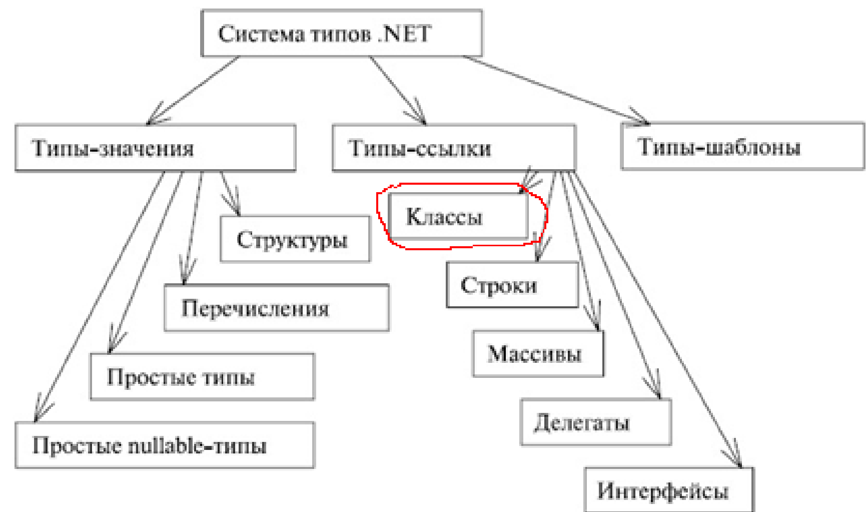
Классы в C#



Класс — это логическая структура, позволяющая создавать свои собственные пользовательские типы.

Класс определяет данные и поведение типа.

Если класс не объявлен **статическим**, то клиентский код может его использовать, создав объекты или экземпляры, назначенные переменной. Переменная остается в памяти, пока все ссылки на нее не выйдут из области видимости. В это время среда **CLR** помечает ее пригодной для сборщика мусора.

Если класс объявляется **статическим**, то в памяти остается только одна копия и клиентский код может получить к ней доступ только посредством самого класса, а не переменной экземпляра

В отличие от структур классы поддерживают **наследование**.

Объявление классов

Классы объявляются с помощью ключевого слова class.

```
public class Customer
{
    //Fields, properties, methods and events go here...
}
```

Ключевому слову class предшествует уровень доступа. Поскольку в данном случае используется public, любой может создавать объекты из этого класса.

Имя класса указывается после ключевого слова class.

Оставшаяся часть определения является телом класса, в котором задаются данные и поведение.

Поля, свойства, методы и события в классе обозначаются термином члены класса.

Создание объектов

Класс и объект — это разные вещи. Класс определяет тип объекта, но не сам объект.

Объект — это конкретная сущность, основанная на классе и иногда называемая экземпляром класса.

Объекты можно создавать с помощью ключевого слова new, за которым следует имя класса, на котором будет основан объект:

```
Customer object1 = new Customer();
```

```
Customer object1 = new Customer();
```

При создании экземпляра класса ссылка на этот объект передается программисту.

В предыдущем примере object1 является ссылкой на объект, основанный на Customer. Эта ссылка указывает на новый объект, но не содержит данные этого объекта. Фактически, можно создать **ссылку на объект** без создания самого объекта:

```
Customer object2;
```

Создание таких ссылок, которые не указывают на объект, **не рекомендуется**, так как попытка доступа к объекту по такой ссылке приведет к сбою во время выполнения. Однако такую ссылку можно сделать указывающей на объект, создав новый объект или назначив ее существующему объекту:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

Инкапсуляция

Об *инкапсуляции* иногда говорят как о первом базовом элементе или принципе объектно-ориентированного программирования. Согласно принципу инкапсуляции класс или структура может задать уровень доступности каждого из членов по отношению к коду вне класса или структуры.

Методы и переменные, которые не предназначены для использования вне класса или сборки, могут быть скрыты, чтобы ограничить потенциальную угрозу возникновения ошибок кода или вредоносное использование

Модификаторы доступа

Все типы и члены типов имеют уровень доступности, который определяет возможность их использования из другого кода в сборке разработчика или других сборках. Можно использовать следующие модификаторы доступа для указания доступности типа или члена при объявлении этого типа или члена.

public

private

protected

internal

protected internal

public

Доступ к типу или члену возможен из любого другого кода в той же сборке или другой сборки, ссылающейся на него.

private

Доступ к типу или члену можно получить только из кода в том же классе или структуре.

protected

Доступ к типу или элементу можно получить только из кода в том же классе или структуре, либо в производном классе.

internal

Доступ к типу или члену возможен из любого кода в той же сборке, но не из другой сборки.

protected internal

Доступ к типу или элементу может осуществляться любым кодом в сборке, в которой он объявлен, или из наследованного класса другой сборки. Доступ из другой сборки должен осуществляться в пределах объявления класса, производного от класса, в котором объявлен защищенный внутренний элемент, и должен происходить через экземпляр типа производного класса.

Пример:

```
public class Animal
{
        public string Name;
        private int Weight;
        protected int Type;
        public int Animal(int W, int T, string N)
        {
                Weight=W;
                Type=T;
                Name=N;
        }
        public int GetWeight()   {return Weight;}
}
```

Наследование классов

Наследование, вместе с инкапсуляцией и полиморфизмом, является одной из трех основных характеристик (или *базовых понятий*) объектно-ориентированного программирования.

Общие свойства и методы наследуются от базового класса, в дополнение к которым добавляются и определяются НОВЫЕ свойства и методы. Таким образом, наследование реализует механизмы расширения базового класса.

Наследование позволяет создавать новые классы, которые повторно используют, расширяют и изменяют поведение, определенное в других классах. Класс, члены которого наследуются, называется ***базовым классом***, а класс, который наследует эти члены, называется ***производным классом***.

Когда класс объявляет базовый тип, он наследует все члены базового класса, **за исключением конструкторов и деструкторов.**

В отличие от C++, класс в C# может *только напрямую* наследовать от одного базового класса.

Однако, поскольку базовый класс может сам наследовать от другого класса, класс может косвенно наследовать несколько базовых классов. Кроме того, класс может напрямую реализовать несколько интерфейсов.

Если **ClassC** является производным от **ClassB**, и **ClassB** является производным от **ClassA**, **ClassC** наследует члены, объявленные в **ClassB** и **ClassA**.

Наследование выполняется с помощью ***образования производных классов***, то есть класс объявляется с помощью ***базового класса***, от которого он наследует данные и поведение. Базовый класс задается добавлением после имени производного класса двоеточия и имени базового класса:

```
public class Manager : Employee
{
  // Employee fields, properties, methods and events are inherited
  // New Manager fields, properties, methods and events go here...
 }
```

Структуры не поддерживают наследование.

Пример реализации принципов наследования:

```
using System;
namespace Inheritance_1
{
   public class A
        { public int val1_A;
          public void fun1_A (String str)
                   { Console.WriteLine("A's fun1_A:" + str); }
         }
    public class B:A
          { public int val1_B;
            public void fun1_B (String str) { Console.WriteLine("B's fun1_B:" + str); }
          }
     class Class1
          {
              static void Main(string[ ] args)
                {  B b0 = new B( );
                   // От имени объекта b0 вызвана собственная функция fun1_B.
                    b0.fun1_B("from B");
                   // От имени объекта b0 вызвана унаследованная от класса A функция fun1_A.
                    b0.fun1_A("from B");
                   }
          }
}
```

Наследование и проблемы доступа

Наследование — это свойство, с помощью которого один объект может приобретать свойства другого. Производный класс наследует от базового класса ВСЕ, что он имеет. Другое дело, что воспользоваться в производном классе можно не всем.

Синтаксис:

```
class имя_класса : имя_родительского_класса
{
        тело_класса
}
```

Пример:

```
class Predator: Animal
{
private int Speed;
}
```

Добавляем в базовый класс **private**-члены:

```
public class A
{
  public int val1_A = 0;
  public void fun1_A (String str)
     { Console.WriteLine("A's fun1_A:" + str);
        this.fun2_A("private function from A:");
      }
    private int val2_A = 0;
    private void fun2_A (String str)
          { Console.WriteLine(str + "A's fun2_A:" + val2_A.ToString()); }
 }
```

Теперь объект класса B в принципе НЕ может получить доступ к **private** данным — членам и функциям — членам класса A.

Косвенное влияние на такие данные-члены и функции — члены – лишь через **public**-функции класса A.

Используем еще один спецификатор доступа – **protected**. Этот спецификатор обеспечивает открытый доступ к членам базового класса, но только для производного класса!

```
public class A
 {   :::::::::
     protected int val3_A = 0;
 }
 public class B:A
 { :::::::::
   public void fun1_B (String str) { ::::::::: this.val3_A = 125; }         // Нормально
}

static void Main(string[ ] args)
{ :::::::::
 // b0.val3_A = 125;  // Это член класса закрыт для внешнего использования!
}
```

Защищенные члены базового класса доступны для ВСЕХ прямых и косвенных наследников данного класса.

Несколько важных замечаний относительно использования спецификаторов доступа:

- в C# структуры НЕ поддерживают наследования. Поэтому спецификатор доступа **protected** в объявлении данных — членов и функций — членов структур НЕ ПРИМЕНЯЕТСЯ;

- спецификаторы доступа действуют и в рамках пространства имен (поэтому и классы в нашем пространстве имен были объявлены со спецификаторами доступа **public**). Но в пространстве имен явным образом можно использовать лишь один спецификатор – спецификатор public, либо не использовать никаких спецификаторов.

Класс может быть объявлен **абстрактным**.

Абстрактный класс содержит абстрактные методы, которые имеют определение сигнатуры, но не имеют реализации. Создавать объекты абстрактного класса с помощью оператора **new** нельзя.

Абстрактные классы могут использоваться только посредством производных классов, реализующих абстрактные методы.

Когда базовый класс объявляет метод как **виртуальный**, производный класс может переопределить метод с помощью своей собственной реализации.

Абстрактные и **виртуальные** члены являются основой для **полиморфизма**, который является третьей основной характеристикой объектно-ориентированного программирования

Если в базовом классе будет создан новый член, имя которого совпадает с именем члена в производном классе, в классе должно быть явно указано, будет ли метод переопределять наследуемый метод, или это новый метод, который будет скрывать наследуемый метод с тем же именем.

В C# производный класс может включать методы с теми же именами, что и у методов базового класса.

- Метод базового класса может быть определен как **виртуальный**.
- Если перед методом в производном классе не указано ключевое слово **new** или **override**, компилятор выдаст предупреждение, и обработка метода будет производиться как в случае наличия ключевого слова **new**.
- Если перед методом в производном классе указано ключевое слово **new**, то этот метод определен как независимый от метода в базовом классе.
- Если перед методом в производном классе указано ключевое слово **override**, то объекты производного класса будут вызывать этот метод вместо метода базового класса.
- Базовый метод можно вызвать из производного класса с помощью ключевого слова **base**.
- Ключевые слова **override**, **virtual** и **new** могут также применяться к свойствам, индексам и событиям.

По умолчанию методы в языке C# не являются виртуальными.

Если метод объявлен как **виртуальный**, то любой класс, наследующий этот метод, может реализовать собственную версию. Чтобы сделать метод виртуальным, в объявлении метода базового класса используется модификатор **virtual**.

Производный класс может переопределить базовый виртуальный метод с помощью ключевого слова **override** или скрыть виртуальный метод в базовом классе с помощью ключевого слова **new**. Если ключевые слова **override** и **new** не указаны, компилятор выдаст предупреждение, и метод в производном классе будет скрывать метод в базовом классе.

Чтобы продемонстрировать это на практике, предположим, что компания A создала класс с именем **GraphicsClass**, используемый вашей программой.

```
class GraphicsClass
{
  public virtual void DrawLine() { }
  public virtual void DrawPoint() { }
}
```

Вы применили его для создания собственного производного класса, добавив новый метод:

```
class YourDerivedGraphicsClass : GraphicsClass
{
  public void DrawRectangle() { }
}
```

Ваше приложение работало без проблем до тех пор, пока компания A не выпустила новую версию класса GraphicsClass со следующим кодом:

```
class GraphicsClass
{
  public virtual void DrawLine() { }
  public virtual void DrawPoint() { }
  public virtual void DrawRectangle() { }
}
```

После перекомпиляции приложения с помощью новой версии класса GraphicsClass компилятор выдаст предупреждение **CS0108**. В этом предупреждении вам будет предложено принять решение относительно работы метода DrawRectangle в вашем приложении.

Чтобы избежать путаницы между двумя методами, вы можете переименовать свой метод. Это отнимает много времени и может привести к возникновению ошибок, поэтому не всегда удобно. Если ваш метод должен переопределить новый метод базового класса, используйте ключевое слово **override**:

```
class YourDerivedGraphicsClass : GraphicsClass
{
   public override void DrawRectangle() { }
}
```

Объекты, являющиеся производными от класса YourDerivedGraphicsClass, могут продолжать использовать версию метода DrawRectangle базового класса, используя ключевое слово "base":

```
base.DrawRectangle();
```

Чтобы избежать появления предупреждения, можно также использовать ключевое слово **new** в определении производного класса.

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

Полиморфизм

Полиморфизм – одна из основных составляющих объектно-ориентированного программирования, позволяющая определять в наследуемом классе методы, которые будут общими для всех наследующих классов, при этом наследующий класс может определять специфическую реализацию некоторых или всех этих методов. Главный принцип полиморфизма: «один интерфейс, несколько методов». Благодаря ему, можно пользоваться методами, не обладая точными знаниями о типе объектов.

Основным инструментом для реализации принципа полиморфизма является использование виртуальных методы и абстрактных классов.

О полиморфизме часто говорят как о третьем базовом элементе объектно-ориентированного программирования, после инкапсуляции и наследования класса.

Полиморфизм — это греческое слово, означающее "наличие многих форм". Это понятие имеет два различающихся аспекта.

1. Во время выполнения объекты производного класса могут рассматриваться как объекты базового класса в таких местах как параметры метода и коллекции массивов. При этом объявленный тип объекта больше не идентичен его типу времени выполнения.

2. Базовые классы могут определять и реализовывать виртуальные методы, а производные классы могут переопределять их. Это означает, что они предоставляют свои собственные определения и реализацию. Во время выполнения, когда клиентский код вызывает метод, среда **CLR** ищет тип времени выполнения объекта и вызывает это переопределение виртуального метода. Таким образом, в исходном коде можно вызвать метод в базовом классе и вызвать выполнение метода с версией производного класса.

Виртуальные методы

Метод, при определении которого в наследуемом классе было указано ключевое слово virtual, и который был переопределен в одном или более наследующих классах, называется виртуальным методом. Следовательно, каждый наследующий класс может иметь собственную версию виртуального метода.

Выбор версии виртуального метода, которую требуется вызвать, осуществляется в соответствии с типом объекта, на который ссылается ссылочная переменная, во время выполнения программы. Другими словами, именно тип объекта, на который указывает ссылка (а не тип ссылочной переменной), определяет вызываемую версию виртуального метода. Таким образом, если класс содержит виртуальный метод и от этого класса были наследованы другие классы, в которых определены свои версии метода, при ссылке переменной типа наследуемого класса на различные типы объектов вызываются различные версии виртуального метода.

При определении виртуального метода в составе наследуемого класса перед типом возвращаемого значения указывается ключевое слово virtual, а при переопределении виртуального метода в наследующем классе используется модификатор override. Виртуальный метод не может быть определен с модификатором static или abstract.

Переопределять виртуальный метод не обязательно. Если наследующий класс не предоставляет собственную версию виртуального метода, то используется метод наследуемого класса.

Переопределение метода положено в основу концепции динамического выбора вызываемого метода - выбора вызываемого переопределенного метода осуществляется во время выполнения программы, а не во время компиляции.

Синтаксис:

virtual тип имя (список_параметров) {тело_метода};

Абстрактные классы

В абстрактном классе определяются лишь общие предназначения методов, которые должны быть реализованы в наследующих классах, но сам по себе этот класс не реализует один, или несколько подобных методов, называемых абстрактными (для них определены только некоторые характеристики, такие как тип возвращаемого значения, имя и список параметров).

При объявлении абстрактного метода используется модификатор abstract.

Абстрактный метод автоматически становится виртуальным, так что модификатор virtual при объявлении метода не используется.

Абстрактный класс предназначен только для создания иерархии классов, нельзя создать объект абстрактного класса.

Пример:

```
abstract class Animal
{
        public string Name;
        protected int Weight;
        private int Type;
        abstract void Feed();
        public int Animal(int W, int T, string N)
        {
                Weight=W;
                Type=T;
                Name=N;
        }
        public int GetWeight()
        {
                return Weight;
        }
}
```

```
class Predator:Animal
{
        private int Speed;
        override void Feed(int Food)
        {
                Weight += Food;
        }
}
```

Пусть имеется приложение рисования, которое дает возможность пользователю создавать на поверхности рисования различные формы.

Во время компиляции неизвестно, какие конкретные формы будет создавать пользователь. Но приложение должно учитывать все различные типы создаваемых форм, и оно должно обновлять их в ответ на действия пользователя.

1. Создадим базовый класс, называемый **Shape**, и производные классы, например **Rectangle**, **Circle** и **Triangle**.
2. Определим в классе **Shape** виртуальный метод, называемый **Draw**, и переопределим его в каждом производном классе для рисования конкретной формы, которую представляет класс.
3. Создадим объект **List<Shape>** и добавим в него круг, треугольник и прямоугольник.
4. Для обновления поверхности рисования используем цикл **foreach** для итерации списка и вызова метода **Draw** на каждом объекте **Shape** в списке. Хотя каждый объект в списке имеет объявленный тип **Shape**, будет вызываться именно тип времени выполнения (переопределенная версия метода в каждом производном классе).

```
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
 */
```

Перегрузка родительских методов

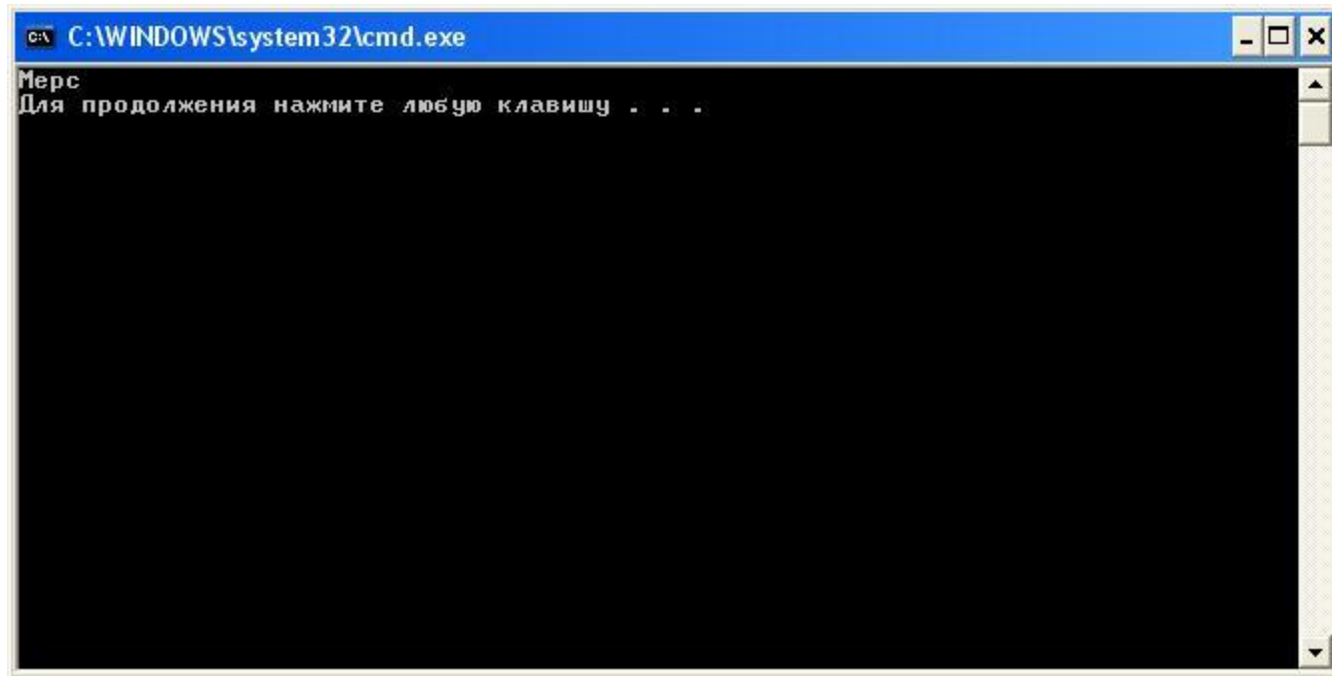
При использовании объектно-ориентированного программирования мы явно или не явно используем наследование. Например при создание любого, объект этого класса будет наследоваться от object. И чаще всего необходимо перегрузить некоторые функции родительского класса.

Перезагрузим метод ToString пользовательского класса. Происходит это с использованием ключевого слова overide

```
namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            car mers = new car("Mepc");
            Console.WriteLine(mers.ToString());
        }

    }
    class car
    {
        string _name;
        public car(string name)
        {
            this._name = name;
        }
        public override string ToString()
        {
            return this._name;
        }
    }
}
```

Теперь после того как мы перегрузили метод ToString, наш экземпляр при вызове данной функции будет возвращать не имя класса как это была в базовой версии ToString( ), а имя машины.



Прекращение наследования. sealed-спецификатор

Принцип наследования допускает неограниченную глубину иерархии наследования. Производный класс, являющийся наследником базового класса, может в свою очередь сам оказаться в роли базового класса. Однако не всегда продолжение цепочки "предок-потомок" может оказаться целесообразным.

Если при разработке класса возникла ситуация, при которой дальнейшее совершенствование и переопределение возможностей класса в деле решения специфических задач окажется нежелательным (сколько можно заниматься переопределением функций форматирования), класс может быть закрыт для дальнейшего наследования. Закрытие класса обеспечивается спецификатором **sealed**.

При этом закрываться для наследования может как класс целиком, так и отдельные его члены.

Запечатанный класс не позволяет другим классам быть от него производными. При применении к классу, модификатор **sealed** запрещает другим классам наследовать от этого класса.

В приведенном примере класс B наследует от класса A, но никакие классы не могут наследовать от класса B.

```
class A {}
sealed class B : A {}
```

Модификатор **sealed** можно использовать для метода или свойства, которое переопределяет виртуальный метод или свойство в базовом классе.

Это позволяет классам наследовать от вашего класса, запрещая им при этом переопределять определенные виртуальные методы или свойства.

В следующем примере класс Z наследуется от класса Y, но Z не может переопределить виртуальную функцию F, которая объявлена в классе X и "запечатана" в классе Y.

```
class X
 {
    protected virtual void F()   { Console.WriteLine("X.F");   }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
  }
class Y : X
{
   sealed protected override void F()   { Console.WriteLine("Y.F");  }
             protected override void F2() { Console.WriteLine("X.F3"); }
 }
class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("C.F"); }
    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

Производный класс может прекратить наследование, объявив переопределение как **sealed**.

```
public class A
{
   public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}

public class C : B
{
      public sealed override void DoWork() { }
 }
```

Метод DoWork больше не является виртуальным для любого класса, производного от класса C.

Запечатанные методы могут быть заменены производными классами с помощью ключевого слова **new**

```
public class D : C
{
  public new void DoWork() { }
}
```

Синтаксис объявления абстрактной функции предполагает использование ключевого слова ***abstract*** и полное отсутствие тела. Объявление абстрактной функции завершается точкой с запятой.

Класс, содержащий вхождения ***абстрактных*** (хотя бы одной) функций, также должен содержать в заголовке объявления спецификатор **abstract**.

В производном классе соответствующая переопределяемая абстрактная функция обязательно должна включать в заголовок функции спецификатор **override**. Его назначение – явное указание факта переопределения абстрактной функции.

Абстрактный класс фактически содержит объявления нереализованных функций базового класса. На основе абстрактного класса невозможно определить объекты. Попытка создания соответствующего объекта — представителя абстрактного класса приводит к ошибке, поскольку в классе не определены алгоритмы, определяющие поведение объекта

```
abstract class ShapesClass
{
   abstract public int Area();
}

class Square : ShapesClass
{
  int side = 0;
  public Square(int n) { side = n; }
  public override int Area() { return side * side; }

  static void Main()
   {
     Square sq = new Square(12);
     Console.WriteLine("Area of the square = {0}", sq.Area());
    }
}
// Output: Area of the square = 144
```

Чтобы предотвратить переопределение производных классов при определении новых методов или свойств, не назначайте их в качестве виртуальных (**virtual**).

Нельзя использовать модификатор **abstract** с **sealed**-классом, поскольку абстрактный класс должен наследоваться классом, реализующим абстрактные методы или свойства.

При применении **sealed** модификатора к методу или свойству его необходимо всегда использовать с **override**.

Поскольку структуры неявно запечатаны, их нельзя наследовать.

Полное квалифицированное имя

Если вы подключили в проекте два пространства имен, например ns1 и ns2

```
using ns1;
using ns;
```

которые включают в себя класс Class1, то при попытке создать экземпляр данного класса возникнет ошибка

```
Class1 c = new Class1();
```

Компилятор не знает создавать класс из пространства имен ns1 или ns2 В таком случае надо конкретно указывать пространство имен

```
ns1.Class1 c = new ns1.Class1();
```

В следующем примере и базовый BC, и производный DC - классы используют одно и то же имя для обозначения объявляемого в обоих классах члена типа **int**. **new** - модификатор подчеркивает факт недоступности члена **x** базового класса в производном классе. Однако из производного класса все-таки можно обратиться к переопределенному полю базового класса с использованием ***полного квалифицированного имени***:

```
using System;
public class BC { public static int x = 55; public static int y = 22; }

public class DC : BC
{   new public static int x = 100; // Переопределили член базового класса
    public static void Main( )
      {
          Console.WriteLine(x);       // Доступ к переопределенному члену x:
          Console.WriteLine(BC.x); // Доступ к члену базового класса x:
          Console.WriteLine(y);      // Доступ к члену y:
      }
}
```

То же с классами. В производном классе **DC** переопределяется вложенный класс **C**. В рамках производного класса легко можно создать объект — представитель вложенного переопределенного класса **C**. Для создания аналогичного объекта — представителя базового класса необходимо использовать полное квалифицированное имя:

```
using System;
public class BC { public class C { public int x = 200; public int y; } }
public class DC:BC
 {
   // Вложенный класс базового класса скрывается
   new public class C  { public int x = 100; public int y; public int z; }
   public static void Main()
    { // Из производного класса виден переопределенный вложенный класс:
      C s1 = new C();
      // Полное имя используется для доступа к классу, вложенному в базовый:
      BC.C s2 = new BC.C();
      Console.WriteLine(s1.x);
      Console.WriteLine(s2.x);
    }
}
```

Вложенные классы

Конструкторы
Деструкторы
Константы
Поля
Методы
Свойства
Индексаторы
Операторы
События
Делегаты
Классы
Интерфейсы
Структуры

Иногда некоторый класс играет чисто вспомогательную роль для другого класса и используется только внутри него. В этом случае логично описать его внутри существующего класса. Вот пример такого описания:

```
using System;

namespace test
{
   class ClassA
   {
      private class ClassB  // Вложенный класс _Класс _Класс
      {
         public int z;
      }
      private ClassB w;      // Переменная типа вложенного класса _Класса

      public ClassA()        // Конструктор
      {
         w = new ClassB();
         w.z = 35;
      }
```

```
        public int SomeMethod( ) //Некоторый метод
        {
            return w.z;
        }

        class Test
        {
            static void Main(string[] args)
            {
                ClassA v = new ClassA();
                int k = v.SomeMethod();
                Console.WriteLine(k);
            }
        }
    }
}
```

После запуска программа выведет результат:
35

Здесь класс ClassB объявлен внутри класса ClassA, причем со словом private, так что его экземпляры мы можем создавать только внутри класса ClassA (что мы и делаем в конструкторе класса ClassA). Методы класса ClassA имеют доступ к экземпляру класса ClassB (как, например, метод SomeMethod).

Вложенный класс имеет смысл использовать тогда, когда его экземпляр используется только в определенном классе.

Кроме того, с вложением классов улучшается читаемость кода — если нас не интересует устройство основного класса, то разбирать работу вложенного класса нет необходимости.

Обратите также внимание, как вложенные классы показываются на вкладке ClassView