**Программирование C#**

# 10. Наследование

Карбаев Д.С., 2015

# Наследование

- В языке C# класс, который наследуется, называется **базовым (родительским, суперклассом)**, а класс, который наследует, — **производным (дочерним, субклассом)**.
- Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним свои собственные элементы.

# Базовый класс TwoDShape

- Рассмотрим класс TwoDShape, содержащий ширину и высоту двухмерного объекта, например квадрата, прямоугольника, треугольника и т.д.

```
// Класс для двумерных объектов,
class TwoDShape
{
    public double Width;
    public double Height;
    public void ShowDim()
    {
        Console.WriteLine("Ширина и высота равны " +
        Width + " и " + Height);
    }
}
```

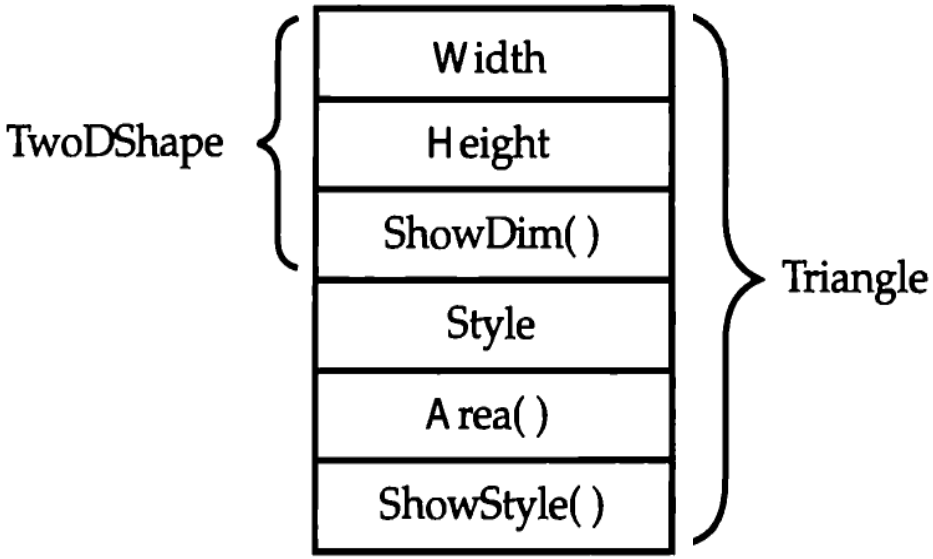
# Производный класс Triangle

```
// Класс Triangle, производный от класса TwoDShape
class Triangle : TwoDShape{
    public string Style; // тип треугольника
    public double Area() {// Возвратить площадь треугольника
        return Width * Height / 2.0;
    }
    public void ShowStyle() {// Показать тип треугольника
        Console.WriteLine("Треугольник " + Style);
    }
    class Shapes {
        static void Main() {
            Triangle t1 = new Triangle();
            Triangle t2 = new Triangle();
            t1.Width = 4.0; t1.Height = 4.0;
            t1.Style = "равнобедренный";
            t2.Width = 8.0; t2.Height = 12.0;
            t2.Style = "прямоугольный";
            Console.WriteLine("Сведения об объекте t1: ");
            t1.ShowStyle(); t1.ShowDim();
            Console.WriteLine("Площадь равна" + t1.Area());
            Console.WriteLine();
            Console.WriteLine("Сведения об объекте t2: ");
            t2.ShowStyle(); t2.ShowDim();
            Console.WriteLine("Площадь равна " + t2.Area());
}    }  }
```

**Результат:**

```
Сведения об объекте t1:
Треугольник равнобедренный
Ширина и высота равны 4 и 4
Площадь равна 8
Сведения об объекте t2:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 4 8
```

# Схематическое представление класса Triangle



# Наследование

- Несмотря на то что класс TwoDShape является базовым для класса Triangle, в то же время он представляет собой совершенно независимый и самодостаточный класс. Например, следующий фрагмент кода считается вполне допустимым.

```
TwoDShape shape = new TwoDShape();
shape.Width = 10;
shape.Height = 20;
shape.ShowDim();
```

- Разумеется, объект класса TwoDShape никак не связан с любым из классов,  производных от класса TwoDShape, и вообще не имеет к ним доступа.
- Общая форма объявления дочернего класса:

```
class имя_производного_класса : имя_базового_класса {
// тело класса
}
```

# Класс инкапсулирующий прямоугольники

```
// Класс для прямоугольников, производный от класса TwoDShape
class Rectangle : TwoDShape {
    // Возвратить логическое значение true, если
    // прямоугольник является квадратом
    public bool IsSquare()
    {
        if (Width == Height) return true;
        return false;
    }
    // Возвратить площадь прямоугольника
    public double Area()
    {
        return Width * Height;
    }
}
```

# Доступ к членам класса

- Закрытый член класса остается закрытым в своем классе. Он не доступен из кода за пределами своего класса, включая и производные классы.

Поскольку переменные Width и Height теперь являются закрытыми, то они доступны только для других членов своего класса, но не для членов производных классов.

```
// Доступ к закрытым членам класса не наследуется
class TwoDShape { // Класс для двумерных объектов
    double Width; // теперь это закрытая переменная
    double Height; // private
    public void ShowDim(){
        Console.WriteLine("Ширина и высота равны " +
        Width + " и " + Height);
    }
}
// Класс Triangle, производный от класса TwoDShape.
class Triangle : TwoDShape{
    public string Style; // тип треугольника
    public double Area(){// площадь треугольника
        return Width * Height / 2.0; // Ошибка, доступ к
                //закрытому члену класса запрещен
    }
    public void ShowStyle(){// Показать тип треугольника
        Console.WriteLine("Треугольник " + Style);
}   }
```

```
// Использовать открытые свойства для
установки и получения значений закрытых
членов класса.
class TwoDShape{
 double pri_width; // private
 double pri_height; // private
// Свойства ширины и высоты
//двумерного объекта
 public double Width{
  get{ return pri_width; }
  set{ pri_width= value<0 ? -value:value; }
 }
 public double Height{
  get{ return pri_height; }
  set{ pri_height= value<0 ?-value:value; }
  }
 public void ShowDim(){    {
  Console.WriteLine("Ширина и высота равны"
   + Width + " и " + Height);
  }
}
```

```
class Triangle : TwoDShape{
  public string Style; // тип треугольника
  // Возвратить площадь треугольника
  public double Area(){
    return Width * Height / 2.0;
  }
  // Напечатать тип треугольника
  public void ShowStyle(){
   Console.WriteLine("Треугольник " + Style);
  }
  class Shapes2{
   static void Main() {
     Triangle t1 = new Triangle();
     Triangle t2 = new Triangle();
     t1.Width = 4.0; t1.Height = 4.0;
     t1.Style = "равнобедренный";
     t2.Width = 8.0; t2.Height = 12.0;
     t2.Style = "прямоугольный";
     Console.WriteLine("Сведения об объекте"
                  + "t1: ");
     t1.ShowStyle(); t1.ShowDim();
     Console.WriteLine("Площадь равна "
                  + t1.Area());
     Console.WriteLine();
     Console.WriteLine("Сведения об объекте"
                  + "t2: ");
     t2.ShowStyle(); t2.ShowDim();
     Console.WriteLine("Площадь равна "
                  + t2.Area());
}    }    }
```

# Организация защищенного доступа

- Защищенное поле создается с помощью модификатора доступа **protected**. Если поле класса объявляется как protected, он становится закрытым, но за исключением одного случая, когда защищенный поле наследуется.

```
class B{
    protected int i, j; // поля, закрытые для класса B,
    // но доступные для класса D
    public void Set(int a, int b){ i = a; j = b; }
    public void Show(){ Console.WriteLine(i + " " + j); }
}
class D : B{
    int k; // закрытое поле
    // поля i и j класса B доступны для класса D
    public void Setk(){ k = i * j; }
    public void Showk(){ Console.WriteLine(k); }
}
class ProtectedDemo{
    static void Main(){
        D ob = new D();
        ob.Set(2, 3); // допустимо,
    // поскольку доступно для класса D
        ob.Show(); // допустимо,
    // поскольку доступно для класса D
        ob.Setk(); // допустимо,
    // поскольку входит в класс D
        ob.Showk(); // допустимо,
    // поскольку входит в класс D
    }
}
```

# Конструкторы и наследование

- конструктор базового класса конструирует базовую часть объекта, а конструктор производного класса — производную часть этого объекта.

```
class Triangle : TwoDShape{
    string Style;
    // Конструктор
    public Triangle(string s, double w, double h){
        // инициализировать поля базового класса
        Width = w; Height = h; Style = s;
    }
    public double Area(){// площадь треугольника
        return Width * Height / 2.0;
    }
    public void ShowStyle(){// показать тип треугольника
        Console.WriteLine("Треугольник " + Style);
    }
    class Shapes3{
     static void Main(){
   Triangle t1 = new Triangle("равнобедренный", 4.0, 4.0);
   Triangle t2 = new Triangle("прямоугольный", 8.0, 12.0);
   Console.WriteLine("Сведения об объекте t1: ");
   t1.ShowStyle(); t1.ShowDim();
   Console.WriteLine("Площадь равна " + t1.Area());
   Console.WriteLine();
   Console.WriteLine("Сведения об объекте t2: ");
   t2.ShowStyle(); t2.ShowDim();
   Console.WriteLine("Площадь равна " + t2.Area());
}    }  }
```

# Вызов конструкторов базового класса

- С помощью формы расширенного объявления конструктора производного класса и ключевого слова **base** в производном классе может быть вызван конструктор, определенный в его базовом классе. общая форма этого расширенного объявления:

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

где список_аргументов обозначает любые аргументы, необходимые конструктору в базовом классе. Обратите внимание на местоположение двоеточия.

# Вызов конструкторов базового класса

```
class TwoDShape{
    double pri_width;
    double pri_height;
    // Конструктор класса TwoDShape
    public TwoDShape(double w, double h) {
        Width = w; Height = h;
    }
//…
}

// Класс для треугольников, производный от класса TwoDShape
class Triangle : TwoDShape{
    string Style;
    // Вызвать конструктор базового класса
    public Triangle(string s, double w, double h): base(w, h) {
        Style = s;
    }
//…
}
```

```
// Добавим дополнительные конструкторы в класс TwoDShape
class TwoDShape{
    //...
    public TwoDShape(){// Конструктор, вызываемый по умолчанию
        Width = Height = 0.0;
    }
    // Конструктор класса TwoDShape.
    public TwoDShape(double w, double h){ Width = w; Height = h; }
    // Сконструировать объект равной ширины и высоты
    public TwoDShape(double x){ Width = Height = x; }
    //...
}
    // Класс для треугольников, производный от класса TwoDShape
class Triangle : TwoDShape{
    string Style;
    /* Конструктор, используемый по умолчанию.
    Автоматически вызывает конструктор, доступный по
    умолчанию в классе TwoDShape. */
    public Triangle(){
        Style = "null";
    }
    // Конструктор, принимающий три аргумента
    public Triangle(string s, double w, double h) : base(w, h) {
        Style = s;
    }
    // Сконструировать равнобедренный треугольник
    public Triangle(double x): base(x) {
        Style = "равнобедренный";
    }
    //...
}
```

# Вызов конструкторов базового класса

Основные принципы действия ключевого слова **base**.

- Когда в производном классе указывается ключевое слово base, вызывается конструктор из его непосредственного базового класса.
- Следовательно, ключевое слово base всегда обращается к базовому классу, стоящему в иерархии непосредственно над вызывающим классом.
- Аргументы передаются базовому конструктору в качестве аргументов метода base (). Если же ключевое слово отсутствует, то автоматически вызывается конструктор, используемый в базовом классе по умолчанию.

# Наследование и сокрытие имен

- В производном классе можно определить поле с таким же именем, как и у поля его базового класса. В этом случае поле базового класса скрывается в производном классе.
- Если поле базового  класса требуется скрыть намеренно, то перед его именем следует указать ключевое слово **new**, чтобы избежать появления подобного предупреждающего сообщения.
- То же справедливо и для методов.

# Наследование и сокрытие имен

```
// Пример сокрытия имени с наследственной связью
class A {
    public int i = 0;
}
// Создать производный класс B
class B : A {
    new int i; // это поле скрывает поле i из класса A
    public B (int b){
        i = b; // поле i в классе B
    }
    public void Show(){
        Console.WriteLine("Поле i в производном классе: " + i);
    }
}
class NameHiding{
    static void Main(){
        B ob = new B(2);
        ob.Show();
    }
}
```



**Результат:** Поле i в производном классе: 2

# Применение ключевого слова base для доступа к скрытому имени

- Имеется еще одна форма ключевого слова base, которая действует подобно  ключевому слову this, за исключением того, что она всегда ссылается на базовый класс в том производном классе, в котором она используется.
- Общий вид:

```
base.member
```

где member - может обозначать метод или переменную экземпляра.

# Применение ключевого слова base для доступа к скрытому имени

```
// Применение ключевого слова base для преодоления
// препятствия, связанного с сокрытием имен.
class A {
    public int i = 0;
}
// Создать производный класс
class B : A {
    new int i; // это поле скрывает поле i из класса A
    public B(int a, int b) {
        base.i = a; // здесь обнаруживается скрытое поле из класса A
        i = b; // поле i из класса B
    }
    public void Show() {
        // Здесь выводится поле i из класса A
        Console.WriteLine("поле i в базовом классе: " + base.i);
        //A здесь выводится поле i из класса B
        Console.WriteLine("поле i в производном классе: " + i);
    }
}
class UncoverName {
    static void Main() {
        B ob = new B(1, 2);
        ob.Show();
    }
}
```

# Применение ключевого слова base для доступа к скрытому имени

```
class A{
    public int i = 0;
    public void Show() {// Метод Show() в классе A
        Console.WriteLine("поле i в базовом классе: " + i);
}   }
class B : A{// Создать производный класс
    new int i; // этот поле скрывает поле i из класса A
    public B(int a, int b){
        base.i = a; // здесь обнаруживается скрытое поле из класса A
        i = b; // поле i из класса B
    }
    // Здесь скрывается метод Show() из класса A
    new public void Show() {
        base.Show(); // здесь вызывается метод Show() из класса A
        // далее выводится поле i из класса B
        Console.WriteLine("поле i в производном классе: " + i);
}   }
class UncoverName{
    static void Main(){
        B ob = new B(1, 2);
        ob.Show();
}   }
```

**Результат:** поле i в базовом классе: 1
поле i в производном классе: 2

# Создание многоуровневой иерархии классов

- ключевое слово base всегда обозначает ссылку на конструктор ***ближайшего*** по иерархии ***базового класса***.

```
class TwoDShape {
//...
}
// Класс для треугольников, производный от класса TwoDShape
class Triangle : TwoDShape {
//...
}
// Расширить класс Triangle
class ColorTriangle : Triangle {
    string color;
    public ColorTriangle(string c, string s,
                double w, double h) : base(s, w, h){
        color = c;
    }
    // Показать цвет треугольника
    public void ShowColor(){
        Console.WriteLine("Цвет " + color);
    }
}
```

# Порядок вызова конструкторов

- в иерархии классов конструкторы вызываются по порядку выведения классов: от базового к производному.

```
// Продемонстрировать порядок вызова конструкторов.
// Создать базовый класс
class A{
    public A(){
        Console.WriteLine("Конструирование класса A.");
}   }
// Создать класс, производный от класса A
class B : A{
    public B(){
        Console.WriteLine("Конструирование класса B.");
}   }
// Создать класс, производный от класса B
class C : B{
    public C(){
        Console.WriteLine("Конструирование класса C.");
}   }
class OrderOfConstruction{
    static void Main(){
        C c = new C();
}   }
```

**Результат:**

Конструирование класса A.
Конструирование класса B.
Конструирование класса C.

# Ссылки на базовый класс и объекты производных классов

- Переменная ссылки на объект класса одного типа, как правило, не может ссылаться на объект класса другого.

```
// Эта программа не подлежит компиляции
class X {
    int a;
    public X(int i) { a = i; }
}
class Y {
    int a;
    public Y(int i) { a = i; }
}
class IncompatibleRef{
    static void Main(){
        X x = new X(0); X x2;
        Y y = new Y();
        x2 = x; // верно, поскольку оба объекта относятся к одному и тому же типу
        x2 = y; // ошибка, поскольку это разнотипные объекты
    }
}
```

# Ссылки на базовый класс и объекты производных классов

```
//По ссылке на объект базового класса можно обращаться
// к объекту производного класса.
class X {
    public int a;
    public X(int i){
        a = i;
}   }
class Y : X {
    public int b;
    public Y(int i, int j) : base(j){
        b = i;
}   }
class BaseRef {
    static void Main(){
        X x = new X(0);
        X x2;
        Y y = new Y(5, 6);
        x2 = x; // верно, поскольку оба объекта относятся к одному и тому же типу
        Console.WriteLine("x2.a: " + x2.a);
        x2 = y; // тоже верно, поскольку класс Y является производным от класса X
        Console.WriteLine("x2.a: " + x2.a);
        // ссылкам на объекты класса X известно только о полях класса X
        x2.a = 19; // верно
        // x2.b = 27; // неверно, поскольку поле b отсутствует у класса X
}   }
```

# Ссылки на базовый класс и объекты производных классов

```
class TwoDShape {
    //...
    // Сконструировать копию объекта TwoDShape
    public TwoDShape(TwoDShape ob) {
        Width = ob.Width;
        Height = ob.Height;
}   }
class Triangle : TwoDShape {
  // Сконструировать копию объекта типа Triangle
  public Triangle(Triangle ob): base(ob){
      Style = ob.Style;
  }
  class Shapes {
     static void Main() {
      Triangle t1 = new Triangle("прямоугольный", 8.0, 12.0);
      // Сделать копию объекта t1
      Triangle t2 = new Triangle(t1);
      Console.WriteLine("Сведения об объекте t1: ");
      t1.ShowStyle(); t1.ShowDim();
      Console.WriteLine("Площадь равна " + t1.Area());
      Console.WriteLine("Сведения об объекте t2: ");
      t2.ShowStyle(); t2.ShowDim();
      Console.WriteLine("Площадь равна " + t2.Area());
} } }
```

**Сведения об объекте t1:**
**Треугольник прямоугольный**
**Ширина и высота равны 8 и 12**
**Площадь равна 48**
**Сведения об объекте t2:**
**Треугольник прямоугольный**
**Ширина и высота равны 8 и 12**
**Площадь равна 48**

# Виртуальные методы и их переопределение

- **Виртуальным** называется такой метод, который объявляется как **virtual** в базовом классе. Виртуальный метод может быть переопределен в одном или нескольких производных классах. Следовательно, у каждого производного класса может быть свой вариант виртуального метода.
- Средствами языка C# определяется именно тот вариант виртуального метода, который следует вызывать, исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения. Поэтому при ссылке на разные типы объектов выполняются разные варианты виртуального метода. Иными словами, вариант выполняемого виртуального метода выбирается по типу объекта, а не по типу ссылки на этот объект.
- Метод объявляется как виртуальный в базовом классе с помощью ключевого слова virtual, указываемого перед его именем. Когда же виртуальный метод *переопределяется в производном классе*, то для этого используется модификатор **override**. А сам процесс повторного определения виртуального метода в производном классе называется **переопределением метода**.
- Переопределение метода служит основанием для воплощения одного из самых эффективных в C# принципов: динамической обработки методов, которая представляет собой механизм разрешения вызова *во время выполнения*, а не компиляции.

# Виртуальные методы и их переопределение

```
class Base{
    // Создать виртуальный метод в базовом классе
    public virtual void Who(){
        Console.WriteLine("Метод Who() в классе Base");
}   }
class Derived1 : Base{
    // Переопределить метод Who() в производном классе
    public override void Who(){
        Console.WriteLine("Метод Who() в классе Derived1");
}   }
class Derived2 : Base{
    // Вновь переопределить метод Who() в еще одном
    // производном классе
    public override void Who(){
        Console.WriteLine("Метод Who() в классе Derived2");
}   }
class OverrideDemo{
    static void Main(){
        Base baseOb = new Base();
        Derived1 dOb1 = new Derived1();
        Derived2 dOb2 = new Derived2();
        Base baseRef; // ссылка на базовый класс
        baseRef = baseOb; baseRef.Who();
        baseRef = dOb1; baseRef.Who();
        baseRef = dOb2; baseRef.Who();
}   }
```

```
Метод Who() в
классе Base.
Метод Who() в
классе Derived1
Метод Who() в
классе Derived2
```

# Виртуальные методы и их переопределение

```
 /* Если виртуальный метод не переопределяется, то
используется его вариант из базового класса. */
class Base{
    // Создать виртуальный метод в базовом классе
    public virtual void Who(){
        Console.WriteLine("Метод Who() в классе Base1");
}   }
class Derived1 : Base{
    // Переопределить метод Who() в производном классе
    public override void Who(){
      Console.WriteLine("Метод Who() в классе Derived1");
}   }
class Derived2 : Base{
    // В этом классе метод Who() не переопределяется
}
class NoOverrideDemo{
    static void Main(){
        Base baseOb = new Base();
        Derived1 dOb1 = new Derived1();
        Derived2 dOb2 = new Derived2();
        Base baseRef; // ссылка на базовый класс
        baseRef = baseOb; baseRef.Who();
        baseRef = dOb1; baseRef.Who();
        // вызывается метод Who() из класса Base
        baseRef = dOb2; baseRef.Who();
}   }
```

```
Метод Who() в классе Base
Метод Who() в классе
Derived1
Метод Who() в классе Base
```

# Что дает переопределение методов

- Благодаря переопределению методов в C# поддерживается **динамический полиморфизм**. В объектно-ориентированном программировании полиморфизм играет очень важную роль, потому что он позволяет определить в общем классе методы, которые становятся общими для всех производных от него классов, а в производных классах — определить конкретную реализацию некоторых или же всех этих методов.
- Переопределение методов — это еще один способ воплотить в C# **главный принцип полиморфизма**: один интерфейс — множество методов.
- Таким образом, сочетая наследование с виртуальными методами, можно определить в базовом классе общую форму методов, которые будут использоваться во всех его производных классах.

```
class TwoDShape {
    //...
    public virtual double Area() {
        Console.WriteLine("Метод Area() должен быть переопределен");
        return 0.0;
}   }
class Triangle : TwoDShape{
    //...
    public override double Area(){
        return Width * Height / 2.0;
}   }
class Rectangle : TwoDShape {
    //...
    public override double Area(){
            return Width * Height;
}   }
class DynShapes{
   static void Main(){
      TwoDShape[] shapes = new TwoDShape[5];
      shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
      shapes[1] = new Rectangle(0);shapes[2] = new Rectangle(0, 4);
      shapes[3] = new Triangle(0);
      shapes[4] = new TwoDShape(0, 20, "общая форма");
      for (int i = 0; i < shapes.Length; i++) {
         Console.WriteLine("Объект - " + shapes[i].name);
         Console.WriteLine("Площадь равна " + shapes[i].Area());
         Console.WriteLine();
}  }
```

```
Объект — треугольник
Площадь равна 48
Объект — прямоугольник
Площадь равна 100
Объект — прямоугольник
Площадь равна 40
Объект — треугольник
Площадь равна 24.5
Объект — общая форма
Метод Area() должен быть
переопределен
Площадь равна 0
```

# Применение абстрактных классов

- В **абстрактном классе** определяется лишь сигнатура методов, которые должны быть конкретно реализованы в производных классах, а не в самом базовом классе. Подобная ситуация возникает, например, в связи с невозможностью получить содержательную реализацию метода в базовом классе.
- Абстрактный метод создается с помощью указываемого модификатора типа **abstract**. У абстрактного метода отсутствует тело, и поэтому он не реализуется в базовом классе.
- Совместное использование модификаторов virtual и abstract считается ошибкой.
- Общая форма:

```
abstract тип имя(список_параметров);
```

```
abstract class TwoDShape {// Теперь нельзя создать объект класса TwoDShape
  //…
  // Теперь метод Area() является абстрактным
  public abstract double Area();
}
class Triangle : TwoDShape {
  // Переопределить метод Area() для класса Triangle
  public override double Area() {
    return Width * Height / 2.0;
} }
class Rectangle : TwoDShape {
  // Сконструировать копию объекта типа Rectangle
  public Rectangle(Rectangle ob) : base(ob) { }
  // Переопределить метод Area() для класса Rectangle
  public override double Area() {
   return Width * Height;
} }
class AbsShape {
  static void Main() {
    TwoDShape[] shapes = new TwoDShape[4];
    shapes[0] = new Triangle("прямоугольный", 8.0, 12.0);
    shapes[1] = new Rectangle(10);
    shapes[2] = new Rectangle(10, 4);
    shapes[3] = new Triangle(2.0);
    for(int i=0; i < shapes.Length; i++) {
      Console.WriteLine("Объект - " + shapes[i].name);
      Console.WriteLine("Площадь равна " + shapes[i].Area());
} }
```

# Предотвращение наследования

- Для того чтобы предотвратить наследование класса, достаточно указать ключевое слово **sealed** перед определением класса. Класс не допускается объявлять одновременно как abstract и sealed.
- Пример объявления класса типа sealed.

```
sealed class A {
// ...
}
// Следующий класс недопустим.
class B : A {
// ОШИБКА! Наследовать
//класс A нельзя
// ...
}
```

```
class B{
   public virtual void MyMethod() { /* ... */ }
}
class D : B{
  // Здесь герметизируется метод MyMethod() и
  // предотвращается его дальнейшее переопределение
  sealed public override void MyMethod() { /*...*/}
}
class X : D{
  // Ошибка! Метод MyMethod() герметизирован!
  public override void MyMethod() { /* ... */ }
}
```

- Ключевое слово sealed может быть также использовано в виртуальных методах для предотвращения их дальнейшего переопределения.

# Класс Object

- В C# предусмотрен специальный класс **object**, который неявно считается базовым классом для всех остальных классов и типов, включая и типы значений.
- Переменная ссылочного типа object может ссылаться на объект любого другого типа. Кроме того, переменная типа object может ссылаться на любой массив, поскольку в C# массивы реализуются как объекты. Формально имя object считается в C# еще одним обозначением класса System.Object, входящего в библиотеку классов для среды .NET Framework.

# Методы класса Object

| Метод | Назначение |
|---|---|
| `public virtual bool Equals(object ob)` | Определяет, является ли вызывающий объект таким же, как и объект, доступный по ссылке *ob* |
| `public static bool Equals(object objA, object objB)` | Определяет, является ли объект, доступный по ссылке *objA*, таким же, как и объект, доступный по ссылке *objB* |
| `protected Finalize()` | Выполняет завершающие действия перед "сборкой мусора". В C# метод `Finalize()` доступен посредством деструктора |
| `public virtual int GetHashCode()` | Возвращает хеш-код, связанный с вызывающим объектом |
| `public Type GetType()` | Получает тип объекта во время выполнения программы |
| `protected object MemberwiseClone()` | Выполняет неполное копирование объекта, т.е. копируются только члены, но не объекты, на которые ссылаются эти члены |
| `public static bool ReferenceEquals(obj objA, object objB)` | Определяет, делаются ли ссылки *objA* и *objB* на один и тот же объект |
| `public virtual string ToString()` | Возвращает строку, которая описывает объект |

# Класс Object

```
// Продемонстрировать применение метода ToString()
    class MyClass {
        static int count = 0; int id;
        public MyClass(){
            id = count;
            count++;
        }
        public override string ToString() {
            return "Объект #" + id + " типа MyClass";
        }
    }
class Test {
    static void Main() {
        MyClass ob1 = new MyClass();
        MyClass ob2 = new MyClass();
        MyClass ob3 = new MyClass();
        Console.WriteLine(ob1);
        Console.WriteLine(ob2);
        Console.WriteLine(ob3);
    }
}
```

```
Результат:
Объект #0 типа
MyClass
Объект #1 типа
MyClass
Объект #2 типа
MyClass
```

# Упаковка и распаковка

- Ссылкой типа object можно  воспользоваться для обращения к любому другому типу, в том числе и к типам значений.
- Когда ссылка на объект класса object используется для обращения к типу значения, то такой процесс называется ***упаковкой***.
- ***Распаковка*** представляет собой процесс извлечения упакованного значения из объекта. Это делается с помощью явного приведения типа ссылки на объект класса object к соответствующему типу значения. Попытка распаковать объект в другой тип может привести к ошибке во время выполнения.

# Упаковка и распаковка

```
// Простой пример упаковки и распаковки
class BoxingDemo {
    static void Main(){
        int x=10; object obj;
        obj = x;
// упаковать значение переменной
// x в объект
        int y = (int)obj;
// распаковать значение из объекта,
// доступного по ссылке obj,
// в переменную типа int
        Console.WriteLine(y);
    }
}
```

```
// Пример упаковки при передаче значения методу
class BoxingDemo{
 static void Main(){
  int x = 10;
  Console.WriteLine("Значение x равно: " + x);
  // значение x автоматически упаковывается,
  // когда оно передается методу Sqr()
  x = BoxingDemo.Sqr(x);
  Console.WriteLine("Значение x в квадрате"
    + "равно: " + x);
 }
 static int Sqr(object o){
     return (int)o*(int)o;
 }
}
```

Значение x равно: 10

Значение x в квадрате равно: 100

```
// Благодаря упаковке становится возможным вызов методов по значению!
class MethOnValue {
  static void Main() {
    Console.WriteLine(10.ToString());
   }
}
```

# Класс Object как универсальный тип данных

```
// Использовать класс object для создания массива "обобщенного" типа.
 class GenericDemo{
     static void Main(){
        object[] ga = new object[10];

        // Сохранить целые значения.
        for (int i = 0; i < 3; i++) ga[i] = i;


        // Сохранить значения типа double.
        for (int i = 3; i < 6; i++) ga[i] = (double)i / 2;


        // Сохранить две строки, а также значения типа bool и char.
        ga[6] = "Привет";
        ga[7] = true;
        ga[8] = 'X';
        ga[9] = "Конец";


        for (int i = 0; i < ga.Length; i++)
            Console.WriteLine("ga[" + i + "] :" + ga[i] + "");
    }
 }
```

```
ga[0] : 0
ga[1] : 1
ga[2] : 2
ga[3] : 1.5
ga[4] : 2
ga[5] : 2.5
ga[6] : Привет
ga[7] : True
ga[8] : X
ga[9] : Конец
```