# 参考文档 Cs参考手册

# C# 编程指南

http://shouce.jb51.net/net/index.html

本节提供有关关键的 C# 语言功能和 C# 可通过 .NET Framework 访问的功能的详细信息。

本节中大部分内容都假定您已了解有关 C# 和一般编程概念的一些知识。如果您刚开始学习编程或 C#，则应从 Visual C# 指导教程开始。您可能还想访问 [Visual C# 开发中心](http://msdn2.microsoft.com/zh-cn/vcsharp/default.aspx)，此处提供很多教程、示例和视频，可帮助您入门。

有关特定的关键字、运算符和预处理器指令的信息，请参见 [C# 参考](http://shouce.jb51.net/net/html/06de3167-c16c-4e1a-b3c5-c27841d4569a.htm)。有关 C# 语言规范的信息，请参见 [C# 语言规范](http://shouce.jb51.net/net/html/e5d5a5cc-636b-4bff-b9c8-a8edc6207c22.htm)。

## C#编译命令

@echo off

if exist .\test.exe del .\test.exe

rem ---0、直接产生test.exe

csc test.cs

rem ---1-1、产生test.dll类库

rem csc /target:library test.cs

rem ---1-2、产生test.exe应用程序

rem csc /reference:test.dll test.cs

test.exe

pause

## 深入 C# 程序

### Hello World -- 您的第一个程序

#### 以下控制台程序是传统“Hello World!”程序的 C# 版，该程序显示字符串 Hello World!。

// A Hello World! program in C#

**using** System;

**namespace** HelloWorld

{

**class** Hello

{

**static** **void** Main()

{

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

// Keep the console window open in debug mode.

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

}

### Main 方法

C# 程序必须包含一个 Main 方法，用于控制程序的开始和结束。在 Main 方法中创建对象和执行其他方法。

Main 方法是驻留在类或结构内的静态方法。在前面的“Hello World!”示例中，此方法驻留在一个名为 Hello 的类中。用下列方式之一声明 Main 方法：

#### 示例：

返回 void：

**static** **void** Main()

{

//...

}

它还可以返回 int：

**static** **int** Main()

{

//...

**return** 0;

}

由于有这两个返回类型，它可以带有参数：

**static** **void** Main(**string**[] args)

{

//...

}

或

**static** **int** Main(**string**[] args)

{

//...

**return** 0;

}

### 输入和输出

C# 程序通常使用 .NET Framework 的运行库提供的输入/输出服务。语句 System.Console.WriteLine("Hello World!"); 使用了 WriteLine 方法，该方法是运行库中的 Console 类的输出方法之一。它显示了标准输出流使用的字符串参数，输出流后面跟一个新行。其他 Console 方法用于不同的输入和输出操作。如果程序开始处包含 using System; 指令，则无需完全限定 System 类和方法即可直接使用它们。例如，您可以改为调用 Console.WriteLine，而不必指定 System.Console.Writeline：

### 编译和执行

可以通过在 Visual Studio IDE 中创建项目或使用命令行来编译“Hello World!”程序。使用 Visual Studio 命令提示符窗口或调用 vsvars32.bat 将 Visual C# 工具集放置在命令提示符下的路径中。

从命令行编译程序：

* 使用任何文本编辑器创建源文件，并将其保存为名如 Hello.cs 的文件。C# 源代码文件使用的扩展名是 .cs。
* 若要激活编译器，请输入命令：

csc Hello.cs

如果程序没有包含任何编译错误，则将创建一个 Hello.exe 文件。

* 若要运行程序，请输入命令：

Hello

## C# 程序的通用结构

C# 程序可由一个或多个文件组成。每个文件都可以包含零个或零个以上的命名空间。一个命名空间除了可包含其他命名空间外，还可包含类、结构、接口、枚举、委托等类型。以下是 C# 程序的主干，它包含所有这些元素。

// A skeleton of a C# program

using System;

using MyNamespace;

namespace MyNamespace

{

class MyClass

{

}

struct MyStruct

{

}

interface IMyInterface

{

}

delegate int MyDelegate();

enum MyEnum

{

}

namespace MyNestedNamespace

{

struct MyStruct

{

}

}

}

namespace zh.test

{

class Hello

{

static void Main()

{

MyClass y = new MyClass();

System.Console.WriteLine("C# 程序的通用结构示例!");

}

}

}

## Main() 和命令行参数

**Main** 方法是 C# 控制台应用程序或窗口应用程序的入口点。（库和服务不要求将 **Main** 方法作为入口点。）应用程序启动时，**Main** 方法是第一个调用的方法。

C# 程序中只能有一个入口点。如果您有多个类都包含 **Main** 方法，则必须使用 **/main** 编译器选项编译您的程序，以指定用作入口点的 **Main** 方法。

### 示例

**class** TestClass

{

**static** **void** Main(**string**[] args)

{

// Display the number of command line arguments:

System.Console.WriteLine(args.Length);

}

}

### 概述

* **Main** 方法是 .exe 程序的入口点，程序控制流在该处开始和结束。
* **Main** 在类或结构的内部声明。**Main** 必须是静态的，且不应该是公用的。（在前面的示例中，它接受默认访问级别 private。）但不要求封闭类或结构是静态的。
* **Main** 的返回类型有两种：**void** 或 **int**。
* 所声明的 **Main** 方法可以具有包含命令行实参的 string[] 形参，也可以不具有这样的形参。使用 Visual Studio 创建 Windows 窗体应用程序时，可以手动添加形参，也可以使用 Environment 类获取命令行实参。形参读取为从零开始编制索引的命令行实参。与 C 和 C++ 不同，在 C# 中不将程序名称视为第一个命令行实参。

# 类型

## 强制转换和类型转换

### 示例

using System;

namespace zh.test

{

class MyTest

{

static void Main()

{

int i1 = 123;

double d1 = i1;

double d2 = 12.34;

int i2 = (int)d2;

string s1 = Convert.ToString(d2);

string s2 = i1.ToString();

double d3 = double.Parse(s1);

int i3 = int.Parse("333");

int i4 = Convert.ToInt32("4444");

//int i3 = (int)s1;//无法将类型“string”转换为“int”

System.Console.WriteLine("i1={0}; d1={1};\r\nd2={2};i2={3};", i1, i1, d2, i2);

System.Console.WriteLine("s1={0};\r\ns2={1};\r\nd3={2};\r\ni3={3};\r\ni4={4};", s1, s2, d3, i3,i4);

try

{

i4 = Convert.ToInt32("abc");

}

catch (Exception e)

{

Console.WriteLine("输入值:{0}\r\n详细信息：{1}","abc",e);

}

}

}

}

## 装箱和取消装箱

### 示例

using System;

namespace zh.test

{

class IsTest

{

static void Main()

{

int i = 123;

object oi = i;// Implicit boxing

//object oi = (object)i;// explicit boxing

i = 456;

int j = (int)oi;//unboxing

System.Console.WriteLine("int i = 123;object oi = i;i = 456;int j = (int)oi;\r\n//i={0}; oi={1}; j={2};", i, oi, j);

}

}

}

## 使用 as 和 is 运算符

### 示例

using System;

namespace zh.test

{

class A

{

// public void Show() { Console.WriteLine("调用A.Show()成功！！"); }

public override string ToString()

{

return "成功调用A.ToString！！";

}

}

class B : A { }

class C : B { }

class D

{

public void Show()

{

System.Console.WriteLine("调用D.Show()成功！！");

}

}

class IsTest

{

static void Main()

{

A a = new A();

B b = new B();

C c = new C();

D d = new D();

Console.WriteLine("-----UseIsOperator----");

UseIsOperator(a);

UseIsOperator(b);

UseIsOperator(c);

UseIsOperator(d);

UseIsOperator("HHH");

System.Console.WriteLine("-----c as B----");

B bb = c as B;

if (bb != null)

System.Console.WriteLine(bb.ToString());

else

System.Console.WriteLine("{0} 不能转换为A ", c.GetType());

System.Console.WriteLine("-----c as D----");

D dd = (object)c as D;

if (dd != null)

dd.Show();

else

System.Console.WriteLine("{0} 不能转换为 D ", c.GetType());

System.Console.WriteLine("-----d as D----");

D ddd = d as D;

if (ddd != null)

ddd.Show();

else

System.Console.WriteLine("{0} 不能转换为 D ", d.GetType());

Console.WriteLine("-----string as string----");

string str = "astr";

string s = str as string;

if (s != null)

System.Console.WriteLine( s);

}

static void UseIsOperator(object o)

{

if (o is A)

System.Console.WriteLine("{0} 可以转换为 A", o.GetType());

else if (o is B)

System.Console.WriteLine("{0} 可以转换为 B", o.GetType());

else if (o is C)

System.Console.WriteLine("{0} 可以转换为 C", o.GetType());

else if (o is D)

System.Console.WriteLine("{0} 可以转换为 D", o.GetType());

else

System.Console.WriteLine("{0} 不能转换为以上任一类型 ", o.GetType());

}

static void UseAsOperator(object obj1, object obj2)

{

;

}

}

}

## using 关键字有两个主要用途：

### 引用命名空间、创建别名或导入命名空间中定义的类型。

using System;

using zh.TestUsing;

using nsUs=zh.TestUsing;

using clssUsing = zh.TestUsing.Using;

namespace zh.TestUsing

{

public class Using

{

public void Show(String s)

{

System.Console.WriteLine(s);

}

}

}

namespace zh.test

{

class MyTest

{

static void Main()

{

Using us1 = new Using();

us1.Show("using zh.TestUsing");

nsUs.Using us2 = new nsUs.Using();

us2.Show("using nsUs=zh.TestUsing;");

clssUsing us3 = new clssUsing();

us3.Show("using clssUsing = zh.TestUsing.Using;");

}

}

}

### 用于定义一个范围，在此范围的末尾将释放对象。

using System;

namespace zh.test

{

class MyTest

{

static void Main()

{

string sPath = System.Environment.CurrentDirectory + "\\ReadMe.txt";

using (System.IO.StreamReader sr = new System.IO.StreamReader(@sPath))

{

string s = null;

while ((s = sr.ReadLine()) != null)

{

Console.WriteLine(s);

}

}

}

}

}

## 泛型

//泛型

using System;

using System.Collections.Generic;

namespace Zh.Test.Generic

{

public class cGenericList<T>

{

void Add(T input) { }

}

public class cProgram

{

private class cExmaple { }

static void Main(string[] args)

{

cGenericList<int> IntList = new cGenericList<int>();

cGenericList<string> strList = new cGenericList<string>();

cGenericList<cExmaple> egList = new cGenericList<cExmaple>();

Console.ReadKey();

}

}

}

### 泛型概述

//泛型概述

using System;

using System.Collections.Generic;

namespace Zh.Test.Generic

{

//泛型类

public class cArr<T>

{

private T[] \_Arr;

public cArr(int index)

{ \_Arr = new T[index]; }

public void Show()

{

Console.Write("\n-------{0}---------\n", typeof(T).ToString());

foreach (var arr in \_Arr)

Console.Write("{0} ", arr);

}

public T[] Arr

{ get { return \_Arr; } }

public T this[int i]

{

get { return \_Arr[i]; }

set { \_Arr[i] = value; }

}

}

public class cProgram

{

static void Main(string[] args)

{

const int IntArrLength = 10;

const int strArrLength = 5;

//定义对象并赋值

cArr<int> intArr = new cArr<int>(IntArrLength);

for (int i = 0; i < intArr.Arr.Length; i++)

intArr[i] = 10 + i;//用索引器赋值

cArr<string> strArr = new cArr<string>(strArrLength);

for (int i = 0; i < strArr.Arr.Length; i++)

strArr.Arr[i] = "A" + i.ToString();//用属性赋值

//遍历输出

intArr.Show();

Show<int>(intArr.Arr);//用泛型方法

strArr.Show();

Show<string>(strArr.Arr);

Console.ReadKey();

}

private static void Show<U>(U[] arr)

{

foreach (var arr1 in arr)

Console.Write("\n{0}", arr1);

}

}

}

### 泛型的优点

#### System.Collections.ArrayList ArrList

//ArrayList的优缺点

using System;

using System.Collections.Generic;

using System.Collections;

namespace Zh.Test.Generic

{

public class cProgram

{

static void Main(string[] args)

{

ArrayList ArrList = new ArrayList();

//ArrayList可存储任意类型数据

//缺点:频繁装箱、拆箱，效率低下

ArrList.Add(31);

ArrList.Add(32);

ArrList.Add("ab1");

ArrList.Add("ab2");

ArrList.Add(2.123);

ArrList.Add(3.123);

foreach (var list1 in ArrList)

Console.Write("{0} ", list1);

Console.Write("\n---------Remove--------\n");

ArrList.Remove(31);

ArrList.RemoveAt(ArrList.Count - 1);

foreach (var list1 in ArrList)

Console.Write("{0} ", list1);

Console.ReadKey();

}

}

}

#### 泛型 List<T> 集合

//泛型 List<T> 集合

using System;

using System.Collections.Generic;

using System.Collections;

namespace Zh.Test.Generic

{

public class cProgram

{

static void Main(string[] args)

{

//高效安全：不需要拆箱、装箱，编译器可以进行类型检查

List<int> list = new List<int>();

for (int i = 0; i < 5; i++)

list.Add(i + 10);

foreach (var list1 in list)

Console.Write("{0} ", list1);

Console.Write("\n---------Remove--------\n");

list.Remove(11);

list.RemoveAt(list.Count - 1);

foreach (var list1 in list)

Console.Write("{0} ", list1);

Console.ReadKey();

}

}

}

#### 泛型类型参数

//类型参数命名准则

using System;

using System.Collections.Generic;

namespace Zh.Test.Generic

{

//1、务必使用描述性名称命名泛型类型参数

public delegate TOutput Converter<TInput, TOutput>(TInput from);

public class List<T> { /\*...\*/ }

//2、务必将“T”作为描述性类型参数名的前缀。

public interface ISessionChannel<TSession>

{

TSession Session { get; }

}

public class cProgram

{

//3、考虑使用 T 作为具有单个字母类型参数的类型的类型参数名。

public int IComparer<TTSession>(){return 0;}

public delegate bool Predicate<T>(T item);

public struct NullAble<T> where T : struct { }

static void Main(string[] args)

{

Console.ReadKey();

}

}

}

#### 类型参数的约束

##### 关键字 where 应用

//关键字 where

using System;

using System.Collections.Generic;

namespace Zh.Test.Generic

{

public class cEmployee

{

string \_Name;

int \_Id;

public cEmployee(string sName,int sId)

{

\_Name = sName;

\_Id = sId;

}

public string Name

{

get { return \_Name; }

set { \_Name = value;}

}

public int Id

{

get { return \_Id; }

set { \_Id = value; }

}

}

public class cGenericList<T>where T:cEmployee

{

private class cNode

{

cNode \_Next;

T \_Data;

public cNode(T t)

{

\_Next = null;

\_Data = t;

}

public cNode Next

{

get { return \_Next; }

set { \_Next = value; }

}

public T Data

{

get { return \_Data; }

set { \_Data = value; }

}

}

private cNode \_Head;

public cGenericList()

{

\_Head = null;

}

public void AddHead(T t)

{

cNode n = new cNode(t);

n.Next = \_Head;

\_Head = n;

}

public IEnumerator<T>GetEnumerator()

{

cNode \_Current = \_Head;

while (\_Current!=null)

{

yield return \_Current.Data;

\_Current = \_Current.Next;

}

}

public T FindFirstOccurrence(string s)

{

cNode \_Current = \_Head;

T t = null;

while (\_Current!=null)

{

if (\_Current.Data.Name==s)

{

t = \_Current.Data;

break;

}

else

{

\_Current = \_Current.Next;

}

}

return t;

}

}

public class cProgram

{

static void Main(string[] args)

{

List<cEmployee> list = new List<cEmployee>();

cGenericList<cEmployee> gl = new cGenericList<cEmployee>();

cEmployee emp = new cEmployee("刘大", 100);

gl.AddHead(emp);

list.Add(emp);

emp = new cEmployee("刘二", 102);

gl.AddHead(emp);

list.Add(emp);

emp = new cEmployee("刘三", 103);

gl.AddHead(emp);

list.Add(emp);

emp = new cEmployee("刘四", 104);

gl.AddHead(emp);

list.Add(emp);

foreach (var temp in list)

Console.WriteLine("姓名:{0} 编号：{1}", temp.Name, temp.Id);

Console.WriteLine();

Console.WriteLine("姓名:{0} 编号：{1}", emp.Name, emp.Id);

emp = gl.FindFirstOccurrence("刘二");

Console.WriteLine("姓名:{0} 编号：{1}", emp.Name, emp.Id);

emp = gl.FindFirstOccurrence("刘大");

Console.WriteLine("姓名:{0} 编号：{1}", emp.Name, emp.Id);

emp = gl.FindFirstOccurrence("刘三");

Console.WriteLine("姓名:{0} 编号：{1}", emp.Name, emp.Id);

Console.ReadKey();

}

}

}

##### 可以对同一类型参数应用多个约束

//可以对同一类型参数应用多个约束

using System;

using System.Collections.Generic;

namespace Zh.Test.Generic

{

public interface IEmployee { }

public class cEmployee:IEmployee { }

public class cEmployeeLis<T>where T:cEmployee,IEmployee,new(){ }

public class cProgram

{

static void Main(string[] args)

{

cEmployeeLis<cEmployee> el = new cEmployeeLis<cEmployee>();

Console.ReadKey();

}

}

}

##### 约束多个参数

//约束多个参数

using System;

using System.Collections.Generic;

namespace Zh.Test.Generic

{

public interface IEmployee { }

public class cEmployee : IEmployee { }

struct structTest { }

public class cBase<T,U>

where T:cEmployee

where U:struct

{ }

public class cProgram

{

static void Main(string[] args)

{

cBase<cEmployee, structTest> el = new cBase<cEmployee, structTest>();

Console.ReadKey();

}

}

}

## 索引指示器

### 示例

using System;

using zh.TestArrary;

namespace zh.TestArrary

{

public class A

{

int[] \_intArr ;//= new int[5];

public A()

{

\_intArr = new int[5];

for (int i = 0; i < \_intArr.Length; i++)

\_intArr[i] = i + 10;

}

public A(int arrLength)

{

\_intArr = new int[arrLength];

for (int i = 0; i < \_intArr.Length; i++)

\_intArr[i] = i + 10;

}

public int this[int index]

{

get { return \_intArr[index]; }

set { \_intArr[index] = value; }

}

public int ArrLength()

{

return \_intArr.Length;

}

}

}

namespace zh.test

{

class MyTest

{

static void Main()

{

System.Console.WriteLine("------A a = new A();------");

A a = new A();

for (int i = 0; i < a.ArrLength(); i++)

System.Console.Write("{0} ", a[i]);

System.Console.WriteLine("");

System.Console.WriteLine("------A a1 = new A(6);------");

A a1 = new A(6);

for (int i = 0; i < a1.ArrLength(); i++)

System.Console.Write("{0} ", a1[i]);

System.Console.WriteLine("");

System.Console.WriteLine("------A a2 = new A(7);------");

A a2 = new A(7);

for (int i = 0; i < a2.ArrLength(); i++)

System.Console.Write("{0} ", a2[i]);

a[2] = 100;

System.Console.WriteLine("");

System.Console.WriteLine("------a[2] a2[2]------");

System.Console.Write("a[2]={0} a2[2]={1}", a[2], a2[2]);

System.Console.WriteLine("");

}

}

}

## 类

### 示例

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using zh.Class;

namespace zh.Class

{

//抽象类：动物

//具有属性：物种、昵称、性别、年龄、嗜好

public abstract class Animal

{

//域

protected string \_Species;//物种:狗、猫、人...

string \_Nickname; //昵称

string \_Sex;

int \_Age;

protected string \_Hobby = "暂空"; //嗜好、爱好

//构造函数

protected Animal(string sSex, int sAge)

{

SetSpecies();

\_Sex = sSex;

\_Age = sAge;

}

//抽象方法 派生类必须实现功能

protected abstract void SetSpecies();

//虚方法

public virtual void SetHobby(string sHobby)

{

\_Hobby = "暂空";

}

//属性

public string Species

{ get { return \_Species; } }

public string Nickname//读写

{

get { return \_Nickname; }

set { \_Nickname = value; }

}

public string Sex

{

get { return \_Sex; }

}

public int Age

{

get { return \_Age; }

}

public string Hobby

{

get { return \_Hobby; }

}

}

//宠物类

public abstract class Pet : Animal

{

//增加域

string \_OwnerName;//主人姓名

protected string \_Cry;//叫声

//构造函数

public Pet(string sOwnerName, string sNickname, string sSex, int sAge)

: base(sSex, sAge)

{

SetSpecies();

//SetHobby("肉骨头");

SetCry();

\_OwnerName = sOwnerName;

Nickname = sNickname;

}

//增加虚方法

protected virtual void SetCry()

{

\_Cry = "暂空";

}

//重载方法

public override void SetHobby(string sHobby)

{

base.\_Hobby = sHobby;

}

//新增方法

public void Show()

{

Console.WriteLine("OwnerName={0};Nickname={1};Sex={2};Age={3}:", OwnerName, Nickname, Sex, Age);

Console.WriteLine("Species={0};嗜好={1};叫声:{2}", Species, Hobby, Cry);

}

//属性

public string OwnerName

{ get { return \_OwnerName; } }

public string Cry

{ get { return \_Cry; } }

}

public class Dog : Pet

{

//构造函数

public Dog(string sOwnerName, string sNickname, string sSex, int sAge)

: base(sOwnerName, sNickname, sSex, sAge)

{

SetHobby("肉骨头");

}

//重载方法

protected override void SetSpecies()

{ base.\_Species = "狗"; }

protected override void SetCry()

{ base.\_Cry = "汪汪"; }

}

public class Cat : Pet

{

//构造函数

public Cat(string sOwnerName, string sNickname, string sSex, int sAge)

: base(sOwnerName, sNickname, sSex, sAge)

{

SetHobby("鱼");

}

//重载方法

protected override void SetSpecies()

{ base.\_Species = "猫"; }

protected override void SetCry()

{ base.\_Cry = "喵喵"; }

}

public abstract class Person : Animal

{

//增加域

string \_Name;

string \_Identity; //身份：学生、工人、军人

//构造函数

protected Person(string sName, string sSex, int sAge, string sIdentity)

: base(sSex, sAge)

{

SetSpecies();

\_Name = sName;

\_Identity = sIdentity;

}

//重载抽象方法SetSpecies

protected override void SetSpecies()

{

base.\_Species = "人";

}

//此处未重载SetHobby，留到派生类重载

//属性

public string Name

{ get { return \_Name; } }

protected string Identity

{

get { return \_Identity; }

}

}

//Person派生类：Group

public class Group : Person

{

string \_Number;//编号:学号，工号，编号

int \_Years;//时长：学龄、工龄、军龄

protected string \_Duty;//职责：努力学习、努力工作、保家卫国

//构造函数

protected Group(string sName, string sSex, int sAge, string sIdentity, string sNumber, int sYears)

: base(sName, sSex, sAge, sIdentity)

{

\_Number = sNumber;

\_Years = sYears;

}

//重载虚方法，若不重写，将默认调用 Animal.SetHobby()

public override void SetHobby(string sHobby)//设置爱好：如比赛、竞技、射击

{

base.\_Hobby = sHobby;

}

//新增方法

public void ShowDuty() //职责

{

Console.WriteLine("{0}的职责：{1}", Identity, Duty);

}

public void Show()

{

Console.WriteLine("Name={0};Sex={1};Age={2}:", Name, Sex, Age);

Console.WriteLine("身份={0};编号={1};年限={2}:", Identity, Number, Years);

Console.WriteLine("昵称={0};爱好={1};职责={2}:", Nickname, Hobby, Duty);

Console.WriteLine("Species={0}", Species);

}

//属性

public string Number//编号

{ get { return \_Number; } }

public int Years//年限

{ get { return \_Years; } }

protected string Duty//职责

{ get { return \_Duty; } }

}

//定义派生类

public class Student : Group

{

string \_School;//所属学校

public Student(string sName, string sSex, int sAge, string sIdentity,

string sNumber, int sYears, string sSchool)

: base(sName, sSex, sAge, sIdentity, sNumber, sYears)

{

base.Nickname = "小顽皮";

SetHobby("电子竞技");

\_School = sSchool;

base.\_Duty = "努力学习";

}

public string School

{ get { return \_School; } }

public new void Show()

{

Console.WriteLine("所在学校:{0}", School);

base.Show();

}

}

public class Worker : Group

{

string \_Factory;//工作单位

public Worker(string sName, string sSex, int sAge, string sIdentity,

string sNumber, int sYears, string sFactory)

: base(sName, sSex, sAge, sIdentity, sNumber, sYears)

{

base.Nickname = "老顽童";

base.\_Duty = "努力工作";

SetHobby("棋牌比赛");

\_Factory = sFactory;

}

public string Factory

{ get { return \_Factory; } }

public new void Show()

{

Console.WriteLine("工作单位:{0}", Factory);

base.Show();

}

}

public class Soldier : Group

{

string \_Troops;//所属部队

public Soldier(string sName, string sSex, int sAge, string sIdentity,

string sNumber, int sYears, string sTroops)

: base(sName, sSex, sAge, sIdentity, sNumber, sYears)

{

base.Nickname = "兵王";

base.\_Duty = "保家卫国";

SetHobby("射击比赛");

\_Troops = sTroops;

}

public string Troops

{ get { return \_Troops; } }

public new void Show()

{

Console.WriteLine("所在部队:{0}", Troops);

base.Show();

}

}

}

namespace ZhTest

{

class MyTest

{

static void Main()

{

Console.WriteLine("-----Dog-----");

Dog dog = new Dog("李小小", "欢欢", "公", 3);

dog.Show();

Console.WriteLine("-----Cat-----");

Cat cat = new Cat("李小小", "乐乐", "母", 2);

cat.Show();

Console.WriteLine("-----Student-----");

Student st = new Student("小华", "男", 12, "学生", "1001", 5, "A小学");

st.Show();

//Console.WriteLine();

//st.Nickname = "机灵鬼1";

//st.Show();

System.Console.WriteLine("-----Worker-----");

Worker wk = new Worker("武大", "男", 30, "工人", "2002", 10, "机械厂");

wk.Show();

System.Console.WriteLine("-----Soldier-----");

Soldier sd = new Soldier("李剑", "男", 25, "特种兵", "3003", 7, "1234部队");

sd.Show();

Console.ReadKey();

}

}

}

## 接口

### 示例

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Zh.Test.AppMain

{

interface IIfA

{

void F();

}

interface IIfB

{

void F();

}

class cA : IIfA

{

public void F()

{

Console.WriteLine("cA.F()");

}

}

class cB : IIfA

{

public void F()

{

Console.WriteLine("cB.F()");

}

}

//显式接口实现

class cC : IIfA, IIfB

{

void IIfA.F()

{

Console.WriteLine("cC.IIfA.F()");

}

void IIfB.F()

{

Console.WriteLine("cC.IIfB.F()");

}

}

class cObject

{

public void G(cA a)

{

a.F();

}

public void G(cB b)

{

b.F();

}

//隐式接口实现

public void G(IIfA iif)

{

iif.F();

}

}

public class AppMain

{

static void Main()

{

cA a = new cA();

cB b = new cB();

Console.WriteLine("--------class A、B--------");

a.F();

b.F();

cObject obj = new cObject();

Console.WriteLine("--------cObject G(cA a) G(cB b)--------");

obj.G(a);

obj.G(b);

IIfA iIfA;

Console.WriteLine("--------IIfA 隐式定义接口--------");

iIfA = a;

iIfA.F();

obj.G(iIfA);

iIfA = b;

iIfA.F();

obj.G(iIfA);

IIfB iIfB;

cC c = new cC();

Console.WriteLine("--------cC 显式声明接口--------");

iIfA = (IIfA)c;

iIfA.F();

iIfB = c;

iIfB.F();

Console.ReadKey();

}

}

}