# 1.怎么识别属性和字段

## 在C#中，一般属性名是字段名的首字母大写（字段名为age,属性名为Age）

**属性（Property）** 是类（class）、结构（structure）和接口（interface）的命名成员。类或结构中的成员变量或方法称为 **字段（Field）**。属性（Property）是字段（Field）的扩展，且可使用相同的语法来访问。它们使用 **访问器（accessors）** 让私有域的值可被读写或操作。

# 2.我们一般是通过属性来操作字段,把属性设置为公有,把字段设置为私有

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”。

## 例子:

|  |
| --- |
| **public class Emp**  **{**  **//字段**  **private int age;**  **//属性**  **public int Age**  **{**  **get { return age; }**  **set { age = value; }**  **}**  **}** |

假如用户输入一个不符合实际的年龄，例如200，或者-18岁，如果我们不做点处理的话，肯定会造成数据不安全不严谨，那么对于这样的问题，我们需要怎样限制呢, 可以这样：

|  |
| --- |
| **public class Emp**  **{**  **//字段**  **private int age;**  **//属性**  **public int Age**  **{**  **get { return age; }**  **set {**  **if(value <= 100 && value>=0){**  **age = value;**  **}else{**  **age = 18;**  **}**  **}**  **}**  **}** |

# 3.字段的特点

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。例如，表示日历日期的类可能有三个整数字段：一个表示月份，一个表示日期，还有一个表示年份。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。

在类块中通过指定字段的访问级别，然后指定字段的类型，再指定字段的名称来声明这些字段。例如：

public class CalendarEntry

{

// private field

private DateTime date;

// public field (Generally not recommended.)

public string day;

// Public property exposes date field safely.

public DateTime Date

{

get

{

return date;

}

set

{

// Set some reasonable boundaries for likely birth dates.

if (value.Year > 1900 && value.Year <= DateTime.Today.Year)

{

date = value;

}

else

throw new ArgumentOutOfRangeException();

}

}

// Public method also exposes date field safely.

// Example call: birthday.SetDate("1975, 6, 30");

public void SetDate(string dateString)

{

DateTime dt = Convert.ToDateTime(dateString);

// Set some reasonable boundaries for likely birth dates.

if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)

{

date = dt;

}

else

throw new ArgumentOutOfRangeException();

}

public TimeSpan GetTimeSpan(string dateString)

{

DateTime dt = Convert.ToDateTime(dateString);

if (dt != null && dt.Ticks < date.Ticks)

{

return date - dt;

}

else

throw new ArgumentOutOfRangeException();

}

}

若要访问对象中的字段，请在对象名称后面添加一个句点，然后添加该字段的名称，比如 objectname.fieldname。例如：

CalendarEntry birthday = new CalendarEntry();

birthday.day = "Saturday";

声明字段时可以使用赋值运算符为字段指定一个初始值。例如，若要自动将 "Monday" 赋给 day 字段，需要声明 day，如下例所示：

public class CalendarDateWithInitialization

{

public string day = "Monday";

//...

}

字段的初始化紧靠调用对象实例的构造函数之前。如果构造函数为字段赋值，则该值将覆盖字段声明期间给出的任何值。有关更多信息，请参见 [使用构造函数（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173115.aspx)。

| **https://wizardforcel.gitbooks.io/msdn-csharp/content/img/note.jpg 注意** |
| --- |
| 字段初始值设定项不能引用其他实例字段。 |

字段可标记为 [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**。这些访问修饰符定义类的使用者访问字段的方式。有关更多信息，请参见 [访问修饰符（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173121.aspx)。

可以选择将字段声明为 [static](https://msdn.microsoft.com/zh-CN/library/98f28cdx.aspx)。这使得调用方在任何时候都能使用字段，即使类没有任何实例。有关更多信息，请参见 [静态类和静态类成员（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/79b3xss3.aspx)。

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。有关更多信息，请参见 [常量（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173119.aspx)。

# 4.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。这使得可以轻松访问数据，还有助于提高方法的安全性和灵活性。

在此示例中，TimePeriod 类存储时间段。该类在内部以秒为单位存储时间，但是名为 Hours 的属性允许客户端以小时为单位指定时间。 Hours 属性的访问器执行小时与秒之间的转换。

class TimePeriod

{

private double seconds;

public double Hours

{

get { return seconds / 3600; }

set { seconds = value \* 3600; }

}

}

class Program

{

static void Main()

{

TimePeriod t = new TimePeriod();

// Assigning the Hours property causes the 'set' accessor to be called.

t.Hours = 24;

// Evaluating the Hours property causes the 'get' accessor to be called.

System.Console.WriteLine("Time in hours: " + t.Hours);

}

}

// Output: Time in hours: 24

## 表达式主体定义

直接只返回表达式结果的属性很常见。下面的语法快捷方式使用 **=>** 来定义这些属性：

public string Name => First + " " + Last;

属性必须为只读，并且你不能使用 **get** 访问器关键字。

## 属性概述

### 1.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。属性或索引器的 [get](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/get) 和 [set](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/set) 部分称为访问器。 默认情况下，这些访问器具有与其所属属性或索引器相同的可见性或访问级别。 有关详细信息，请参阅[可访问性级别](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/accessibility-levels)。 不过，有时限制对其中某个访问器的访问是有益的。 通常，限制 set 访问器的可访问性，同时保持 get 访问器可公开访问。 例如：

C#

private string \_name = "Hello";

public string Name

{

get

{

return \_name;

}

protected set

{

\_name = value;

}

}

在此示例中，名为 Name 的属性定义 get 访问器和 set 访问器。 get 访问器接收该属性本身的可访问性级别（此示例中为 public），而对于 set 访问器，则通过对该访问器本身应用 [protected](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/protected) 访问修饰符来进行显式限制。

**备注**

本文中的示例不使用[**自动实现的属性**](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/properties#auto-implemented-properties)。 自动实现的属性提供了一种简洁的语法，用于在不需要自定义支持字段时声明属性。

### 对访问器的访问修饰符的限制

对属性或索引器使用访问修饰符受以下条件的制约：

* 不能对接口或显式[接口](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/interface)成员实现使用访问器修饰符。
* 仅当属性或索引器同时具有 set 和 get 访问器时，才能使用访问器修饰符。 这种情况下，只允许对其中一个访问器使用修饰符。
* 如果属性或索引器具有 [override](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/override) 修饰符，则访问器修饰符必须与重写的访问器的访问器（如有）匹配。
* 访问器的可访问性级别必须比属性或索引器本身的可访问性级别具有更严格的限制。

### 重写访问器的访问修饰符

重写属性或索引器时，被重写的访问器对重写代码而言必须是可访问的。 此外，属性/索引器及其访问器的可访问性都必须与相应的被重写属性/索引器及其访问器匹配。 例如：

C#复制

public class Parent

{

public virtual int TestProperty

{

// Notice the accessor accessibility level.

protected set { }

// No access modifier is used here.

get { return 0; }

}

}

public class Kid : Parent

{

public override int TestProperty

{

// Use the same accessibility level as in the overridden accessor.

protected set { }

// Cannot use access modifier here.

get { return 0; }

}

}

### 实现接口

使用访问器实现接口时，访问器不一定有访问修饰符。 但如果使用一个访问器（如 get）实现接口，则另一个访问器可以具有访问修饰符，如下面的示例所示：

C#复制

public interface ISomeInterface

{

int TestProperty

{

// No access modifier allowed here

// because this is an interface.

get;

}

}

public class TestClass : ISomeInterface

{

public int TestProperty

{

// Cannot use access modifier here because

// this is an interface implementation.

get { return 10; }

// Interface property does not have set accessor,

// so access modifier is allowed.

protected set { }

}

}

### 访问器可访问性域

如果对访问器使用访问修饰符，则访问器的[可访问性域](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/accessibility-domain)由该修饰符确定。

如果未对访问器使用访问修饰符，则访问器的可访问性域由属性或索引器的可访问性级别确定。

#### 示例

下面的示例包含三个类：BaseClass、DerivedClass 和 MainClass。 每个类的 BaseClass、Name 和 Id 都有两个属性。 该示例演示在使用限制性访问修饰符（如 [protected](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/protected) 或 [private](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/private)）时，BaseClass 的 Id 属性如何隐藏 DerivedClass 的 Id 属性。 因此，向该属性赋值时将调用 BaseClass 类中的属性。 将访问修饰符替换为 [public](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/public) 将使该属性可供访问。

该示例还演示了 DerivedClass 中 Name 属性的 set 访问器的限制性访问修饰符（例如 private 或 protected）如何阻止访问派生类中的访问器。 在向该修饰符赋值时，它会生成一个错误，或者访问同名的基类属性（如果可访问）。

C#

public class BaseClass

{

private string \_name = "Name-BaseClass";

private string \_id = "ID-BaseClass";

public string Name

{

get { return \_name; }

set { }

}

public string Id

{

get { return \_id; }

set { }

}

}

public class DerivedClass : BaseClass

{

private string \_name = "Name-DerivedClass";

private string \_id = "ID-DerivedClass";

new public string Name

{

get

{

return \_name;

}

// Using "protected" would make the set accessor not accessible.

set

{

\_name = value;

}

}

// Using private on the following property hides it in the Main Class.

// Any assignment to the property will use Id in BaseClass.

new private string Id

{

get

{

return \_id;

}

set

{

\_id = value;

}

}

}

class MainClass

{

static void Main()

{

BaseClass b1 = new BaseClass();

DerivedClass d1 = new DerivedClass();

b1.Name = "Mary";

d1.Name = "John";

b1.Id = "Mary123";

d1.Id = "John123"; // The BaseClass.Id property is called.

System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);

System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

// Keep the console window open in debug mode.

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

System.Console.ReadKey();

}

}

/\* Output:

Base: Name-BaseClass, ID-BaseClass

Derived: John, ID-BaseClass

\*/

#### 注释

注意，如果将声明 new private string Id 替换为 new public string Id，则得到如下输出：

Name and ID in the base class: Name-BaseClass, ID-BaseClass Name and ID in the derived class: John, John123

### 2.[value](https://msdn.microsoft.com/zh-cn/library/a1khb4f8.aspx) 关键字用于定义由 **set** 访问器分配的值。

### 3.不实现 **set** 访问器的属性均为只读。

### 4.对于不需要任何自定义访问器代码的简单属性，请考虑选择使用自动实现的属性的选项。

当属性访问器中不需要任何其他逻辑时，自动实现的属性会使属性声明更加简洁。 它们还允许客户端代码创建对象。 当你声明以下示例中所示的属性时，编译器将创建仅可以通过该属性的 get 和 set 访问器访问的专用、匿名支持字段。 init 访问器也可以声明为自动实现的属性。

#### 示例

下列示例演示一个简单的类，它具有某些自动实现的属性：

C#复制

// This class is mutable. Its data can be modified from

// outside the class.

public class Customer

{

// Auto-implemented properties for trivial get and set

public double TotalPurchases { get; set; }

public string Name { get; set; }

public int CustomerId { get; set; }

// Constructor

public Customer(double purchases, string name, int id)

{

TotalPurchases = purchases;

Name = name;

CustomerId = id;

}

// Methods

public string GetContactInfo() { return "ContactInfo"; }

public string GetTransactionHistory() { return "History"; }

// .. Additional methods, events, etc.

}

class Program

{

static void Main()

{

// Initialize a new object.

Customer cust1 = new Customer(4987.63, "Northwind", 90108);

// Modify a property.

cust1.TotalPurchases += 499.99;

}

}

不能在接口中声明自动实现的属性。 自动实现的属性声明一个私有实例支持字段，并且接口无法声明实例字段。 如果在接口中声明属性而不定义主体，请使用访问器声明属性，访问器必须由实现该接口的每个类型实现。

你可以像字段一样初始化自动实现属性：

C#

public string FirstName { get; set; } = "Jane";

上一示例中所示的类是可变的。 客户端代码在创建后可以更改对象中的值。 在包含重要行为（方法）以及数据的复杂类中，通常有必要具有公共属性。 但是，对于那些仅封装一组值（数据）且很少或没有行为的小型类或结构，应该使用以下选项之一使对象不可变：

* 只声明 get 访问器（除了能在构造函数中可变，在其他任何位置都不可变）。
* 声明 get 访问器和 init 访问器（除了能在对象构造函数中可变，在其他任何位置都不可变）。
* 将 set 访问器声明为[专用](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/private)（对使用者不可变）。