# 数组

# 类和对象

类和结构 蓝图--》生成对象内存快  引用地址存储在存储在命名变量中，也可以存储在数组或集合中

使用这些变量来调用对象方法及访问对象公共属性的代码称为客户端代码

由于结构是值类型，因此结构对象的变量具有整个对象的副本

比较两个对象是否相等的情况分析：

1. 若要确定两个类实例是否引用内存中的同一位置（意味着它们具有相同的标识），可使用静态 [Equals](https://msdn.microsoft.com/zh-cn/library/bsc2ak47.aspx) 方法。 （[System.Object](https://msdn.microsoft.com/zh-cn/library/system.object.aspx) 是所有值类型和引用类型的隐式基类，其中包括用户定义的结构和类。）
2. 若要确定两个结构实例中的实例字段是否具有相同的值，可使用 [ValueType.Equals](https://msdn.microsoft.com/zh-cn/library/2dts52z7.aspx) 方法。 由于所有结构都隐式继承自 [System.ValueType](https://msdn.microsoft.com/zh-cn/library/system.valuetype.aspx)，因此可以直接在对象上调用该方法，Equals 的 [System.ValueType](https://msdn.microsoft.com/zh-cn/library/system.valuetype.aspx) 实现使用反射，因为它必须能够确定任何结构中有哪些字段。 在创建您自己的结构时，重写 Equals 方法可以提供针对您的类型的高效求等算法。
3. 要确定两个类实例中字段的值是否相等，您可以使用 [Equals](https://msdn.microsoft.com/zh-cn/library/bsc2ak47.aspx) 方法或 [== 运算符](https://msdn.microsoft.com/zh-cn/library/53k8ybth.aspx)。 但是，只有类通过已重写或重载提供关于那种类型对象的相等含义的自定义时，才能使用它们。 类也可以实现 [IEquatable<T>](https://msdn.microsoft.com/zh-cn/library/ms131187.aspx) 接口或 [IEqualityComparer<T>](https://msdn.microsoft.com/zh-cn/library/ms132151.aspx) 接口。 这两个接口都提供可用于测试值相等性的方法。 设计好重写 **Equals** 的类后，请务必遵循[如何：为类型定义值相等性（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/dd183755.aspx)和 [Object.Equals(Object)](https://msdn.microsoft.com/zh-cn/library/bsc2ak47.aspx) 中介绍的准则。

 //结构  
public struct Person  
{    public string Name;    public int Age;    public Person(string name, int age)  
    {  
        Name = name;  
        Age = age;  
    }  
}

继承

结构不支持继承，但可以实现接口。

构造函数不能继承。

抽象成员和虚成员是多态性的基础，多态性是面向对象的编程的第二个主要特性。

多态：

1.在运行时，派生类 当作基类使用，派生类向上赋值

2.基类生命虚方法，派生类实现，在运行时，使用派生类的重写方法；基类向下执行；

基类是统一，派生类是运行时具体；

字段不能是虚拟的，只有方法、属性、事件和索引器才可以是虚拟的。被派生类override 重写后，不管派生类当作基类还是本身类型，都会执行的是重写的自身成员 ，；virutal 和 abstruct 的基类修饰

使用新成员隐藏基类成员：重写；当作什么类的类型，执行什么类的方法；

public class BaseClass  
{    public void DoWork() { WorkField++; }    public int WorkField;    public int WorkProperty  
    {        get { return 0; }  
    }  
}public class DerivedClass : BaseClass  
{    public new void DoWork() { WorkField++; }    public new int WorkField;    public new int WorkProperty  
    {        get { return 0; }  
    }  
}

通过将派生类的实例强制转换为基类的实例，仍然可以从客户端代码访问隐藏的基类成员

虚拟成员不管继承了多少层，都是虚拟方法，

public sealed override void DoWork() { }

用上述关键字可以将虚拟方法变为真正的成员，派生类只能用new 重写；

通过base 访问基类；

静态类和静态成员

静态类是密封的，因此不可被继承。 它们不能从除 [Object](https://msdn.microsoft.com/zh-cn/library/system.object.aspx) 外的任何类中继承。 静态类不能包含实例构造函数，但可以包含静态构造函数。

 如果非静态类包含需要进行重要的初始化的静态成员，也应定义静态构造函数。

静态成员:

 静态方法和属性不能访问其包含类型中的非静态字段和事件，并且不能访问任何对象的实例变量（除非在方法参数中显式传递）。

 静态字段有两个常见的用法：一是记录已实例化对象的个数，二是存储必须在所有实例之间共享的值。

静态方法可以被重载但不能被重写，因为它们属于类，不属于类的任何实例。

虽然字段不能声明为 static const，但 [const](https://msdn.microsoft.com/zh-cn/library/e6w8fe1b.aspx) 字段的行为在本质上是静态的。

对静态方法的调用以 Microsoft 中间语言 (MSIL) 生成调用指令，而对实例方法的调用生成 callvirt 指令，该指令还检查 null 对象引用。 但是，两者之间的性能差异在大多数时候并不明显。

访问修饰符

[public](https://msdn.microsoft.com/zh-cn/library/yzh058ae.aspx)、[private](https://msdn.microsoft.com/zh-cn/library/st6sy9xe.aspx)、[protected](https://msdn.microsoft.com/zh-cn/library/bcd5672a.aspx)、[internal](https://msdn.microsoft.com/zh-cn/library/7c5ka91b.aspx) 或 **protected internal**。

# 成员

字段

字段”是直接在[类](https://msdn.microsoft.com/zh-cn/library/0b0thckt.aspx)或[结构](https://msdn.microsoft.com/zh-cn/library/ah19swz4.aspx)中声明的任何类型的变量。字段是其包含类型的“成员”。

通常应仅为具有私有或受保护可访问性的变量使用字段。您的类向客户端代码公开的数据应通过[方法](https://msdn.microsoft.com/zh-cn/library/ms173114.aspx)、[属性](https://msdn.microsoft.com/zh-cn/library/x9fsa0sw.aspx)和[索引器](https://msdn.microsoft.com/zh-cn/library/6x16t2tx.aspx)提供。通过使用这些构造间接访问内部字段，可以针对无效的输入值提供防护。

存储由公共属性公开的数据的私有字段称为“后备存储”或“支持字段”。

字段通常存储这样的数据：该数据必须可供多个类方法访问，并且其存储期必须长于任何单个方法的生存期。

不在单个方法范围外部使用的变量应在方法体自身范围内声明为局部变量。

字段初始值设定项不能引用其他实例字段。

可以将字段声明为 [readonly](https://msdn.microsoft.com/zh-cn/library/acdd6hb7.aspx)。只读字段只能在初始化期间或在构造函数中赋值。 **static** **readonly** 字段非常类似于常数，只不过 C# 编译器不能在编译时访问静态只读字段的值，而只能在运行时访问。

常量

常量是在编译时已知并在程序的生存期内不发生更改的不可变值。常量使用 [const](https://msdn.microsoft.com/zh-cn/library/e6w8fe1b.aspx) 修饰符进行声明。

只有 C# 内置类型（[System.Object](https://msdn.microsoft.com/zh-cn/library/system.object.aspx) 除外）可以声明为 **const**。

用户定义的类型（包括类、结构和数组）不能为 **const**。请使用 [readonly](https://msdn.microsoft.com/zh-cn/library/acdd6hb7.aspx) 修饰符创建在运行时初始化一次即不可再更改的类、结构或数组。

C# 不支持 **const** 方法、属性或事件。

可以使用枚举类型为整数内置类型（例如 **int**、**uint**、**long** 等等）定义命名常量。

常量必须在声明时初始化。

若要定义整数类型（**int**、**byte** 等）的常量值，请使用枚举类型。

属性

属性是一种成员，它提供灵活的机制来读取、写入或计算私有字段的值。属性可用作公共数据成员，但它们实际上是称为“访问器”的特殊方法。

有助于提高方法的安全性和灵活性

public string Name => First + " " + Last;直接返回计算值,字段为只读,不能再使用get;

属性允许类公开获取和设置值的公共方法，而隐藏实现或验证代码。

[get](https://msdn.microsoft.com/zh-cn/library/ms228503.aspx) 属性访问器用于返回属性值，而 [set](https://msdn.microsoft.com/zh-cn/library/ms228368.aspx) 访问器用于分配新值。这些访问器可以具有不同的访问级别。

属性结合了字段和方法的多个方面。

对于对象的用户，属性显示为字段，访问该属性需要相同的语法。

对于类的实现者，属性是一个或两个代码块，表示一个 [get](https://msdn.microsoft.com/zh-cn/library/ms228503.aspx) 访问器和/或一个 [set](https://msdn.microsoft.com/zh-cn/library/ms228368.aspx) 访问器。

属性的数据的真实位置经常称为属性的“后备存储”。属性使用作为后备存储的私有字段是很常见的。

**set** 访问器类似于返回类型为 [void](https://msdn.microsoft.com/zh-cn/library/yah0tteb.aspx) 的方法。它使用称为 value 的隐式参数，此参数的类型是属性的类型。

可以使用 **static** 关键字将属性声明为静态属性。

可以使用 [virtual](https://msdn.microsoft.com/zh-cn/library/9fkccyh4.aspx) 关键字将属性标记为虚属性。

接口属性

接口属性的访问器不具有体。因此，访问器的用途是指示属性是否为读写、只读或只写。

interface IEmployee

{

string Name

{

get;

set;

}

int Counter

{

get;

}

}

除了声明和调用语法不同外，虚拟属性的行为与抽象方法一样。

修改修饰符: 虚拟属性的重写 用override

可访问域的修改:

方法:

方法是包含一系列语句的代码块。程序通过调用该方法并指定任何所需的方法参数使语句得以执行

在 C# 中，每个执行的指令均在方法的上下文中执行。

Main 方法是每个 C# 应用程序的入口点，并在启动程序时由公共语言运行时 (CLR) 调用。

方法签名包含:

访问级别（如 **public** 或 **private**）、可选修饰符（如 **abstract** 或 **sealed**）、返回值、方法的名称以及任何方法参数

按引用传递和按值传递

值类型可以设置为引用类型传递

数组传递时,默认不可操作传入的数组对象,修改数组内部文件时,改变传入的数组;使用ref 改为绝对的引用模式;

异步方法:

如果用 [async](https://msdn.microsoft.com/zh-cn/library/hh156513.aspx) 修饰符标记方法，则可以使用该方法中的 [await](https://msdn.microsoft.com/zh-cn/library/hh156528.aspx) 运算符。当控件到达异步方法中的 await 表达式时，控件将返回到调用方，并在等待任务完成前，方法中进度将一直处于挂起状态。任务完成后，可以在方法中恢复执行。

private async void startButton\_Click(object sender, RoutedEventArgs e)

{

await DoSomethingAsync();

}

private async Task DoSomethingAsync()

{

Task<int> delayTask = DelayAsync();

int result = await delayTask;

// The previous two statements may be combined into

// the following statement.

//int result = await DelayAsync();

Debug.WriteLine("Result: " + result);

}

private async Task<int> DelayAsync()

{

await Task.Delay(100);

return 5;

}