<http://www.codeproject.com/Articles/6118/All-about-abstract-classes>

**Override and new**

**Parent child combination**

1 : Child=new Base => **New** : Base Function || **Override** : Child function

2: Child=new Child => New : Child Function || Override : Child function

3: Base =new Base => New : Base Function || Override : Base function

4: Base =new Child => Upcasting not allowed

**Abstract class vs. Interface**

An abstract class can have abstract members as well non abstract members. But in an interface all the members are implicitly abstract and all the members of the interface must override to its derived class.

Defining an abstract class with abstract members has the same effect to defining an interface.

The members of the interface are public with no implementation. Abstract classes can have protected parts, static methods, etc.

A class can inherit one or more interfaces, but only one abstract class.

Abstract classes can add more functionality without destroying the child classes that were using the old version. In an interface, creation of additional functions will have an effect on its child classes, due to the necessary implementation of interface methods to classes.

The selection of interface or abstract class depends on the need and design of your project. You can make an abstract class, interface or combination of both depending on your needs.

Interface is programming structure where you define your functions/services that you want to expose to public or other modules. Kind of a contract where you promise that you are providing some functionalities or services, but hiding the implementation so that implementation can be changed without affecting your contract.

A good way to distinguish between a case for the one or the other for me has always been the following:  
  
1. Are there many classes that can be "grouped together" and described by one ***noun***? If so, have an abstract class by the name of this noun, and inherit the classes from it. (A key decider is that these classes share functionality, and you would never instantiate just an **Animal**... you would always instantiate a certain kind of **Animal**: an implementation of your **Animal**base class)  
*Example*: **Cat**and **Dog** can both inherit from abstract class **Animal**, and this abstract base class will implement a method **void Breathe()**which all animals will thus do in exactly the same fashion. (I might make this method virtual so that I can override it for certain animals, like **Fish**, which does not breath the same as most animals).  
  
2. What kinds of ***verbs***can be applied to my class, that might in general also be applied to others? Create an interface for each of these verbs.  
*Example*: All animals can be fed, so I will create an interface called **IFeedable**and have Animal implement that. Only **Dog**and **Horse**are nice enough though to implement **ILikeable** - I will not implement this on the base class, since this does not apply to **Cat**.  
  
By creating an **interface**, you can **move your implementation** to any class that implements your interface.  
By creating an **abstract** class, you can **share implementation** for all derived classes in one central place, and avoid lots of bad things like code duplication.

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.

|  |  |
| --- | --- |
| class BaseClass  {  public void Method1()  {  Console.WriteLine("Base - Method1");  }  }  class DerivedClass : BaseClass  {  public void Method2()  {  Console.WriteLine("Derived-Method2");  }  }  class Program  {  static void Main(string[] args)  {  BaseClass bc = new BaseClass();  DerivedClass dc = new DerivedClass();  BaseClass bcdc = new DerivedClass();  bc.Method1();  dc.Method1();  dc.Method2();  bcdc.Method1();  }  // Output:  // Base - Method1  // Base - Method1  // Derived - Method2  // Base - Method1  } | class BaseClass  {  public void Method1()  {  Console.WriteLine("Base - Method1");  }  public void Method2()  {  Console.WriteLine("Base - Method2");  }  }  class DerivedClass : BaseClass  {  Public new void Method2()  {  Console.WriteLine("Derived-Method2");  }  }  class Program  {  static void Main(string[] args)  {  BaseClass bc = new BaseClass();  DerivedClass dc = new DerivedClass();  BaseClass bcdc = new DerivedClass();  bc.Method1();  bc.Method2();  dc.Method1();  dc.Method2();  bcdc.Method1();  bcdc.Method2();    }  // Output:  // Base - Method1  // Base - Method2  // Base - Method1  // Derived - Method2  // Base - Method1  // Base - Method2  } |
| class BaseClass  {  Public virtual void Method1()  {  Console.WriteLine("Base - Method1");  }  public void Method2()  {  Console.WriteLine("Base - Method2");  }  }  class DerivedClass : BaseClass  {  public new void Method2()  {  Console.WriteLine("Derived-Method2");  }  public override void Method1()  {  Console.WriteLine("Derived - Method1");  }  }  class Program  {  static void Main(string[] args)  {  BaseClass bc = new BaseClass();  DerivedClass dc = new DerivedClass();  BaseClass bcdc = new DerivedClass();  bc.Method1();  bc.Method2();  dc.Method1();  dc.Method2();  bcdc.Method1();  bcdc.Method2();    }  // Output:  // Base - Method1  // Base - Method2  // Derived - Method1  // Derived - Method2  // Derived - Method1  // Base - Method2 |  |
|  |  |
|  |  |

using System;

using System.Text;

namespace OverrideAndNew

{

class Program

{

static void Main(string[] args)

{

BaseClass bc = new BaseClass();

DerivedClass dc = new DerivedClass();

BaseClass bcdc = new DerivedClass();

// The following two calls do what you would expect. They call

// the methods that are defined in BaseClass.

bc.Method1();

bc.Method2();

// Output:

// Base - Method1

// Base - Method2

// The following two calls do what you would expect. They call

// the methods that are defined in DerivedClass.

dc.Method1();

dc.Method2();

// Output:

// Derived - Method1

// Derived - Method2

// The following two calls produce different results, depending

// on whether override (Method1) or new (Method2) is used.

bcdc.Method1();

bcdc.Method2();

// Output:

// Derived - Method1

// Base - Method2

}

}

class BaseClass

{

public virtual void Method1()

{

Console.WriteLine("Base - Method1");

}

public virtual void Method2()

{

Console.WriteLine("Base - Method2");

}

}

class DerivedClass : BaseClass

{

public override void Method1()

{

Console.WriteLine("Derived - Method1");

}

public new void Method2()

{

Console.WriteLine("Derived - Method2");

}

}

}

class Person

{

public string Name { get; set; }

public int Age { get; set; }

public override string ToString()

{

return "Person: " + Name + " " + Age;

}

}

You can test the ToString method as shown in the following code example:

C#Copy

Person person = new Person { Name = "John", Age = 12 };

Console.WriteLine(person);

// Output:

// Person: John 12

The [abstract](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract) keyword enables you to create classes and [class](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/class) members that are incomplete and must be implemented in a derived class.

The [sealed](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed) keyword enables you to prevent the inheritance of a class or certain class members that were previously marked [virtual](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual).

## Abstract Classes and Class Members

Classes can be declared as abstract by putting the keyword abstract before the class definition. For example:

C#Copy

public abstract class A

{

// Class members here.

}

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

Abstract classes may also define abstract methods. This is accomplished by adding the keyword abstract before the return type of the method. For example:

C#Copy

public abstract class A

{

public abstract void DoWork(int i);

}

Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods. When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method. For example:

C#Copy

// compile with: -target:library

public class D

{

public virtual void DoWork(int i)

{

// Original implementation.

}

}

public abstract class E : D

{

public abstract override void DoWork(int i);

}

public class F : E

{

public override void DoWork(int i)

{

// New implementation.

}

}

If a virtual method is declared abstract, it is still virtual to any class inheriting from the abstract class. A class inheriting an abstract method cannot access the original implementation of the method—in the previous example, DoWork on class F cannot call DoWork on class D. In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

## Sealed Classes and Class Members

Classes can be declared as [sealed](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed) by putting the keyword sealed before the class definition. For example:

C#Copy

public sealed class D

{

// Class members here.

}

A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes prevent derivation. Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster.

A method, indexer, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the sealed keyword before the [override](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override)keyword in the class member declaration. For example:

C#Copy

public class D : C

{

public sealed override void DoWork() { }

}

A [static](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static) class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, you cannot use the [new](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/new) keyword to create a variable of the class type. Because there is no instance variable, you access the members of a static class by using the class name itself.

The following list provides the main features of a static class:

* Contains only static members.
* Cannot be instantiated.
* Is sealed.
* Cannot contain [Instance Constructors](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/instance-constructors).