Иерархия классов

Наследование

Наследование применяется для следующих взаимосвязанных целей:

1) исключения из программы повторяющихся фрагментов кода;

2) упрощения модификации программы;

3) упрощения создания новых программ на основе существующих.

- Наследование является единственной возможностью использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.

- Класс в C# может иметь произвольное количество потомков и только одного предка.

- При описании класса имя его предка записывается в заголовке класса после двоеточия.

- Если имя предка не указано, предком считается базовый класс всей иерархии System.Object.

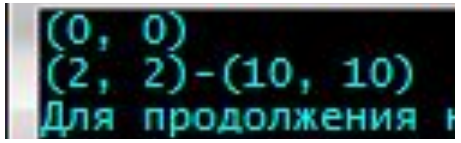
[атрибуты] [спецификаторы] class имя_класса [: предки]
{ тело_класса}

- Класс наряду с единственным предком-классом может наследовать интерфейсы (специальный вид классов, не имеющих реализации).

- Класс, который наследуется, называется ***базовым***.
- Класс, который наследует базовый класс, называется ***производным***.
- Производный класс, наследует все переменные, методы, свойства, операторы и индексаторы, определенные в базовом классе, кроме того в производный класс могут быть добавлены уникальные элементы или переопределены существующие.

// наследование классов на примере геометрических фигур на плоскости.
В качестве базового класса - класс DemoPoint (точка на плоскости),
производный класс от DemoPoint - класс DemoLine (отрезок на плоскости):

```
  class DemoPoint //базовый класс
{     public int x;
      public int y;
      public void Show()    {
        Console.WriteLine("({0}, {1})", x, y);              }
}
class DemoLine : DemoPoint //производный класс
{     public int xEnd;
      public int yEnd;
      public  void Show(){
        Console.WriteLine("({0}, {1})-({2}, {3})", x, y ,xEnd, yEnd);     }
}
class Program{
static void Main(){
DemoPoint point = new DemoPoint();
point.x = 0; point.y = 0; point.Show();
DemoLine line = new DemoLine();
line.x = 2; line.y = 2; line.xEnd = 10; line.yEnd = 10;
line.Show();
}    } }
```



```
class DemoPoint {
    protected int x;
    protected int y;
    public void Show() {
      Console.WriteLine("({0}, {1})",x, y);}
}
class DemoLine : DemoPoint {
    public int xEnd;
    public int yEnd;
    public new void Show() {
      x=2; y=2;  //доступ к закрытым полям базового класса
    Console.WriteLine("({0}, {1})-({2}, {3})", x, y, xEnd, yEnd); }
}

class Program {
static void Main() {
    DemoPoint point = new DemoPoint();
    point.Show();
    DemoLine line = new DemoLine();
    //line.x = 2; line.y = 2; //доступ к полям закрыт
    line.xEnd = 10;line.yEnd = 10;
    line.Show(); }
}
```

Если убрать public, то поля автоматически станут закрытыми для доступа (private), в том числе и для доступа из производного класса.

Доступ к закрытым полям базового класса из производного:

1. используя свойства класса
2. спецификатор protected (при объявлении члена класса с помощью protected, он становится закрытым для всех классов, кроме производных).

доступ к полям x и y из класса Program невозможен, а из производного класса DemoLine возможен.

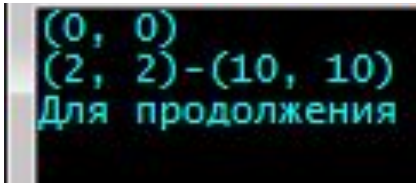
Наследование конструкторов

- В иерархии классов как базовые, так и производные классы могут иметь собственные конструкторы.
- Конструктор базового класса создает часть объекта, соответствующую базовому классу
- Конструктор производного класса — часть объекта, соответствующую производному классу.
- Так как базовый класс не имеет доступа к элементам производного класса, то их конструкторы должны быть раздельными.

// предыдущий пример классы + конструктор только в производный класс DemoLine:

```
class DemoPoint  {
      protected int x;
      protected int y;
public void Show()        {
Console.WriteLine("({0}, {1})",x, y);  }
}

class DemoLine : DemoPoint  {
      public int xEnd;
      public int yEnd;
new public void Show() {
Console.WriteLine("({0}, {1})-({2}, {3})", x, y, xEnd, yEnd);     }

public DemoLine(int x1, int y1, int x2, int y2) //конструктор производного класса
{
      x = x1;    y = y1;
      xEnd = x2; yEnd = y2;       }
}

class Program      {
static void Main() {
DemoPoint point = new DemoPoint(); //вызывается конструктор по умолчанию
point.Show();
DemoLine line = new DemoLine(2, 2, 10, 10); //вызывается собственный конструктор
line.Show();         }
}
```

В данном случае конструктор определяется только в производном классе, поэтому часть объекта, соответствующая базовому классу, создается автоматически с помощью конструктора по умолчанию, а часть объекта, соответствующая производному классу, создается собственным конструктором.



- Если же конструкторы определены и в базовом, и в производном классе, то процесс создания объектов несколько усложняется, т.к. должны ***выполниться*** конструкторы обоих классов.

- В этом случае используется ключевое слово ***base***, которое имеет два назначения:

1) позволяет вызвать конструктор базового класса:

- Производный класс может вызывать конструктор, определенный в его базовом классе, используя расширенную форму объявления конструктора и ключевое слово ***base***.

```
конструктор_производного_класса (список_параметров) : base (список_аргументов)
 {
тело конструктора
}
```

- с помощью элемента списка аргументов передаются параметры конструктору базового класса.

```
class DemoPoint          {
        protected int x;
        protected int y;
        public void Show(){
            Console.WriteLine("({0}, {1})",x, y);          }
        public DemoPoint (int x, int y)              //конструктор базового класса
        {           this.x=x;    this.y=y;    }
    }

class DemoLine : DemoPoint {
        public int xEnd;
        public int yEnd;
        new public void Show()        {
            Console.WriteLine("({0}, {1})-({2}, {3})", x, y, xEnd, yEnd);
        }

        public DemoLine(int x1, int y1, int x2, int y2):base(x1, y1) //конструктор производного класса
        {           xEnd = x2; yEnd = y2;  }
    }

class Program{
        static void Main(){
            DemoPoint point= new DemoPoint(5, 5);          point.Show();
            DemoLine line = new DemoLine( 2, 2, 10, 10); line.Show();
        }
    }
```

```
/*С помощью ключевого слова base можно вызвать конструктор любой формы, определенный в базовом классе, но реально выполнится тот конструктор, параметры которого будут соответствовать переданным при вызове аргументам*/
class DemoPoint          {
  protected int x;                    protected int y;
  public void Show()     {
            Console.WriteLine("({0}, {1})",x, y);     }
  public DemoPoint () //конструктор базового класса по умолчанию
  {                 this.x=1;         this.y=1;         }
  public DemoPoint (int x, int y) //конструктор базового класса с параметрами
  {                 this.x=x;         this.y=y;         }
}
class DemoLine : DemoPoint{
  public int xEnd;          public int yEnd;
  new public void Show(){
            Console.WriteLine("({0}, {1})-({2}, {3})", x, y, xEnd, yEnd);        }
  public DemoLine() //конструктор производного класса по умолчанию
  {                 xEnd = 100; yEnd = 100;       }
  public DemoLine(int x2, int y2) //конструктор производного класса с двумя параметрами
  {                 xEnd = x2; yEnd = y2;         }
  //конструктор производного класса с четырьмя параметрами
  public DemoLine(int x1, int y1, int x2, int y2):base(x1, y1)
  {                 xEnd = x2; yEnd = y2;         }
}
class Program{
  static void Main()       {
            DemoPoint point1= new DemoPoint(); //вызов конструктора по умолчанию
            DemoPoint point2= new DemoPoint(5, 5); //вызов конструктора с параметрами
            point1.Show();          point2.Show();
            DemoLine line1 = new DemoLine(); //вызов конструктора по умолчанию
            DemoLine line2 = new DemoLine(4, 4);       //вызов конструктора с двумя параметрами
            //вызов конструктора с четырьмя параметрами
            DemoLine line3 = new DemoLine(2, 2, 10, 10);      line1.Show();  line2.Show();  line3.Show();
  }
}
```

- ***2) base позволяет получить доступ к члену базового класса, который скрыт "за" членом производного класса.***

В этом случае ключевое слово ***base*** действует подобно ссылке ***this***, за исключением того, что ссылка ***base*** всегда указывает на базовый класс для производного класса, в котором она используется.

base.член_класса

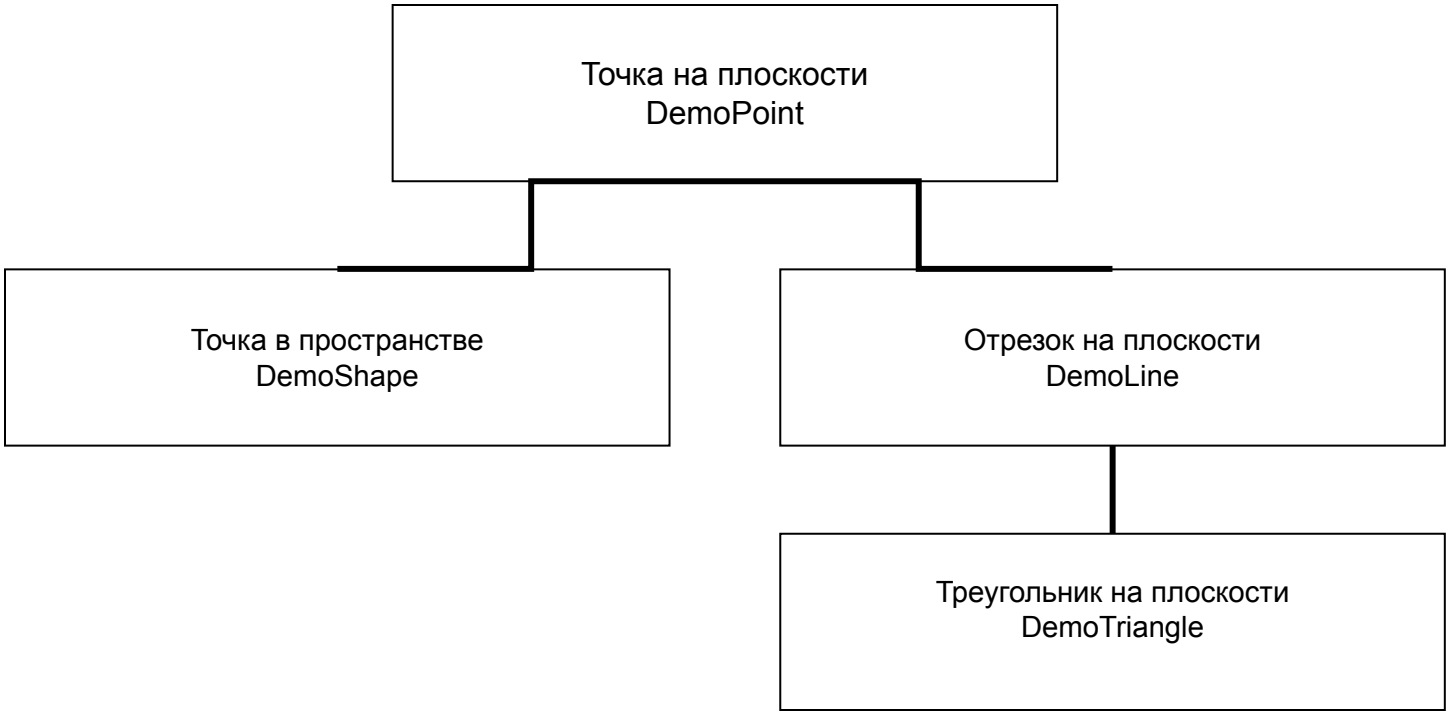
- в качестве элемента *член_класса* можно указывать либо метод, либо поле экземпляра.

- Эта форма ссылки **base** наиболее применима в тех случаях, когда имя члена в производном классе скрывает член с таким же именем в базовом классе.

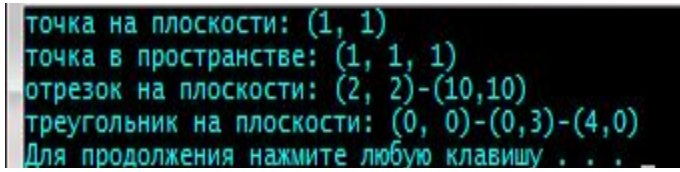
```
class DemoPoint  {
        protected int x;
        protected int y;
        public void Show()       {
                Console.Write("({0}, {1})",x, y);
        }
        public DemoPoint (int x, int y)                //конструктор базового класса
        {               this.x=x;     this.y=y;     }
      }

      class DemoLine : DemoPoint    {
        public int xEnd;
        public int yEnd;
        new public void Show()        {
                base.Show();        //вызов члена базового класса
                Console.WriteLine("-({0}, {1})", xEnd, yEnd);
        }

        public DemoLine(int x1, int y1, int x2, int y2):base(x1, y1) //конструктор производного класса
        {               xEnd = x2; yEnd = y2;  }
      }

      class Program{
        static void Main(){
                DemoLine line = new DemoLine( 2, 2, 10, 10); line.Show();
        }
      }
```

Несмотря на то, что метод Show в классе DemoLine скрывает одноименный метод в классе DemoPoint, ссылка base позволяет получить доступ к методу Show в базовом классе. Аналогично с помощью ссылки base можно получить доступ к одноименным полям базового класса.

Многоуровневая иерархия



```
class DemoPoint    {
    protected int x;       protected int y;
    public void Show()       {          Console.WriteLine("точка на плоскости: ({0}, {1})", x, y);       }
    public DemoPoint(int x, int y)
    {          this.x = x; this.y = y;       }    }
class DemoShape : DemoPoint    {       protected int z;
    new public void Show()       {          Console.WriteLine("точка в пространстве: ({0}, {1}, {2})", x, y, z);       }
    public DemoShape(int x, int y, int z)          : base(x, y)
    {          this.z = z;       }    }
class DemoLine : DemoPoint    {       protected int x2;       protected int y2;
    new public void Show()       {
        Console.WriteLine("отрезок на плоскости: ({0}, {1})-({2},{3})", x, y, x2, y2);       }
    public DemoLine(int x1, int y1, int x2, int y2)          : base(x1, y1)
    {          this.x2 = x2; this.y2 = y2;       }    }
class DemoTriangle : DemoLine    {       protected int x3;       protected int y3;
    new public void Show()       {
        Console.WriteLine("треугольник на плоскости: ({0}, {1})-({2},{3})-({4},{5})", x, y, x2, y2, x3, y3);       }
    public DemoTriangle(int x1, int y1, int x2, int y2, int x3, int y3)          : base(x1, y1, x2, y2)
    {          this.x3 = x3; this.y3 = y3;       }    }
class Program    {
    static void Main()       {
        DemoPoint point = new DemoPoint(1, 1);          point.Show();
        DemoShape pointShape = new DemoShape(1, 1, 1);                pointShape.Show();
        DemoLine line = new DemoLine(2, 2, 10, 10);                line.Show();
        DemoTriangle triangle = new DemoTriangle(0, 0, 0, 3, 4, 0);     triangle.Show();       }
}     }
```



Переменные базового класса и производного класса

- C# является языком со строгой типизацией, в нем требуется строгое соблюдение совместимости типов с учетом стандартных преобразований типов.

- Переменная одного типа обычно ***не может ссылаться*** на объект другого ссылочного типа.

- ***Исключение*** – ссылочная переменная базового класса может ссылаться на объект любого производного класса.

```
class DemoPoint    {
      public int x;       public int y;
      public void Show()       {
         Console.WriteLine("точка на плоскости: ({0}, {1})", x, y);
      }
      public DemoPoint(int x, int y)
      {          this.x = x; this.y = y;       }
   }
   class DemoShape : DemoPoint    {
      public int z;
      new public void Show()       {
         Console.WriteLine("точка в пространстве: ({0}, {1}, {2})", x, y, z);
      public DemoShape(int x, int y, int z)          : base(x, y)
      {          this.z = z;       }
   }
   class Program    {
      static void Main()       {
         DemoPoint point1 = new DemoPoint(0, 1);
         Console.WriteLine("({0}, {1})", point1.x, point1.y);
         DemoShape pointShape = new DemoShape(2, 3, 4);
         Console.WriteLine("({0}, {1}, {2})", pointShape.x, pointShape.y, pointShape.z);
         DemoPoint point2 = pointShape;    //допустимая операция
         //ошибка - не соответствие типов указателей
         //pointShape=point1;
         Console.WriteLine("({0}, {1})", point2.x, point2.y);
         //ошибка, т.к. в классе DemoPoint нет поля z
         //Console.WriteLine("({0}, {1}, {2})", point2.x, point2.y, point2.z);
    }   }
```

Ошибка возникнет и при попытке через объект point2 обратиться к методу Show. Например, point2.Show(). В этом случае компилятор не сможет определить, какой метод Show вызвать – для базового или для производного класса. Для решения данной проблемы можно воспользоваться таким понятием как *полиморфизм*, который основывается на механизме виртуальных методов.

Виртуальные методы

- Виртуальный метод – это метод, который объявлен в базовом классе с использованием ключевого слова ***virtual***, и затем переопределен в производном классе с помощью ключевого слова ***override***.

- При этом если реализована многоуровневая иерархия классов, то каждый производный класс может иметь свою собственную версию виртуального метода.

- Этот факт особенно полезен в случае, когда доступ к объекту производного класса осуществляется через ссылочную переменную базового класса.

- В этой ситуации C# сам выбирает какую версию виртуального метода нужно вызвать. Этот выбор производится по типу объекта, на которую ссылается данная ссылка.

```
class DemoPoint        //базовый класс
{       protected int x;        protected int y;
   public virtual void Show()        //виртуальный метод
   {          Console.WriteLine("точка на плоскости: ({0}, {1})", x
   public DemoPoint(int x, int y)
   {          this.x = x; this.y = y;       }   }
class DemoShape : DemoPoint     //производный класс
{       protected int z;
   public override void Show() //перегрузка виртуального метода
   {          Console.WriteLine("точка в пространстве: ({0}, {1}, {2})", x, y, z);       }
   public DemoShape(int x, int y, int z)    : base(x, y) //конструктор производного класса
   {          this.z = z;       }   }
class DemoLine : DemoPoint        //производный класс
{       protected int x2;       protected int y2;
   public override void Show()      //перегрузка виртуального метода
   {          Console.WriteLine("отрезок на плоскости: ({0}, {1})-({2},{3})", x, y, x2, y2);       }
   public DemoLine(int x1, int y1, int x2, int y2)          : base(x1, y1)
   {          this.x2 = x2; this.y2 = y2;       }   }
class Program   {
   static void Main()       {
      DemoPoint point1 = new DemoPoint(0, 1);              point1.Show();
      DemoShape pointShape = new DemoShape(2, 3, 4);   pointShape.Show();
      DemoLine line = new DemoLine(0, 0, 10, 10);    line.Show();
      Console.WriteLine();
      //использование ссылки базового класса на объекты производных классов
      DemoPoint point2 = pointShape;          point2.Show();
      point2 = line;          point2.Show();
   }    }
```

```
точка на плоскости: (0, 1)
точка в пространстве: (2, 3, 4)
отрезок на плоскости: (0, 0)-(10,10)

точка в пространстве: (2, 3, 4)
отрезок на плоскости: (0, 0)-(10,10)
```

благодаря полиморфизму через ссылочную переменную возможно обращаться к объектам разного типа, а также с помощью одного и того же имени выполнять различные действия

Абстрактные методы и классы

- Иногда полезно создать базовый класс, определяющий только своего рода "пустой бланк", который унаследуют все производные классы, причем каждый из них заполнит этот "бланк" собственной информацией.

- Такой класс определяет структуру методов, которые производные классы должны реализовать, но сам при этом не обеспечивает реализации этих методов.

- Подобная ситуация может возникнуть, когда базовый класс попросту не в состоянии реализовать метод. В данной ситуации разрабатываются *абстрактные методы* или целые *абстрактные классы*.

- Абстрактный метод создается с помощью модификатора ***abstract***.

- Он не имеет тела и, следовательно, не реализуется базовым классом, а производные классы должны его обязательно переопределить.

- Абстрактный метод автоматически является виртуальным, однако использовать спецификатор ***virtual*** не нужно.

- Если попытаться использовать два спецификатора одновременно, ***abstract*** и ***virtual***, то компилятор выдаст сообщение об ошибке.

- Если класс содержит один или несколько абстрактных классов, то его также нужно объявить как абстрактный, используя спецификатор ***abstract*** перед ***class***.

- Т.к. абстрактный класс полностью не реализован, то невозможно создать экземпляр класса с помощью операции ***new***. Например, если класс Demo определен как абстрактный, то попытка создать экземпляр класса Demo повлечет ошибку:

Demo a = new Demo();

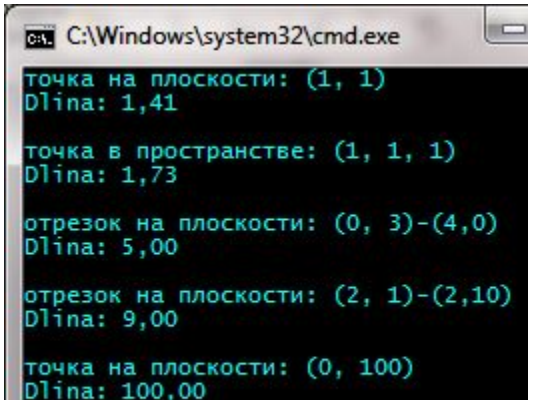
Но, можно создать массив ссылок, используя этот же абстрактный класс:

Demo [] Ob=new Demo[5];

- Если производный класс наследует абстрактный, то он должен полностью переопределить все абстрактные методы базового класса или также быть объявлен как абстрактный.

- Спецификатор ***abstract*** наследуется до тех пор, пока в производном классе не будут реализованы все абстрактные методы.

```
abstract class Demo                    //абстрактный класс
{   abstract public void Show();    abstract public double Dlina(); }    //абстрактный метод
    class DemoPoint : Demo          //производный класс от абстрактного
{   protected int x;            protected int y;
    public DemoPoint(int x, int y)    {       this.x = x; this.y = y;     }
    public override void Show()     //переопределение абстрактного метода
    {       Console.WriteLine("точка на плоскости: ({0}, {1})", x, y);     }
    public override double Dlina()  //переопределение абстрактного метода
    {       return Math.Sqrt(x * x + y * y);     }    }
class DemoShape : DemoPoint         //производный класс
{   protected int z;
    public DemoShape(int x, int y, int z)  : base(x, y)     {       this.z = z;     }
    public override void Show()     //переопределение абстрактного метода
    {       Console.WriteLine("точка в пространстве: ({0}, {1}, {2})", x, y, z);     }
    public override double Dlina()  //переопределение абстрактного метода
    {       return Math.Sqrt(x * x + y * y + z * z);     }     }
class DemoLine : DemoPoint //производный класс
{   protected int x2;    protected int y2;
    public DemoLine(int x1, int y1, int x2, int y2)       : base(x1, y1)
    {       this.x2 = x2; this.y2 = y2;     }
    public override void Show()     //переопределение абстрактного метода
    {       Console.WriteLine("отрезок на плоскости: ({0}, {1})-({2},{3})", x, y, x2, y2);     }
    public override double Dlina()  //переопределение абстрактного метода
    {       return Math.Sqrt((x - x2) * (x - x2) + (y - y2) * (y - y2));     }     }
class Program   {   static void Main()    {       Demo[] Ob = new Demo[5];    //массив ссылок
        //заполнения массива ссылками на объекты производных классов
        Ob[0] = new DemoPoint(1, 1);        Ob[1] = new DemoShape(1, 1, 1);
        Ob[2] = new DemoLine(0, 3, 4, 0);   Ob[3] = new DemoLine(2, 1, 2, 10);      Ob[4] = new DemoPoint(0, 100);
        foreach (Demo a in Ob)          //просмотр массива
        {       a.Show();           Console.WriteLine("Dlina: {0:f2}\n", a.Dlina());      }   }       }
```



Запрет наследования

Ключевое слово sealed,описывает класс, от которого запрещено наследование.

```
sealed class Demo {   … }
class newDemo: Demo {   …  }  // ошибка
```