**继承（C# 编程指南）**

继承（加上封装和多态性）是面向对象的编程的三个主要特性（也称为“支柱”）之一。 继承用于创建可重用、扩展和修改在其他类中定义的行为的新类。其成员被继承的类称为“基类”，继承这些成员的类称为“派生类”。派生类只能有一个直接基类。但是，继承是可传递的。如果 ClassB 派生出 ClassC，ClassA 派生出 ClassB，则 ClassC 会继承 ClassB 和 ClassA 中声明的成员。

| **https://wizardforcel.gitbooks.io/msdn-csharp/content/img/note.jpg 注意** |
| --- |
| 结构不支持继承，但可以实现接口。有关更多信息，请参见 [接口（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173156.aspx)。 |

从概念上来说，派生类是基类的特例。 例如，如果您有一个基类 Animal，则可以有一个名为 Mammal 的派生类和一个名为 Reptile 的派生类。 Mammal 是一个 Animal，Reptile 也是一个 Animal，但每个派生类均表示基类的不同专用化。

定义一个类从其他类派生时，派生类隐式获得基类的除构造函数和析构函数以外的所有成员。因此，派生类可以重用基类中的代码而无需重新实现这些代码。可以在派生类中添加更多成员。派生类以这种方式扩展基类的功能。

下图演示一个 WorkItem 类，该类表示某业务流程中的一个工作项。和所有的类一样，该类派生自 [System.Object](https://msdn.microsoft.com/zh-CN/library/system.object.aspx) 并继承其所有方法。 WorkItem 添加了自己的五个成员。其中包括一个构造函数，因为构造函数不能继承。类ChangeRequest 继承自 WorkItem 并表示特定种类的工作项。 ChangeRequest 在它从 WorkItem 和 [Object](https://msdn.microsoft.com/zh-CN/library/system.object.aspx) 继承的成员中另外添加了两个成员。它必须添加其自己的构造函数，还添加 originalItemID。利用属性 originalItemID，可将 ChangeRequest 实例与更改请求将应用到的原始 WorkItem 相关联。

类继承

下面的示例演示如何以 C# 表示上图所示的类关系。该示例还演示 WorkItem 如何重写虚方法 [Object.ToString](https://msdn.microsoft.com/zh-CN/library/system.object.tostring.aspx)，以及 ChangeRequest 类如何继承该方法的 WorkItem 实现。

// WorkItem implicitly inherits from the Object class.

public class WorkItem

{

// Static field currentID stores the job ID of the last WorkItem that

// has been created.

private static int currentID;

//Properties.

protected int ID { get; set; }

protected string Title { get; set; }

protected string Description { get; set; }

protected TimeSpan jobLength { get; set; }

// Default constructor. If a derived class does not invoke a base-

// class constructor explicitly, the default constructor is called

// implicitly.

public WorkItem()

{

ID = 0;

Title = "Default title";

Description = "Default description.";

jobLength = new TimeSpan();

}

// Instance constructor that has three parameters.

public WorkItem(string title, string desc, TimeSpan joblen)

{

this.ID = GetNextID();

this.Title = title;

this.Description = desc;

this.jobLength = joblen;

}

// Static constructor to initialize the static member, currentID. This

// constructor is called one time, automatically, before any instance

// of WorkItem or ChangeRequest is created, or currentID is referenced.

static WorkItem()

{

currentID = 0;

}

protected int GetNextID()

{

// currentID is a static field. It is incremented each time a new

// instance of WorkItem is created.

return ++currentID;

}

// Method Update enables you to update the title and job length of an

// existing WorkItem object.

public void Update(string title, TimeSpan joblen)

{

this.Title = title;

this.jobLength = joblen;

}

// Virtual method override of the ToString method that is inherited

// from System.Object.

public override string ToString()

{

return String.Format("{0} - {1}", this.ID, this.Title);

}

}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)

// and two constructors.

public class ChangeRequest : WorkItem

{

protected int originalItemID { get; set; }

// Constructors. Because neither constructor calls a base-class

// constructor explicitly, the default constructor in the base class

// is called implicitly. The base class must contain a default

// constructor.

// Default constructor for the derived class.

public ChangeRequest() { }

// Instance constructor that has four parameters.

public ChangeRequest(string title, string desc, TimeSpan jobLen,

int originalID)

{

// The following properties and the GetNexID method are inherited

// from WorkItem.

this.ID = GetNextID();

this.Title = title;

this.Description = desc;

this.jobLength = jobLen;

// Property originalItemId is a member of ChangeRequest, but not

// of WorkItem.

this.originalItemID = originalID;

}

}

class Program

{

static void Main()

{

// Create an instance of WorkItem by using the constructor in the

// base class that takes three arguments.

WorkItem item = new WorkItem("Fix Bugs",

"Fix all bugs in my code branch",

new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in

// the derived class that takes four arguments.

ChangeRequest change = new ChangeRequest("Change Base Class Design",

"Add members to the class",

new TimeSpan(4, 0, 0),

1);

// Use the ToString method defined in WorkItem.

Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the

// ChangeRequest object.

change.Update("Change the Design of the Base Class",

new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.

Console.WriteLine(change.ToString());

// Keep the console open in debug mode.

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

Console.ReadKey();

}

}

/\* Output:

1 - Fix Bugs

2 - Change the Design of the Base Class

\*/

**抽象方法和虚方法**

当基类将方法声明为 [virtual](https://msdn.microsoft.com/zh-CN/library/9fkccyh4.aspx) 时，派生类可以用自己的实现[重写](https://msdn.microsoft.com/zh-CN/library/ebca9ah3.aspx)该方法。如果基类将成员声明为 [abstract](https://msdn.microsoft.com/zh-CN/library/sf985hc5.aspx)，则在直接继承自该类的任何非抽象类中都必须重写该方法。如果派生类自身是抽象的，则它继承抽象成员而不实现它们。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。有关更多信息，请参见 [多态性（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173152.aspx)。

**抽象基类**

如果希望禁止通过 [new](https://msdn.microsoft.com/zh-CN/library/51y09td4.aspx)关键字直接进行实例化，可以将类声明为[abstract](https://msdn.microsoft.com/zh-CN/library/sf985hc5.aspx)如果这样做，则仅当从该类派生新类时才能使用该类。抽象类可以包含一个或多个自身声明为抽象的方法签名。这些签名指定参数和返回值，但没有实现（方法体）。抽象类不必包含抽象成员；但是，如果某个类确实包含抽象成员，则该类自身必须声明为抽象类。自身不是抽象类的派生类必须为抽象基类中的任何抽象方法提供实现。有关更多信息，请参见 [抽象类、密封类及类成员（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173150.aspx)。

**接口**

“接口”是一种引用类型，有点像仅包含抽象成员的抽象基类。类在从接口实现时必须为该接口的所有成员提供实现。类虽然只能从一个直接基类派生，但可以实现多个接口。

接口用于为不一定具有“是”关系的类定义特定功能。例如，[System.IEquatable<T>](https://msdn.microsoft.com/zh-CN/library/ms131187.aspx) 接口可由任何类或构造实现，这些类或构造必须启用代码来确定该类型的两个对象是否等效（但是该类型定义等效性）。 [IEquatable<T>](https://msdn.microsoft.com/zh-CN/library/ms131187.aspx) 不表示基类和派生类之间存在的同一种“是”关系（例如 Mammal 是 Animal）。有关更多信息，请参见 [接口（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173156.aspx)。

**派生类对基类成员的访问**

派生类可以访问基类的公共成员、受保护成员、内部成员和受保护内部成员。即使派生类继承基类的私有成员，仍不能访问这些成员。但是，所有这些私有成员在派生类中仍然存在，且执行与基类自身中相同的工作。例如，假定一个受保护基类方法访问私有字段。要使继承的基类方法正常工作，派生类中必须有该字段。

**禁止进一步派生**

类可以将自身或其成员声明为 [sealed](https://msdn.microsoft.com/zh-CN/library/88c54tsw.aspx)，从而禁止其他类从该类自身或其任何成员继承。有关更多信息，请参见 [抽象类、密封类及类成员（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173150.aspx)。

**派生类隐藏基类成员**

派生类可以通过以相同的名称和签名声明基类成员来隐藏这些成员。可以使用 [new](https://msdn.microsoft.com/zh-CN/library/51y09td4.aspx) 修饰符显式指示成员不作为基类成员的重写。不是必须要使用 [new](https://msdn.microsoft.com/zh-CN/library/51y09td4.aspx)，但如果不使用 [new](https://msdn.microsoft.com/zh-CN/library/51y09td4.aspx)，将生成编译器警告。有关更多信息，请参见[使用 Override 和 New 关键字进行版本控制（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/6fawty39.aspx)和 [了解何时使用 Override 和 New 关键字（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173153.aspx)。

# 多态性（C# 编程指南）

多态性常被视为自封装和继承之后，面向对象的编程的第三个支柱。Polymorphism（多态性）是一个希腊词，指“多种形态”，多态性具有两个截然不同的方面：

* 在运行时，在方法参数和集合或数组等位置，派生类的对象可以作为基类的对象处理。发生此情况时，该对象的声明类型不再与运行时类型相同。
* 基类可以定义并实现[虚](https://msdn.microsoft.com/zh-cn/library/9fkccyh4.aspx)方法，派生类可以[重写](https://msdn.microsoft.com/zh-cn/library/ebca9ah3.aspx)这些方法，即派生类提供自己的定义和实现。在运行时，客户端代码调用该方法，CLR 查找对象的运行时类型，并调用虚方法的重写方法。因此，你可以在源代码中调用基类的方法，但执行该方法的派生类版本。

虚方法允许你以统一方式处理多组相关的对象。例如，假定你有一个绘图应用程序，允许用户在绘图图面上创建各种形状。你在编译时不知道用户将创建哪些特定类型的形状。但应用程序必须跟踪创建的所有类型的形状，并且必须更新这些形状以响应用户鼠标操作。你可以使用多态性通过两个基本步骤解决这一问题：

1. 创建一个类层次结构，其中每个特定形状类均派生自一个公共基类。
2. 使用虚方法通过对基类方法的单个调用来调用任何派生类上的相应方法。

首先，创建一个名为 Shape 的基类，并创建一些派生类，例如 Rectangle、Circle 和 Triangle。为 Shape 类提供一个名为 Draw 的虚方法，并在每个派生类中重写该方法以绘制该类表示的特定形状。创建一个 List<Shape> 对象，并向该对象添加 Circle、Triangle 和 Rectangle。若要更新绘图图面，请使用 [foreach](https://msdn.microsoft.com/zh-cn/library/ttw7t8t6.aspx) 循环对该列表进行循环访问，并对其中的每个 Shape 对象调用 Draw 方法。虽然列表中的每个对象都具有声明类型 Shape，但调用的将是运行时类型（该方法在每个派生类中的重写版本）。

public class Shape

{

// A few example members

public int X { get; private set; }

public int Y { get; private set; }

public int Height { get; set; }

public int Width { get; set; }

// Virtual method

public virtual void Draw()

{

Console.WriteLine("Performing base class drawing tasks");

}

}

class Circle : Shape

{

public override void Draw()

{

// Code to draw a circle...

Console.WriteLine("Drawing a circle");

base.Draw();

}

}

class Rectangle : Shape

{

public override void Draw()

{

// Code to draw a rectangle...

Console.WriteLine("Drawing a rectangle");

base.Draw();

}

}

class Triangle : Shape

{

public override void Draw()

{

// Code to draw a triangle...

Console.WriteLine("Drawing a triangle");

base.Draw();

}

}

class Program

{

static void Main(string[] args)

{

// Polymorphism at work #1: a Rectangle, Triangle and Circle

// can all be used whereever a Shape is expected. No cast is

// required because an implicit conversion exists from a derived

// class to its base class.

System.Collections.Generic.List<Shape> shapes = new System.Collections.Generic.List<Shape>();

shapes.Add(new Rectangle());

shapes.Add(new Triangle());

shapes.Add(new Circle());

// Polymorphism at work #2: the virtual method Draw is

// invoked on each of the derived classes, not the base class.

foreach (Shape s in shapes)

{

s.Draw();

}

// Keep the console open in debug mode.

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

Console.ReadKey();

}

}

/\* Output:

Drawing a rectangle

Performing base class drawing tasks

Drawing a triangle

Performing base class drawing tasks

Drawing a circle

Performing base class drawing tasks

\*/

在 C# 中，每个类型都是多态的，因为包括用户定义类型在内的所有类型都继承自 [Object](https://msdn.microsoft.com/zh-cn/library/system.object.aspx)。

## 多态性概述

### 虚成员

当派生类从基类继承时，它会获得基类的所有方法、字段、属性和事件。派生类的设计器可以选择是否

* 重写基类中的虚拟成员。
* 继承最接近的基类方法而不重写它
* 定义隐藏基类实现的成员的新非虚实现

仅当基类成员声明为 [virtual](https://msdn.microsoft.com/zh-cn/library/9fkccyh4.aspx) 或 [abstract](https://msdn.microsoft.com/zh-cn/library/sf985hc5.aspx) 时，派生类才能重写基类成员。派生成员必须使用 [override](https://msdn.microsoft.com/zh-cn/library/ebca9ah3.aspx) 关键字显式指示该方法将参与虚调用。以下代码提供了一个示例：

public class BaseClass

{

public virtual void DoWork() { }

public virtual int WorkProperty

{

get { return 0; }

}

}

public class DerivedClass : BaseClass

{

public override void DoWork() { }

public override int WorkProperty

{

get { return 0; }

}

}

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。当派生类重写某个虚拟成员时，即使该派生类的实例被当作基类的实例访问，也会调用该成员。以下代码提供了一个示例：

DerivedClass B = new DerivedClass();

B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;

A.DoWork(); // Also calls the new method.

虚方法和属性允许派生类扩展基类，而无需使用方法的基类实现。有关详细信息，请参阅[使用 Override 和 New 关键字进行版本控制（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/6fawty39.aspx)。接口提供另一种方式来定义将实现留给派生类的方法或方法集。有关详细信息，请参阅[接口（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/ms173156.aspx)。

### 使用新成员隐藏基类成员

如果希望派生成员具有与基类中的成员相同的名称，但又不希望派生成员参与虚调用，则可以使用 [new](https://msdn.microsoft.com/zh-cn/library/51y09td4.aspx) 关键字。 **new** 关键字放置在要替换的类成员的返回类型之前。以下代码提供了一个示例：

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。例如：

DerivedClass B = new DerivedClass();

B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;

A.DoWork(); // Calls the old method.

### 阻止派生类重写虚拟成员

无论在虚拟成员和最初声明虚拟成员的类之间已声明了多少个类，虚拟成员永远都是虚拟的。如果类 A 声明了一个虚拟成员，类 B 从 A 派生，类 C 从类 B 派生，则类 C 继承该虚拟成员，并且可以选择重写它，而不管类 B 是否为该成员声明了重写。以下代码提供了一个示例：

public class A

{

public virtual void DoWork() { }

}

public class B : A

{

public override void DoWork() { }

}

派生类可以通过将重写声明为 [sealed](https://msdn.microsoft.com/zh-cn/library/88c54tsw.aspx) 来停止虚拟继承。这需要在类成员声明中的 **override** 关键字前面放置 **sealed** 关键字。以下代码提供了一个示例：

public class C : B

{

public sealed override void DoWork() { }

}

在上一个示例中，方法 DoWork 对从 C 派生的任何类都不再是虚拟方法。即使它们转换为类型 B 或类型 A，它对于 C 的实例仍然是虚拟的。通过使用 **new** 关键字，密封的方法可以由派生类替换，如下面的示例所示：

public class D : C

{

public new void DoWork() { }

}

在此情况下，如果在 D 中使用类型为 D 的变量调用 DoWork，被调用的将是新的 DoWork。如果使用类型为 C、B 或 A 的变量访问 D 的实例，对 DoWork 的调用将遵循虚拟继承的规则，即把这些调用传送到类 C 的 DoWork 实现。

### 从派生类访问基类虚拟成员

已替换或重写某个方法或属性的派生类仍然可以使用基关键字访问基类的该方法或属性。以下代码提供了一个示例：

public class Base

{

public virtual void DoWork() {/\*...\*/ }

}

public class Derived : Base

{

public override void DoWork()

{

//Perform Derived's work here

//...

// Call DoWork on base class

base.DoWork();

}

}

有关详细信息，请参阅 [base](https://msdn.microsoft.com/zh-cn/library/hfw7t1ce.aspx)。

| **https://wizardforcel.gitbooks.io/msdn-csharp/content/img/note.jpg 注意** |
| --- |
| 建议虚拟成员在它们自己的实现中使用 **base** 来调用该成员的基类实现。允许基类行为发生使得派生类能够集中精力实现特定于派生类的行为。未调用基类实现时，由派生类负责使它们的行为与基类的行为兼容。 |

# 微软官方教程

* [使用 Override 和 New 关键字进行版本控制（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/6fawty39.aspx)
* [了解何时使用 Override 和 New 关键字（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/ms173153.aspx)
* [如何：重写 ToString 方法（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/ms173154.aspx)