i % 3) != 0)
        continue;
    Console.Write(" {0}", i);
}
```

当数值不能被 3 整除（即除以 3 后余数不为 0）时，执行 continue 语句跳过本次循环；如果可以被 3 整除，就输出该数值。

步骤 3：按 F5 快捷键运行程序，输出结果如图 3-13 所示。

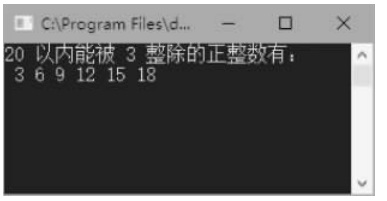

图 3-13 20 以内可被 3 整除的正整数

实例 50 做一道选择题

【导语】

switch 语句首先提取指定表达式的值，然后将该值与各个 case 开关所表示的分支进行匹配，如果与其中一个分支的值相等，那么就执行该 case 开关下的代码。

switch 语法如下：

```
switch ( <匹配表达式> )
{
    case n1:
        …
```

```
        break;
    case n2:
        …
        break;
    …
    default:
        …
        break;
}
```

每个分支后都要加上 break、goto、return 等语句，是为了避免代码把后面的分支都执行，如果某个分支匹配后就继续执行后面的分支，就损坏分支语句的逻辑结构了。default 分支是可选的，如果上面各个 case 都不匹配，则执行 default 分支的代码。

本实例将模拟一道选择题，用户通过输入选择进行作答，代码会对用户选择进行分析，并给出被选项的说明。

【操作流程】

步骤 1：新建一个控制台应用程序项目。

步骤 2：设定一道有 4 个选项的选择题。

```
Console.WriteLine("请问,以下哪位历史人物生活在唐朝?");
Console.WriteLine("1、蔡邕");
Console.WriteLine("2、唐寅");
Console.WriteLine("3、王勃");
Console.WriteLine("4、苏轼");
```

步骤 3：读取用户输入。

```
string input = Console.ReadLine();
```

ReadLine 方法读取整行文本并返回，按下 Enter 键进行确认。

步骤 4：对用户输入的选项字符串进行分析。

```
switch (input)
{
    case "1":
        Console.WriteLine("蔡邕生活在东汉时期。");
        break;
    case "2":
        Console.WriteLine("唐寅是明朝人。");
        break;
    case "3":
        Console.WriteLine("恭喜你,答对了。");
        break;
    case "4":
        Console.WriteLine("苏轼生活在北宋时期。");
        break;
```

```
        default:
            Console.WriteLine("你未做出有效选择。");
            break;
    }
```

如果用户选择“3”，就会输出“恭喜你，答对了”；如果用户选择“1”，并非正确答案，因而告诉用户“蔡邕生活在东汉时期”。

步骤 5：按下 F5 快捷键，运行应用程序，输入一个选项，然后按 Enter 键确认，如图 3-14 所示。

图 3-14　选择一个答案

实例 51　switch 语句的类型匹配

【导语】

switch 语句中的 case 开关除了可以匹配常量外，还可以匹配类型。当测试表达式的类型能够与某个 case 子语句所指定的类型相匹配时，就会执行该 case 分支的代码。代码清单 3-4 演示了类型匹配的简单用法。

代码清单 3-4　switch 语句类型匹配示例

```
object vx = "abcde";
switch (vx)
{
    case int n:
        // 这是一个整数值
        break;
    case string t:
        // 这是字符串
        break;
    default:
        // 未知类型
        break;
}
```

变量声明为 object 类型，而实际赋值时使用了 string 类型的值。第一个 case 子语句需要 int 类型的值，类型不匹配，因此此分支不会执行；第二个 case 子语句需要的是 string 类型的值，类型匹配，所以该分支会被执行。

在使用类型匹配时，一定要注意类型的兼容性问题，例如以下有 A、B、C 三个类，其中，B 类从 A 类派生，C 类从 B 类派生。

```
class A { }
class B : A { }
class C : B { }
```

然后在 switch 语句中使用类型匹配。

```
object o = new B();
switch (o)
{
    case A x:
        // A、B、C 类型的实例均匹配
        break;
    case B x:
        // 只有 B、C 类型的实例匹配
        break;
    case C x:
        // 只有 C 类型的实例匹配
        break;
}
```

这段代码无法通过编译，而且在 Visual Studio 的代码编辑器中也会提示错误，错误的原因是第二个与第三个 case 分支是永远不会被执行的。第一个 case 子语句匹配类型为 A，即无论测试表达式中的实例是 A 类型，B 类型还是 C 类型，都能够与该 case 分支匹配，这是因为 A、B、C 三种类型的实例都能赋值给 A 类型的变量。

要解决该错误，最简单的方法就是调换各 case 子句的顺序，即把代码改为：

```
switch (o)
{
    case C x:
        // A、B、C 类型的实例均匹配
        break;
    case B x:
        // 只有 B、C 类型的实例匹配
        break;
    case A x:
        // 只有 C 类型的实例匹配
        break;
}
```

修改后，如果测试表达式是 C 类型的实例，那么它只能匹配第一个 case 分支；同理，如果实例是 B 类型，它无法赋值给 C 类型的变量，所以只能匹配第二个 case 分支；如果实例是 A 类型，它无法赋值给 B 和 C 的变量，只能匹配第三个 case 分支了。

【操作流程】

步骤 1：新建一个控制台应用程序项目。

步骤 2：声明一个 object 类型的变量，并赋一个 double 类型的数值。

```
object obj = 0.0001d;
```

步骤 3：使用 switch 语句进行类型匹配。

```
switch (obj)
```

```
{
    case int v:
        Console.WriteLine( $ "{v} 是一个 int 值。");
        break;
    case decimal v:
        Console.WriteLine( $ "{v} 是一个 decimal 值。");
        break;
    case double v:
        Console.WriteLine( $ "{v} 是一个 double 值。");
        break;
    default:
        Console.WriteLine("未知类型。");
        break;
}
```

上述代码中，只有第三个 case 子语句能够匹配，因为 obj 变量的实际值为 double 类型。屏幕输出如图 3-15 所示。

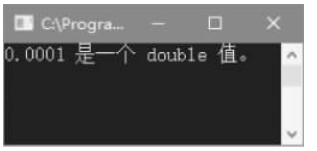

图 3-15　匹配 double 类型表达式

实例 52　在 case 语句中使用 when 子句

【导语】

case 语句后除了使用常量值外，还可以进行类型匹配。为了让类型匹配更加精确，可以在 case 语句后加上 when 子语句。when 子语句所使用的表达式必须返回布尔值，只有当 when 子语句返回 true 时，该 case 语句才会被执行。

所以，when 子语句就相当于给 case 分支增加一个额外的条件，以进行更细致的筛选。例如，通过类型匹配可以匹配出一个数组对象实例，可是这个数组有可能是空数组（元素个数为 0），在 case 分支处理时，开发者可能会考虑当出现空数组时做另外处理，此时，when 子语句就发挥作用了。以下代码在 switch 语句块中设定两个 case 分支，这两个分支都接受数组类型的对象，只是其中一个明确接受空数组。

```
case Array arr when arr.Length == 0:
    …
    break;
case Array arr:
    …
    break;
```

下面的实例将会进一步演示 when 子句的用法。

【操作流程】

步骤 1：新建控制台应用程序项目。

步骤 2：在生成的 Program 类中增加一个静态方法。该方法接收一个 object 类型的参数，并使用 switch 语句块进行分支处理，详见代码清单 3-5。

代码清单 3-5　DisplayInfo 方法

```
static void DisplayInfo(object instance)
{
    switch (instance)
    {
        case null:
            Console.WriteLine("对象未实例化。");
            break;
        case Array arr when arr.Length == 0:
            Console.WriteLine("这是个空数组。");
            break;
        case Array arr:
            Console.WriteLine( $ "数组包含 {arr.Length} 个元素。");
            break;
        case IList ls when ls.Count == 0:
            Console.WriteLine("这是个空列表。");
            break;
        case IList ls:
            Console.WriteLine( $ "列表总共有 {ls.Count} 项。");
            break;
    }
}
```

null 值不会匹配任何类型，所以要作为一个常量值来筛选。在使用 when 子语句时一定要把握好 case 的顺序。例如上面代码中对空数组的分析，假设把两个 case 语句做以下调换。

```
case Array arr:
    Console.WriteLine( $ "数组包含 {arr.Length} 个元素。");
    break;
case Array arr when arr.Length == 0:
    Console.WriteLine("这是个空数组。");
    break;
```

这样会发生编译错误。因为不论数组中是否包含元素，第一个 case 语句都能匹配，这样会使得第二个 case 语句永远无法匹配，等同于第二个 case 语句后的代码永远不会被执行。

步骤 3：分别向 DisplayInfo 方法传递不同的对象进行测试。

```
// 测试一：空引用
DisplayInfo(null);

// 测试二：空数组
int[] intarr = { };
DisplayInfo(intarr);
```

```
// 测试三:空列表
List<long> listet = new List<long>();
DisplayInfo(listet);

// 测试四:包含元素的数组
byte[] btarr = { 36, 2, 54, 7 };
DisplayInfo(btarr);

// 测试五:包含列表项的列表
List<int> listint = new List<int>{ 21, 13, 62, 8, 19 };
DisplayInfo(listint);
```

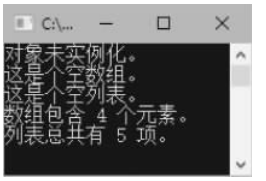

图 3-16　when 子句演示结果

步骤 4：按下 F5 快捷键运行项目，会看到如图 3-16 所示的输出。

实例 53　代码跳转

【导语】

在代码文档中，可以在某段代码前写上标签，然后在代码的其他位置使用 goto 关键字跳转到指定的标签处，并继续执行标签后的代码。

注意：此处所说的标签是在代码逻辑上定义的标签，而非在 Visual Studio 中为代码设置的标签。

【操作流程】

步骤 1：新建一个控制台应用程序项目。

步骤 2：在 Main 方法中定义五处标签，详见代码清单 3-6。

代码清单 3-6　在 Main 方法中定义五处标签

```
left:
Console.WriteLine("你按下了左方向键。");
Console.Read();
return;

right:
Console.WriteLine("你按下了右方向键。");
Console.Read();
return;

up:
Console.WriteLine("你按下了向上键。");
Console.Read();
```

```
return;

down:
Console.WriteLine("你按下了向下键。");
Console.Read();
return;

other:
Console.WriteLine("你按下了其他键。");
Console.Read();
return;
```

标签的定义方法是在标签名称后面紧跟一个英文的冒号。上面代码在每个标签后的代码中都加上了 Console.Read 方法的调用以及 return 语句，这是为了防止代码继续往下执行。例如，假设代码跳转到 left 标签处，那么 left 标签后面的所有代码都会被执行，包括下面 right、up 等标签后面的代码也会执行。因为代码跳转到某个标签后，就会从该标签处继续往下执行，直到程序退出或者没有可执行的代码。

步骤 3：调用 ReadKey 方法从键盘输入中读取一个键码，并判断哪个键被激活了。如果按下了 Left 键，就执行 left 标签后的代码；如果按下了 Right 键，就执行 right 标签后的代码；如果按下了 Up 键，就执行 up 标签后的代码；如果按下了 Down 键就执行 down 标签后的代码；如果按下了其他键，就执行 other 标签后的代码。

```
var keyinfo = Console.ReadKey(true);

if (keyinfo.Key == ConsoleKey.LeftArrow)
    goto left;
else if (keyinfo.Key == ConsoleKey.RightArrow)
    goto right;
else if (keyinfo.Key == ConsoleKey.UpArrow)
    goto up;
else if (keyinfo.Key == ConsoleKey.DownArrow)
    goto down;
else
    goto other;
```

在 goto 关键字后面直接写上要跳转的代码标签即可。

图 3-17 goto 语句测试

步骤 4：运行项目后，在键盘上按下一个键来测试，结果如图 3-17 所示。