OO Programming Modifiers (C#) (msdn.microsoft.com)

Modifier	Purpose
Access Modifiers • public • private • internal • protected	Specifies the declared accessibility of types and type members.
abstract	Indicates that a class is intended only to be a base class of other classes.
async	Indicates that the modified method, lambda expression, or anonymous method is asynchronous.
const	Specifies that the value of the field or the local variable cannot be modified.
event	Declares an event.
extern	Indicates that the method is implemented externally.
new	Explicitly hides a member inherited from a base class.
override	Provides a new implementation of a virtual member inherited from a base class.
partial	Defines partial classes, structs and methods throughout the same assembly.
readonly	Declares a field that can only be assigned values as part of the declaration or in a constructor in the same class.
sealed	Specifies that a class cannot be inherited.
static	Declares a member that belongs to the type itself instead of to a specific object.

unsafe	Declares an unsafe context.
virtual	Declares a method or an accessor whose implementation can be changed by an overriding member in a derived class.
volatile	Indicates that a field can be modified in the program by something such as the operating system, the hardware, or a concurrently executing thread.

abstract (C# Reference)

The **abstract** modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the **abstract** modifier in a class declaration to indicate that a class is intended only to be a base class of other classes. Members marked as abstract, or included in an abstract class, **must** be implemented by non-abstract classes that derive from the abstract class, thereby overriding those abstract members.

An ***abstract*** method is a virtual method with no implementation. An abstract method is declared with the `abstract` modifier and is permitted only in a class that is also declared `abstract`. An abstract method must be overridden in every non-abstract derived class.

Abstract classes have the following features:

- An abstract class cannot be instantiated.
- An abstract class **may** contain abstract methods and accessors.
- It is not possible to modify an abstract class with the sealed (C# Reference) modifier because the two modifiers have opposite meanings. The **sealed** modifier prevents a class from being inherited and the **abstract** modifier requires a class to be inherited.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors

Example:

```
using System;
namespace AbstractClass1
{
    abstract class A
    {
        public abstract void F();
    }
    abstract class B : A
    {
        public void G() { }
    }
    class C : B
    {
        public override void F()
        {
            // actual implementation of F
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!!!");
            Console.ReadKey();
        }
    }
}
```

Not using "override" keyword produces an error:

```
    class C : B
    {
        public new void F()
        {
            // actual implementation of F
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!!!");
            Console.ReadKey();
        }
    }
}
```

100 %

Error List

1 Error | 0 Warnings | 0 Messages

Description

1 'AbstractClass1.C' does not implement inherited abstract member 'AbstractClass1.A.F()'

virtual (C# Reference)

The **virtual** keyword is used to modify a method, property, indexer, or event declaration and **allow** for it to be overridden in a derived class.

The implementation of a virtual member **can** be changed by an overriding member in a derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

By default, methods are non-virtual. You **cannot override** a non-virtual method.

You **cannot** use the virtual modifier with the static, abstract, private, or override modifiers. (Why?...)

override (C# Reference)

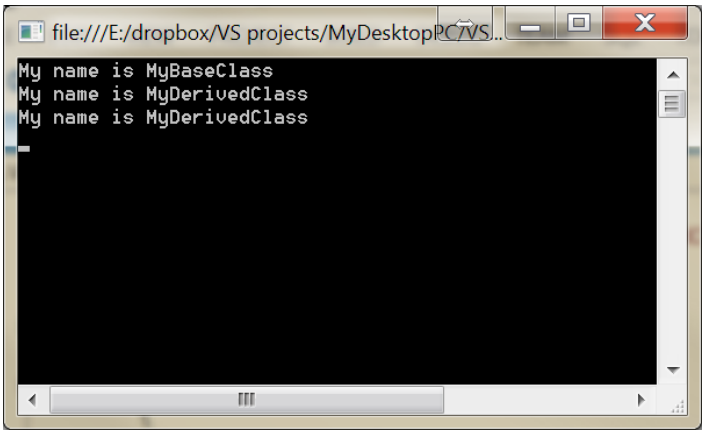
The **override** modifier is required to extend or modify the **abstract or virtual** implementation of an inherited method, property, indexer, or event.

An **override** method provides a new implementation of a member that is inherited from a base class. The method that is overridden by an **override** declaration is known as the overridden base method. The overridden base method must have the same signature as the **override** method.

You cannot override a non-virtual or static method. The overridden base method must be **virtual**, **abstract**, or **override**.
An **override** declaration cannot change the accessibility of the **virtual** method. Both the **override** method and the **virtual** method must have the same access level modifier.

Example:

```
using System;
namespace VirtualMethod1
{
    class MyBaseClass
    {
        public virtual void printname()
        {
            Console.WriteLine("My name is MyBaseClass");
        }
    }
    class MyDerivedClass : MyBaseClass
    {
        public override void printname()
        {
            Console.WriteLine("My name is MyDerivedClass");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyBaseClass obj1 = new MyBaseClass();
            MyDerivedClass obj2 = new MyDerivedClass();
            MyBaseClass obj3 = new MyDerivedClass();
            //MyDerivedClass obj4 = new MyBaseClass(); Why is this an error?...
            obj1.printname();obj2.printname();obj3.printname();
            Console.ReadLine();
        }
    }
}
```



new Modifier (C# Reference)

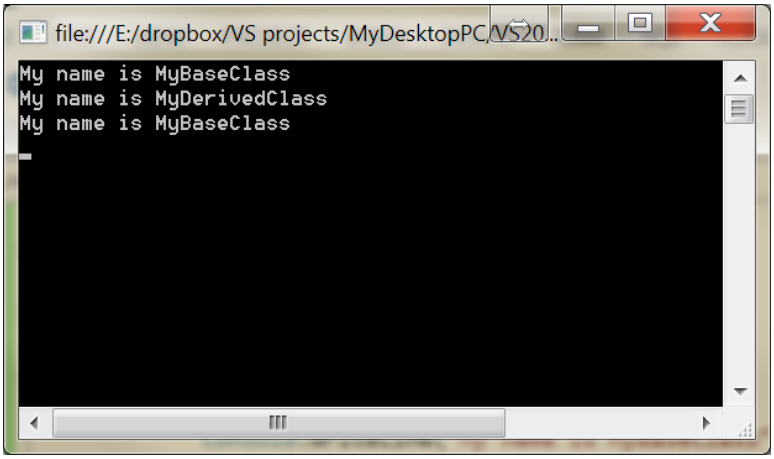
When used as a declaration modifier, the **new** keyword explicitly hides a member that is inherited from a base class. When you hide an inherited member, the derived version of the member replaces the base class version. Although you can hide members without using the **new** modifier, you get a compiler warning. If you use **new** to explicitly hide a member, it suppresses this warning.

To hide an inherited member, declare it in the derived class by using the same member name, and modify it with the **new** keyword.

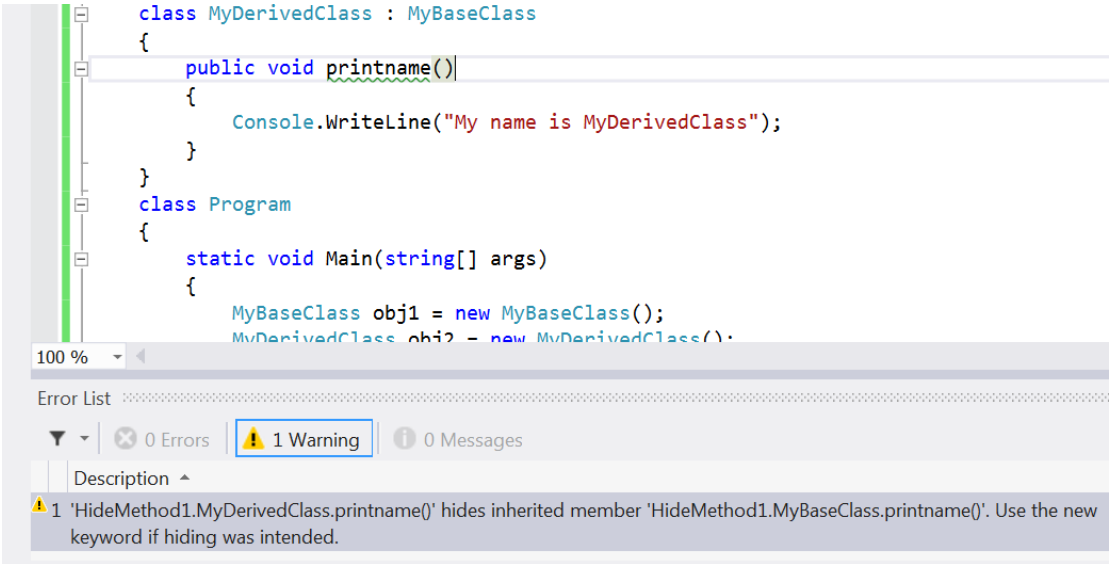
It is an error to use both new and override on the same member, because the two modifiers have mutually exclusive meanings. The new modifier creates a new member with the same name and causes the original member to become hidden. The override modifier extends the implementation for an inherited member.

Example:

```
using System;
namespace HideMethod1
{
    class MyBaseClass
    {
        public void printname()
        {
            Console.WriteLine("My name is MyBaseClass");
        }
    }
    class MyDerivedClass : MyBaseClass
    {
        public new void printname()
        {
            Console.WriteLine("My name is MyDerivedClass");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            MyBaseClass obj1 = new MyBaseClass();
            MyDerivedClass obj2 = new MyDerivedClass();
            MyBaseClass obj3 = new MyDerivedClass();
            obj1.printname();obj2.printname();obj3.printname();
            Console.ReadLine();
        }
    }
}
```



See the warning:



sealed (C# Reference)

When applied to a class, the **sealed** modifier prevents other classes from inheriting from it.

It is an error to use the abstract modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

You can also use the sealed modifier on a method or property **that overrides** a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.

Versioning with the Override and New Keywords (C# Programming Guide)

The C# language is designed so that versioning between base and derived classes in different libraries can evolve and maintain backward compatibility. This means, for example, that the introduction of a new member in a baseclass with the same name as a member in a derived class is completely supported by C# and does not lead to unexpected behavior. It also means that a class must explicitly state whether a method is intended to override an inherited method, or whether a method is a new method that hides a similarly named inherited method.

In C#, derived classes can contain methods with the same name as base class methods.

- The base class method must be defined virtual.
- If the method in the derived class is not preceded by new or override keywords, the compiler will issue a warning and the method will behave as if the **new** keyword were present.
- If the method in the derived class is preceded with the **new** keyword, the method is defined as being independent of the method in the base class.
- If the method in the derived class is preceded with the **override** keyword, objects of the derived class will call that method instead of the base class method.
- The base class method can be called from within the derived class using the **base** keyword.
- The **override**, **virtual**, and **new** keywords can also be applied to properties, indexers, and events.

By default, C# methods are not virtual. If a method is declared as virtual, any class inheriting the method can implement its own version. To make a method virtual, the **virtual** modifier is used in the method declaration of the base class. The derived class can then override the base virtual method by using the **override** keyword or hide the virtual method in the base class by using the **new** keyword. If neither the **override** keyword nor the **new** keyword is specified, the compiler will issue a warning and the method in the derived class will hide the method in the base class.