# 0.类成员

类和结构具有表示其数据和行为的成员。类的成员包括在类中声明的所有成员，以及在该类的继承层次结构中的所有类中声明的所有成员（构造函数和析构函数除外）。基类中的私有成员被继承，但不能从派生类访问。

下表列出类或结构中可包含的成员类型：

| **成员** | **描述** |
| --- | --- |
| [字段](https://msdn.microsoft.com/zh-CN/library/ms173118.aspx) | 字段是在类范围声明的变量。字段可以是内置数值类型或其他类的实例。例如，日历类可能具有一个包含当前日期的字段。 |
| [常量](https://msdn.microsoft.com/zh-CN/library/ms173119.aspx) | 常量是在编译时设置其值并且不能更改其值的字段或属性。 |
| [属性](https://msdn.microsoft.com/zh-CN/library/x9fsa0sw.aspx) | 属性是类中可以像类中的字段一样访问的方法。属性可以为类字段提供保护，以避免字段在对象不知道的情况下被更改。 |
| [方法](https://msdn.microsoft.com/zh-CN/library/ms173114.aspx) | 方法定义类可以执行的操作。方法可以接受提供输入数据的参数，并且可以通过参数返回输出数据。方法还可以不使用参数而直接返回值。 |
| [事件](https://msdn.microsoft.com/zh-CN/library/awbftdfh.aspx) | 事件向其他对象提供有关发生的事情（如单击按钮或成功完成某个方法）的通知。事件是使用委托定义和触发的。 |
| [运算符](https://msdn.microsoft.com/zh-CN/library/ms173145.aspx) | 重载运算符被视为类成员。在重载运算符时，在类中将该运算符定义为公共静态方法。预定义运算符（**+**、**\***、**<** 等）不考虑作为成员。有关详细信息，请参阅[可重载运算符（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/8edha89s.aspx)。 |
| [索引器](https://msdn.microsoft.com/zh-CN/library/6x16t2tx.aspx) | 使用索引器可以用类似于数组的方式为对象建立索引。 |
| [构造函数](https://msdn.microsoft.com/zh-CN/library/ace5hbzh.aspx) | 构造函数是在第一次创建对象时调用的方法。它们通常用于初始化对象的数据。 |
| [析构函数](https://msdn.microsoft.com/zh-CN/library/66x5fx1b.aspx) | C# 中极少使用析构函数。析构函数是当对象即将从内存中移除时由运行时执行引擎调用的方法。它们通常用来确保任何必须释放的资源都得到适当的处理。 |
| [嵌套类型](https://msdn.microsoft.com/zh-CN/library/ms173120.aspx) | 嵌套类型是在其他类型中声明的类型，通常用于描述仅由包含它们的类型所使用的对象。 |

# 1.创建对象

尽管有时类和对象可互换，但它们是不同的概念。类定义对象的类型，但它不是对象本身。对象是基于类的具体实体，有时称为类的实例。

通过使用 [new](https://msdn.microsoft.com/zh-CN/library/51y09td4.aspx) 关键字（后跟对象将基于的类的名称）可以创建对象，如下所示：

Customer object1 = new Customer();

创建类的实例后，将向程序员传递回对该对象的引用。在前面的示例中，object1 是对基于 Customer 的对象的引用。此引用引用新对象，但不包含对象数据本身。实际上，可以在根本不创建对象的情况下创建对象引用：

Customer object2;//这样子创建的对象根本无法使用.除非用另外一个对象的引用给他赋值

建议不要创建像这样的不引用对象的对象引用，因为在运行时通过这样的引用来访问对象的尝试将会失败。但是，可以创建这样的引用来引用对象，方法是创建新对象，或者将它分配给现有的对象，如下所示：

Customer object3 = new Customer();

Customer object4 = object3;

此代码创建了两个对象引用，它们引用同一个对象。因此，通过 object3 对对象所做的任何更改都将反映在随后使用的 object4 中。由于基于类的对象是按引用来引用的，因此类称为引用类型。

# 2.类继承

继承是通过使用“派生”来实现的，而派生意味着类是使用“基类”声明的，它的数据和行为从基类继承。通过在派生的类名后面追加冒号和基类名称，可以指定基类，如下所示：

public class Manager : Employee

{

// Employee fields, properties, methods and events are inherited

// New Manager fields, properties, methods and events go here...

}

当类声明基类时，它继承基类除构造函数以外的所有成员。

与 C++ 不同，C# 中的类只能直接从一个基类继承。但是，因为基类自身也可能继承自另一个类，所以类可以间接继承多个基类。而且，一个类可以直接实现一个以上的接口

类可以声明为[抽象](https://msdn.microsoft.com/zh-CN/library/sf985hc5.aspx)类。抽象类包含具有签名定义但没有实现的抽象方法。抽象类不能进行实例化。只能通过实现抽象方法的派生类使用抽象类。相比之下，[密封](https://msdn.microsoft.com/zh-CN/library/88c54tsw.aspx)类不允许其他类从其派生. 类定义可在不同的源文件之间进行拆分,如部分类.

## 实例

|  |
| --- |
| public class Person  {  // Field  public string name;  // Constructor that takes no arguments.  public Person()  {  name = "unknown";  }  // Constructor that takes one argument.  public Person(string nm)  {  name = nm;  }  // Method  public void SetName(string newName)  {  name = newName;  }  }  class TestPerson  {  static void Main()  {  // Call the constructor that has no parameters.  Person person1 = new Person();  Console.WriteLine(person1.name);  person1.SetName("John Smith");  Console.WriteLine(person1.name);  // Call the constructor that has one parameter.  Person person2 = new Person("Sarah Jones");  Console.WriteLine(person2.name);  // Keep the console window open in debug mode.  Console.WriteLine("Press any key to exit.");  Console.ReadKey();  }  }  // Output:  // unknown  // John Smith  // Sarah Jones |

# 3类对象

对象是按照此蓝图分配和配置的内存块。程序可以创建同一个类的多个对象。对象也称为实例，可以存储在命名变量中，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。在 C# 等面向对象的语言中，典型的程序由动态交互的多个对象组成。

| **https://wizardforcel.gitbooks.io/msdn-csharp/content/img/note.jpg 注意** |
| --- |
| 静态类型的行为与此处介绍的不同 |

由于类是引用类型，因此类对象的变量引用该对象在托管堆上的地址。如果将同一类型的第二个对象分配给第一个对象，则两个变量都引用该地址的对象

类的实例是使用 [new 运算符](https://msdn.microsoft.com/zh-CN/library/fa0ab757.aspx)创建的

在下面的示例中，Person 为类型，person1 和 person 2 为该类型的实例（即对象）。

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

//Other properties, methods, events...

}

class Program

{

static void Main()

{

Person person1 = new Person("Leopold", 6);

Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

// Declare new person, assign person1 to it.

Person person2 = person1;

//Change the name of person2, and person1 also changes.

person2.Name = "Molly";

person2.Age = 16;

Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name, person2.Age);

Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

// Keep the console open in debug mode.

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

Console.ReadKey();

}

}

/\*

Output:

person1 Name = Leopold Age = 6

person2 Name = Molly Age = 16

person1 Name = Molly Age = 16

\*/

# 4.结构体对象

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。结构的实例也可以使用 **new** 运算符来创建，但这不是必需的，如下面的示例所示：

public struct Person

{

public string Name;

public int Age;

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

public class Application

{

static void Main()

{

// Create struct instance and initialize by using "new".

// Memory is allocated on thread stack.

Person p1 = new Person("Alex", 9);

Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

// Create new struct object. Note that struct can be initialized

// without using "new".

Person p2 = p1;

// Assign values to p2 members.

p2.Name = "Spencer";

p2.Age = 7;

Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

// p1 values remain unchanged because p2 is copy.

Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

// Keep the console open in debug mode.

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

Console.ReadKey();

}

}

/\*

Output:

p1 Name = Alex Age = 9

p2 Name = Spencer Age = 7

p1 Name = Alex Age = 9

\*/

p1 和 p2 的内存在线程堆栈上进行分配。该内存随声明它的类型或方法一起回收。这就是在赋值时复制结构的一个原因。相比之下，当对类实例对象的所有引用都超出范围时，为该类实例分配的内存将由公共语言运行时自动回收（垃圾回收）。无法像在 C++ 中那样明确地销毁类对象。

| **注意** |
| --- |
| 公共语言运行时中高度优化了托管堆上内存的分配和释放。在大多数情况下，在堆上分配类实例与在堆栈上分配结构实例在性能开销上没有显著的差别。 |

# 5. 对象标识与. 值相等性

在比较两个对象是否相等时，首先必须明确您是想知道两个变量是否表示内存中的同一对象，还是想知道这两个对象的一个或多个字段的值是否相等。如果您要对值进行比较，则必须考虑这两个对象是值类型（结构）的实例，还是引用类型（类、委托、数组）的实例。

* 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。）
* 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，如下面的示例所示：

## 类示例

下面的示例演示如何在类（引用类型）中实现值相等性。

C#复制

namespace ValueEqualityClass;

class TwoDPoint : IEquatable<TwoDPoint>

{

public int X { get; private set; }

public int Y { get; private set; }

public TwoDPoint(int x, int y)

{

if (x is (< 1 or > 2000) || y is (< 1 or > 2000))

{

throw new ArgumentException("Point must be in range 1 - 2000");

}

this.X = x;

this.Y = y;

}

public override bool Equals(object obj) => this.Equals(obj as TwoDPoint);

public bool Equals(TwoDPoint p)

{

if (p is null)

{

return false;

}

// Optimization for a common success case.

if (Object.ReferenceEquals(this, p))

{

return true;

}

// If run-time types are not exactly the same, return false.

if (this.GetType() != p.GetType())

{

return false;

}

// Return true if the fields match.

// Note that the base class is not invoked because it is

// System.Object, which defines Equals as reference equality.

return (X == p.X) && (Y == p.Y);

}

public override int GetHashCode() => (X, Y).GetHashCode();

public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)

{

if (lhs is null)

{

if (rhs is null)

{

return true;

}

// Only the left side is null.

return false;

}

// Equals handles case of null on right side.

return lhs.Equals(rhs);

}

public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs == rhs);

}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.

class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>

{

public int Z { get; private set; }

public ThreeDPoint(int x, int y, int z)

: base(x, y)

{

if ((z < 1) || (z > 2000))

{

throw new ArgumentException("Point must be in range 1 - 2000");

}

this.Z = z;

}

public override bool Equals(object obj) => this.Equals(obj as ThreeDPoint);

public bool Equals(ThreeDPoint p)

{

if (p is null)

{

return false;

}

// Optimization for a common success case.

if (Object.ReferenceEquals(this, p))

{

return true;

}

// Check properties that this class declares.

if (Z == p.Z)

{

// Let base class check its own fields

// and do the run-time type comparison.

return base.Equals((TwoDPoint)p);

}

else

{

return false;

}

}

public override int GetHashCode() => (X, Y, Z).GetHashCode();

public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)

{

if (lhs is null)

{

if (rhs is null)

{

// null == null = true.

return true;

}

// Only the left side is null.

return false;

}

// Equals handles the case of null on right side.

return lhs.Equals(rhs);

}

public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs) => !(lhs == rhs);

}

class Program

{

static void Main(string[] args)

{

ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);

ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);

ThreeDPoint pointC = null;

int i = 5;

Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));

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

Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));

Console.WriteLine("Compare to some other type = {0}", pointA.Equals(i));

TwoDPoint pointD = null;

TwoDPoint pointE = null;

Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD == pointE);

pointE = new TwoDPoint(3, 4);

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

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

Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

System.Collections.ArrayList list = new System.Collections.ArrayList();

list.Add(new ThreeDPoint(3, 4, 5));

Console.WriteLine("pointE.Equals(list[0]): {0}", pointE.Equals(list[0]));

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

}

/\* Output:

pointA.Equals(pointB) = True

pointA == pointB = True

null comparison = False

Compare to some other type = False

Two null TwoDPoints are equal: True

(pointE == pointA) = False

(pointA == pointE) = False

(pointA != pointE) = True

pointE.Equals(list[0]): False

\*/

在类（引用类型）上，两种 [Object.Equals(Object)](https://learn.microsoft.com/zh-cn/dotnet/api/system.object.equals#system-object-equals(system-object)) 方法的默认实现均执行引用相等性比较，而不是值相等性检查。 实施者替代虚方法时，目的是为其指定值相等性语义。

即使类不重载 == 和 != 运算符，也可将这些运算符与类一起使用。 但是，默认行为是执行引用相等性检查。 在类中，如果重载 Equals 方法，则应重载 == 和 != 运算符，但这并不是必需的。

**重要**

前面的示例代码可能无法按照预期的方式处理每个继承方案。 考虑下列代码：

C#复制

TwoDPoint p1 = new ThreeDPoint(1, 2, 3);

TwoDPoint p2 = new ThreeDPoint(1, 2, 4);

Console.WriteLine(p1.Equals(p2)); // output: True

根据此代码报告，尽管 z 值有所不同，但 p1 等于 p2。 由于编译器会根据编译时类型选取 IEquatable 的 TwoDPoint 实现，因而会忽略该差异。

record 类型的内置值相等性可以正确处理这类场景。 如果 TwoDPoint 和 ThreeDPoint 是 record 类型，则 p1.Equals(p2) 的结果会是 False。 有关详细信息，请参阅 [**record 类型继承层次结果中的相等性**](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/record#equality-in-inheritance-hierarchies)。

## 结构示例

下面的示例演示如何在结构（值类型）中实现值相等性：

C#复制

namespace ValueEqualityStruct

{

struct TwoDPoint : IEquatable<TwoDPoint>

{

public int X { get; private set; }

public int Y { get; private set; }

public TwoDPoint(int x, int y)

: this()

{

if (x is (< 1 or > 2000) || y is (< 1 or > 2000))

{

throw new ArgumentException("Point must be in range 1 - 2000");

}

X = x;

Y = y;

}

public override bool Equals(object? obj) => obj is TwoDPoint other && this.Equals(other);

public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;

public override int GetHashCode() => (X, Y).GetHashCode();

public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) => lhs.Equals(rhs);

public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs == rhs);

}

class Program

{

static void Main(string[] args)

{

TwoDPoint pointA = new TwoDPoint(3, 4);

TwoDPoint pointB = new TwoDPoint(3, 4);

int i = 5;

// True:

Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));

// True:

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

// True:

Console.WriteLine("object.Equals(pointA, pointB) = {0}", object.Equals(pointA, pointB));

// False:

Console.WriteLine("pointA.Equals(null) = {0}", pointA.Equals(null));

// False:

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

// True:

Console.WriteLine("(pointA != null) = {0}", pointA != null);

// False:

Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));

// CS0019:

// Console.WriteLine("pointA == i = {0}", pointA == i);

// Compare unboxed to boxed.

System.Collections.ArrayList list = new System.Collections.ArrayList();

list.Add(new TwoDPoint(3, 4));

// True:

Console.WriteLine("pointA.Equals(list[0]): {0}", pointA.Equals(list[0]));

// Compare nullable to nullable and to non-nullable.

TwoDPoint? pointC = null;

TwoDPoint? pointD = null;

// False:

Console.WriteLine("pointA == (pointC = null) = {0}", pointA == pointC);

// True:

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

TwoDPoint temp = new TwoDPoint(3, 4);

pointC = temp;

// True:

Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA == pointC);

pointD = temp;

// True:

Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD == pointC);

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

Console.ReadKey();

}

}

/\* Output:

pointA.Equals(pointB) = True

pointA == pointB = True

Object.Equals(pointA, pointB) = True

pointA.Equals(null) = False

(pointA == null) = False

(pointA != null) = True

pointA.Equals(i) = False

pointE.Equals(list[0]): True

pointA == (pointC = null) = False

pointC == pointD = True

pointA == (pointC = 3,4) = True

pointD == (pointC = 3,4) = True

\*/

}

对于结构，[Object.Equals(Object)](https://learn.microsoft.com/zh-cn/dotnet/api/system.object.equals#system-object-equals(system-object))（[System.ValueType](https://learn.microsoft.com/zh-cn/dotnet/api/system.valuetype) 中的替代版本）的默认实现通过使用反射来比较类型中每个字段的值，从而执行值相等性检查。 实施者替代结构中的 Equals 虚方法时，目的是提供更高效的方法来执行值相等性检查，并选择根据结构字段或属性的某个子集来进行比较。

除非结构显式重载了 [==](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/equality-operators#equality-operator-) 和 [!=](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/equality-operators#inequality-operator-) 运算符，否则这些运算符无法对结构进行运算。

## 结构体实例2.

// Person is defined in the previous example.

//public struct Person

//{

// public string Name;

// public int Age;

// public Person(string name, int age)

// {

// Name = name;

// Age = age;

// }

//}

Person p1 = new Person("Wallace", 75);

Person p2;

p2.Name = "Wallace";

p2.Age = 75;

if (p2.Equals(p1))

Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.

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。

* 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

# 6.结构体

结构是使用 [struct](https://msdn.microsoft.com/zh-cn/library/ah19swz4.aspx) 关键字定义的，例如：

public struct PostalAddress

{

// Fields, properties, methods and events go here...

}

结构与类共享大多数相同的语法，但结构比类受到的限制更多：

* 在结构声明中，除非字段被声明为 const 或 static，否则无法初始化。
* 结构不能声明默认构造函数（没有参数的构造函数）或析构函数。
* 结构在赋值时进行复制。将结构赋值给新变量时，将复制所有数据，并且对新副本所做的任何修改不会更改原始副本的数据。在使用值类型的集合（如 Dictionary<string, myStruct>）时，请务必记住这一点。
* 结构是值类型，而类是引用类型。
* 与类不同，结构的实例化可以不使用 **new** 运算符。
* 结构可以声明带参数的构造函数。
* 一个结构不能从另一个结构或类继承，而且不能作为一个类的基。所有结构都直接继承自 **System.ValueType**，后者继承自 **System.Object**。
* 结构可以实现接口。
* 结构可用作可以为 null 的类型，因而可向其赋 null 值

## 请参阅

[C# 编程指南](https://msdn.microsoft.com/zh-CN/library/67ef8sbd.aspx)

[类（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/x9afc042.aspx)

[方法（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173114.aspx)

[构造函数（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ace5hbzh.aspx)

[析构函数（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/66x5fx1b.aspx)

[属性（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/x9fsa0sw.aspx)

[字段（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173118.aspx)

[索引器（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/6x16t2tx.aspx)

[事件（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/awbftdfh.aspx)

[嵌套类型（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173120.aspx)

[运算符（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173145.aspx)

[可重载运算符（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/8edha89s.aspx)