

Классы и структуры в С#

Класс (class) – ссылочный тип.

Структура (struct) – тип-значение.

Структура не может

- иметь деструктор;
- иметь конструктор без параметров;
- использоваться как базовый тип.

Члены класса

- константы;
- поля (field);
- конструкторы (в том числе без параметров);
- деструктор;
- методы (статические и экземплярные);
- свойства (property);
- индексаторы (свойства с параметрами);
- события (event);
- вложенные типы.

Доступ к типам и членам класса (полям и методам)

- ✓ Модификаторы доступа **для типов**:
 - public - тип доступен из любой сборки;
 - internal - тип доступен только внутри данной сборки (умолчание).
- ✓ Модификаторы доступа **для членов класса**

public	член класса доступен в любом методе любой сборки
protected	в методах самого класса, в методах производных классов любой сборки
internal	в любом методе данной сборки
protected internal	в любом методе данной сборки, в производных классах других сборок
private	только в методах самого класса

```
using System;
namespace First_Sample
{
    public class Person
    {
        private string[] names;
        private DateTime date { get;set;}

        public Person(string first_name, string second_name, DateTime date)
        {
            names = new string[] { first_name, second_name };
            this.date = date;
        }
    }
}
```

- ✓ Вложенный тип может иметь любой из 5 модификаторов доступа, но уровень доступа не может быть выше уровня доступа объемлющего типа.

Константы и поля readonly

- ✓ Локальные переменные, поля классов и структур могут иметь модификатор `const`. Константы инициализируются при объявлении. Значения присваиваются при компиляции.
- ✓ Поля классов и структур могут иметь модификатор `readonly`. Поле с модификатором `readonly` можно инициализировать только при объявлении и/или в конструкторе и нельзя изменить в любом другом методе.

```
partial class Model
{
    const double MaxWaveLength = 64.5;
    readonly int nThreads = 2;

    public Model()
    { nThreads = Environment.ProcessorCount;
    }

    public override string ToString()
    { return "MaxWaveLength = " + MaxWaveLength +
        " \nnThreads = " + nThreads;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Model model = new Model();
        Console.WriteLine(model.ToString());
    }
}
```

Вывод:

```
MaxWaveLength = 64,5
nThreads = 8
```

Статические методы и данные

- ✓ Методы и поля данных объявляются как статические с помощью модификатора `static`.
- ✓ Статические данные совместно используются всеми экземплярами класса.
- ✓ Статический метод может быть вызван еще до создания экземпляра (instance) класса.
- ✓ Статический метод вызывается через класс. В примере вызывается статический метод `Sin` из статического класса `System.Math`.

```
double res = Math.Sin(Math.PI/6);  
Console.WriteLine("res= {0}", res);
```

- ✓ В версиях C# 6.0 и выше можно использовать директиву `using static`, чтобы не указывать имя типа при вызове его статических членов.

```
using System;  
using static System.Math;  
namespace Samples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine( Sin(PI / 6));  
        }  
    }  
}
```

Инициализация объектов: конструкторы

- ✓ При создании объекта выполняются два действия:
 - выделяется память под объект;
 - выделенный участок памяти инициализируется.

```
T t = new T();
```

Выделение памяти / *Инициализация объекта*

- ✓ Инициализацию выделенного участка памяти выполняет специальный метод класса – конструктор.
- ✓ В классе может быть определено несколько конструкторов с различными списками параметров.
- ✓ Конструктор без параметров называется конструктором по умолчанию.
- ✓ Если конструктор по умолчанию не задан, компилятор сам создает конструктор по умолчанию.
- ✓ Если в классе определен хотя бы один конструктор, конструктор по умолчанию не создается.

Список инициализаторов

- ✓ Чтобы избежать дублирования кода, можно вызывать один конструктор из другого при помощи специальной синтаксической конструкции - списка инициализаторов.

```
public T(string key)
{
    this.key = key;
    nobj++;
}
```

```
public T(string key, double []dt) : this(key)
{
    this.dt = new double[dt.Length] ;
    Array.Copy(dt, this.dt, dt.Length);
}
```

- ✓ Список инициализаторов применим только в конструкторах и не может ссылаться сам на себя (рекурсивный вызов конструкторов).

Статический конструктор

- ✓ В статическом конструкторе обычно инициализируют статические поля класса.
- ✓ Статический конструктор
 - не может иметь модификатор доступа;
 - не может принимать параметры.
- ✓ Гарантируется, что статический конструктор
 - будет вызван перед созданием первого экземпляра объекта;
 - будет вызван перед обращением к любому статическому полю класса или статическому методу класса;
 - будет вызван не более одного раза за все время выполнения программы.

```
partial class T
{
    static int nobj;

    static T()
    { // Инициализация статических полей
    }
}
```

Статические классы

- ✓ Статические классы поддерживаются версиями 2.0 и выше.
- ✓ Нельзя создать экземпляр статического класса.
- ✓ Любой статический класс имеет непосредственный базовый класс `object`, но не может быть базовым для другого класса.
- ✓ Статический класс может содержать только статические члены с типом доступа `private`, `internal` или `public`.
- ✓ В статическом классе нельзя определить операции.
- ✓ Примеры статических классов – `System.Console` и `System.Math`.

Свойства

- ✓ Свойство - пара методов со специальными именами:
 - метод `get()` вызывается при получении значения свойства;
 - метод `set()` вызывается при задании значения свойства.
- ✓ Каждое свойство имеет имя и тип.
- ✓ Если определен только метод `get`, свойство доступно только для чтения. Если определен только метод `set`, свойство доступно только для “записи”.
- ✓ Свойство может быть связано с закрытым полем класса.
- ✓ Свойства работают немного медленнее прямого доступа к полю.

Свойства. Пример

```
partial class Book
{   private string title;
    private string author;
    private int year;

    public Book(string title, string author, int year)
    {   this.title = title;
        this.author = author;
        this.year = year;
    }

    public int Year
    {   get { return year; }
        set { year = value; }
    }

    public string Title
    {   get { Console.WriteLine("Title get");
            return title;
        }
        set { Console.WriteLine("Title set");
            Console.WriteLine("title = " + title);
            Console.WriteLine("value = " + value);
            title = value;
        }
    }

    public override string ToString()
    {   return title + " " + author + " " + year; }
}
```

```
static void Main(string[] args)
{
    Book book =
        new Book("С#", "Эндрю Троелсен", 2005);
    Console.WriteLine(book);

    string book_title = book.Title;
    Console.WriteLine(book_title);

    book.Title = "???";
    Console.WriteLine(book);
}
```

Вывод:

```
С# Эндрю Троелсен 2005
Title get
С#
Title set
title = С#
value = ???
?? Эндрю Троелсен 2005
```

Автореализуемые свойства.

- ✓ Для автореализуемых свойств компилятор генерирует закрытые поля для хранения значений и методы для доступа к ним.
- ✓ В примере все свойства объявлены как автореализуемые

```
partial class Book
{
    public string Title {get; set;}
    public string Author {get; set;}
    public int Year {get; set;}

    public Book()
    { Title = "C#";
      Author = "Э.Троелсен";
      Year = 2017;
    }

    public Book(string title, string author, int year)
    {
        Title= title;
        Author = author;
        Year = year;
    }

    public override string ToString()
    { return Title + " " + Author + " " + Year; }
}
```

- ✓ Свойствам, в том числе автореализуемым, можно присвоить значения в инициализаторах объектов.

```
static void Main(string[] args)
{
    Book book_1 = new Book();
    Console.WriteLine(book_1);

    Book book_2 =
        new Book() {Author = "Джеффри Рихтер"};
    Console.WriteLine(book_2);
}
```

Вывод:

```
C# Э.Троелсен 2017
C# Джеффри Рихтер 2017
```

Индексаторы (свойства с параметрами)

- ✓ В C# для определения оператора индексирования [] используется специальный синтаксис.
- ✓ Индексатор может быть перегружен – можно определить индексаторы различных типов с различным числом и типами параметров.
- ✓ В примере определены два индексатора – индексатор типа `KeyValuePair` с одним параметром типа `int` и индексатор типа `KeyValuePair` с одним параметром типа `string`.

```
class SampleClass
{
    KeyValuePair[] pairs =
        new KeyValuePair[] { new KeyValuePair(1, "one"),
                            new KeyValuePair(2, "two") };

    public KeyValuePair this[int index]
    {
        get { return pairs[index]; }
        set { pairs[index] = value; }
    }

    public KeyValuePair this[string key]
    {
        get
        {
            for (int j=0; j<pairs.Length; j++)
                if ( key == pairs[j].Key) return pairs[j];
            return null;
        }
    }
}
```

Вывод:

```
Key = one Value = 1
Key = two Value = 2
```

```
class KeyValuePair
{
    public int Value { get; set; }
    public string Key { get; set; }
    public KeyValuePair(int value, string key)
    {
        Value = value;
        Key = key;
    }
    public override string ToString()
    {
        return "Key = " + Key + " Value = " + Value;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        SampleClass sampleClass = new SampleClass();
        Console.WriteLine(sampleClass[0]);
        Console.WriteLine(sampleClass["two"]);
    }
}
```

Передача параметров размерных типов

```
struct S {...};
```

```
class Program
```

```
{  
    void F (S p1, ref S p2, out S p3) {...}  
    ...  
}
```

p1	передается по значению - делается ограниченная (shallow) копия, изменения не возвращаются в контекст вызова;
p2 и p3	передаются по ссылке, изменения возвращаются в контекст вызова;
p2	перед вызовом должен быть инициализирован;
p3	может быть не инициализирован, в методе должно быть присвоено значение.

Передача параметров размерных типов. Пример

```
class Program
{
    static void Main(string[] args)
    {
        int i1 = 1;
        int i2 = 1;
        int i3;

        fV(i1, ref i2, out i3);

        Console.WriteLine("i1 = " + i1 + "\ni2 = " + i2 + "\ni3 = " + i3 );
    }

    static void fV(int par1, ref int par2, out int par3)
    {
        par1 = 2;
        par2 = 2;
        par3 = 2;
    }
}
```

Вывод:

```
i1 = 1
i2 = 2
i3 = 2
```

Передача параметров ссылочных типов

```
class Program
{
    static void f1 (double[] darr) // аналог в C++ double*
    {
        darr[0] = 555;
        darr= new double[]{5,15};
    }

    static void f2 (ref double[] darr) // аналог в C++ double* &
    {
        darr[0] = 777;
        darr= new double[]{7,17};
    }
}
```

```
static void Main(string[] args)
{
    double[] dm = {1,2,3};

    f1(dm);
    for (int j=0; j<dm.Length; j++)
        Console.Write(dm[j]);    // 555 2 3

    f2(ref dm);
    for (int j=0; j<dm.Length; j++)
        Console.Write(dm[j]);    // 7 17
}
}
```

f1

делается копия ссылки dm, при выходе из f1 значение dm не изменится, но элементы массива изменятся

f2

при выходе из f2 значение dm изменится

Методы с переменным числом параметров

```
public static void f (T1 p1, T2 p2, params T[] p)
{...}
```

- ✓ Модификатор `params` используется для объявления метода с переменным числом параметров.
- ✓ Только последний параметр метода может иметь модификатор `params`, только один параметр метода может иметь модификатор `params`.
- ✓ Формальный параметр с модификатором `params` должен быть одномерным массивом.
- ✓ При вызове метода с модификатором `params` как фактическое значение параметра можно передать
 - одномерный массив значений;
 - любое число разделенных запятой значений типа, который указан в объявлении метода как тип элементов массива.
- ✓ Если при вызове не найден метод с точным совпадением типов фактических параметров, последние параметры собираются в массив и вызывается метод с модификатором `params`.
- ✓ При вызове метода можно не передавать значение параметру с модификатором `params`. В этом случае считается, что длина массива равна 0.

Методы с переменным числом параметров. Пример

```
class Program
{
    static void Main(string[] args)
    { string[] sarray = { "xx", "yy", "zz" };

        fp(1, "xyz");
        fp(sarray);
        fp("ab", "cd", "ef");
        fp(1, "xyz", 5, 8, "uvw");
    }

    static void fp(int i, string s)
    { Console.WriteLine("\nfp(int, string)\n i={0} s={1}", i, s);
    }

    static void fp(int i, string s, params object[] objs)
    { Console.WriteLine("\nfp(int, string, params object[] \n i={0} s={1}", i, s);
      for (int j = 0; j < objs.Length; j++ ) Console.WriteLine(objs[j]);
      Console.WriteLine();
    }

    public static void fp( params string[] str)
    { Console.WriteLine("\nfp(params string[])");
      for (int j = 0; j < str.Length; j++) Console.WriteLine(strs[j]);
      Console.WriteLine();
    }
}
```

Вывод:

fp(int, string)

i=1 s=xyz

fp(params string[])

xx

yy

zz

fp(params string[])

ab

cd

ef

fp(int, string, params object[])

i=1 s=xyz

5

8

uvw

Именованные и опциональные параметры

- ✓ Именованные и опциональные параметры поддерживаются версией языка C# Visual C# 2010. Могут использоваться в методах, индексаторах, конструкторах и делегатах.
- ✓ Именованные параметры позволяют при вызове метода указать значение для формального параметра с использованием его имени (с изменением порядка параметров в списке формальных параметров).
- ✓ Опциональные параметры дают возможность при вызове метода опустить значения тех параметров, для которых при определении метода были указаны значения по умолчанию.
- ✓ При использовании именованных и опциональных параметров они вычисляются в том порядке, в котором они были перечислены в списке формальных параметров.

Именованные и опциональные параметры. Пример

```
class Abc
{
    string str_1;
    string str_2;
    string str_3;

    public Abc()
    {
        str_1 = "Default_1";
        str_2 = "Default_2";
        str_3 = "Default_3";
    }

    public Abc ( string str_1 = "s1",
                string str_2 = "s2",
                string str_3 = "s3")
    {
        this.str_1 = str_1;
        this.str_2 = str_2;
        this.str_3 = str_3;
    }

    public static string F( char symbol = '1', int count = 2)
    {
        return new string(symbol, count);
    }

    public override string ToString()
    {
        return str_1 + " " + str_2 + " " + str_3;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        string res_1 = Abc.F(count: 5);
        Console.WriteLine(res_1);

        Abc abc_1 = new Abc();
        Console.WriteLine(abc_1);

        Abc abc_2 = new Abc("test_1");
        Console.WriteLine(abc_2);

        Abc abc_3 = new Abc(str_2: "test_2");
        Console.WriteLine(abc_3);
    }
}
```

Вывод:

```
11111
Default_1 Default_2 Default_3
test_1 s2 s3
s1 test_2 s3
```

Перегрузка операторов для классов и структур

- ✓ Возможна перегрузка бинарных операторов:

+ - * % / | & ^ << >>

и унарных операторов

+ - ~ ! ++ --

- ✓ Операторы перегружаются с помощью **открытых статических методов**. Один из параметров должен совпадать с объемлющим типом.

- ✓ Операторы (составные операции присваивания)

+= -= *= /= %= >>= <<= &= |= ^=

не могут быть перегружены, но когда перегружены соответствующие им бинарные операции, последние

используются при вычислении выражений с операторами += -= *= /= %= >>=

<<= &= |= ^=

Перегрузка операторов. Пример

```
partial struct Rational
```

```
{  
    long a, b;  
    public Rational(long a, long b)  
    { this.a = a;  
      this.b = b;  
      if (b < 0)  
          { this.b = -b;  
            this.a = -a;  
          }  
    }  
}
```

```
// бинарный +
```

```
public static Rational operator+ (Rational ls, Rational rs)  
    { return new Rational (ls.a*rs.b + ls.b*rs.a , ls.b*rs.b); }
```

```
// унарный -
```

```
public static Rational operator- (Rational r)  
    { return new Rational(-r.a, r.b); }
```

```
}
```

```
static void Main(string[] args)
```

```
{ Rational r1 = new Rational(1,2);  
  Rational r2 = new Rational(1,3);  
  Rational r3 = r1 + r2;  
  r2 += r1; // Вычисляется как r2 = r2 + r1;  
  Rational r4 = -r1;  
}
```

Перегрузка операторов ++ и --

Операторы ++ и -- могут применяться в префиксной и постфиксной форме:

```
r1 = ++a1; // Вычисляется как a1 = operator ++ (a1); r1 = a1;
```

```
r2 = a2++; // Вычисляется как r2 = a2; a2 = operator ++ (a2);
```

```
partial struct Rational
{
    public static Rational operator ++ (Rational r)
        { return new Rational(r.a + r.b, r.b); }

    public override string ToString()
        { return "(" + a + "," + b + ")"; }
}
```

```
static void Main(string[] args)
{
    Rational a1 = new Rational(1,2);
    Rational a2 = new Rational(1,2);
    Rational r1 = ++a1;
    Console.WriteLine("a1={0} r1={1}", a1, r1); // a1=(3,2) r1=(3,2)
    Rational r2 = a2++;
    Console.WriteLine("a2={0} r2={1}", a2, r2); // a2=(3,2) r2=(1,2)
}
```

Перегрузка операторов сравнения

- ✓ Перегрузка операторов сравнения возможна только в парах:
 <= и >= < и > == и !=
- ✓ Чтобы избежать дублирования кода и связанных с ним ошибок, рекомендуется поместить код для сравнения в единственный статический метод и вызывать его из методов, перегружающих операторы.

```
partial struct Rational
{
    public static int Compare(Rational ls, Rational rs)
    {
        long d = ls.a*rs.b - rs.a*ls.b;
        if(d < 0)
            return -1; // Первый аргумент меньше второго
        else if(d > 0)
            return 1; // Первый аргумент больше второго
        else
            return 0;
    }

    public static bool operator < (Rational ls, Rational rs)
    { return (Compare(ls,rs) < 0); }

    public static bool operator > (Rational ls, Rational rs)
    { return (Compare(ls,rs) > 0); }
}
```

Операторы преобразования типов

✓ Можно определить явные и неявные преобразования для структуры или класса.

```
partial struct Rational
{
    // Оператор явного преобразования bool -> Rational
    public static explicit operator Rational ( bool val )
        { int a = val ? 1 : -1;
          return new Rational( a,1);
        }

    // Оператор неявного преобразования int -> Rational
    public static implicit operator Rational ( int val )
        { return new Rational(val,1); }

    // Оператор явного преобразования Rational -> bool
    public static explicit operator bool ( Rational r )
        { return r.a > 0; }
}
```

```
static void Main(string[] args)
{
    Rational r1 = 5;
    bool b1 = true;
    Rational r2 = (Rational) b1;
    bool b2 = ( bool) r1;
}
```