Программирование C#

9. Виды классов

Карбаев Д.С., 2015

Виды классов, индексаторы и свойства

```
using System;
class CommandLineApp {
    public static void Main(string[] args)    {
        foreach (string arg in args)         {
            Console.WriteLine("Аргумент: {0}", arg);
        }
    }
}
```

Результат:

```
D:\projects\CommandLine>
CommandLineApp 100 200
Аргумент: 100
Аргумент: 200
```

```
public static int Main()
{
    // Вернуть некоторое значение типа int,
    // представляющее код завершения.
    return 0;
}
```

Статические члены класса

```
using System;
class InstCount{
 static int instanceCount;
    //instanceCount = 0;
    public InstCount() {
        instanceCount++;
    }

  class AppClass {
    public static void Main() {
     Console.WriteLine(InstCount.instanceCount);
     InstCount ic1 = new InstCount();
     Console.WriteLine(InstCount.instanceCount);
     InstCount ic2 = new InstCount();
     Console.WriteLine(InstCount.instanceCount);
    }
  }
}
```

Результат:

0

1

2

Константы и неизменяемые поля

```
using System;
class MagicNumbers { //объявление констант
    public const double pi = 3.1415;
    public const int g = 10;
    class ConstApp {
        public static void Main() {
            Console.WriteLine("pi = {0}, g = {1}", MagicNumbers.pi, MagicNumbers.g);
}   }   }
```

```
using System;
class GraphicsPackage {//объявление неизменяемых полей
    public readonly int ScreenWidth;
    public readonly int ScreenHeight;
    public GraphicsPackage()    {
        this.ScreenWidth = 1024;
        this.ScreenHeight = 768;
    }
    class ReadOnlyApp    {
        public static void Main()        {
            GraphicsPackage graphics = new GraphicsPackage();
            Console.WriteLine("Ширина = {0}, Высота = {1}", graphics.ScreenWidth,
graphics.ScreenHeight);
} } }
```

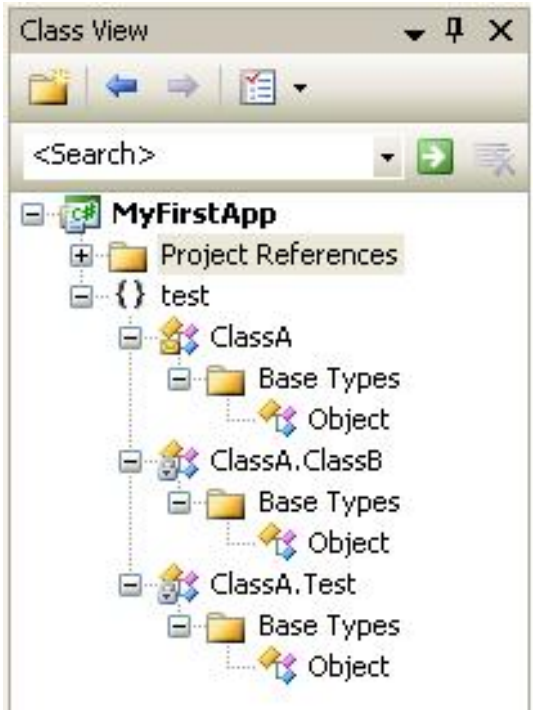
Статические неизменяемые поля

```
using System;
class ScreenResolution {
    //объявление статических неизменяемых полей
    public static readonly int ScreenWidth;
    public static readonly int ScreenHeight;
    static ScreenResolution() {//статический конструктор
        //код для вычисления разрешения экрана
        ScreenWidth = 1024;
        ScreenHeight = 768;
    }
    class ReadOnlyApp {
        public static void Main() {
            Console.WriteLine("Ширина = {0}, Высота = {1}",
ScreenResolution.ScreenMidth, ScreenBesolution.ScreenHeight);
        }
    }
}
```

Вложенные классы

```
class ClassA     {
        //Вложенный класс _Класс _Класс
        private class ClassB {
            public int z;
        }
        //Переменная типа вложенного класса _Класса
        private ClassB w;
        //Конструктор
        public ClassA() {
            w = new ClassB();
            w.z = 35;
        }
        //Некоторый метод
        public int SomeMethod() {
            return w.z;
        }
        class Test{
            static void Main(string[] args) {
                ClassA v = new ClassA();
                int k = v.SomeMethod();
                Console.WriteLine(k);
 } } } }
```

Результат:
35



Наследование

Синтаксис:
class <производный_класс>: <базовый_класс>

```
class Database {
    public Database(){ CommonField = 42; }
    public int CommonField;
    public void CommonMethod() {
        Console.WriteLine("Database.Common Method");
}   }
class SQLServer : Database {
    public void SomeMethodSpecificToSQLServer(){
        Console.WriteLine("SQLServer.SomeMethodSpecificToSQLServer");
}   }
class Oracle : Database {
    public void SomeMethodSpecificToOracle(){
        Console.WriteLine("Oracle.SomeMethodSpecificToOracle");
}   }
class InheritanceApp {
    public static void Main(){
      SQLServer sqlserver = new SQLServer();
      sqlserver.SomeMethodSpecificToSQLServer();
      sqlserver.CommonMethod();
      Console.WriteLine("Inherited common field = {0} ", sqlserver.CommonField);
}   }
```

```
class Worker { //Класс Worker
        protected int age = 0;
        public void setAge(int age) {
            if (age > 0 && age < 100)
                this.age = age;
            else
                this.age = 0;
        }
        public int getAge(){
            return age;
        }
        //Класс Boss
        class Boss : Worker {
            public int numOfWorkers; //Количество подчиненных
            public new void setAge(int age){
                if (age > 0 && age < 45) this.age = age;
                else this.age = 0;
            }
        }
        class Test {
            static void Main(string[] args) {
                Worker wrk1 = new Worker();
                Boss boss = new Boss();
                wrk1.setAge(50); boss.setAge(50);
                boss.numOfWorkers = 4;
                Console.WriteLine("Возраст работника " + wrk1.getAge());
                Console.WriteLine("Возраст босса " + boss.getAge() + " \nКоличество"
                + "подчиненных " + boss.numOfWorkers);
}}}
```

Результат:

Возраст работника 50
Возраст босса 0
Количество
подчиненных 4

Инициализаторы конструкторов

```
class A {
    public A(){
        Console.WriteLine("A");
    }
    public A(int x)
        Console.WriteLine("A = {0}", x);
}  }
class B: A {
    public B(int x){
        Console.WriteLine("B = {0}", x);
}   }


class DerivedInitializerApp {
    public static void Main(){
        B b = new B(42);
}   }
```

Результат:
A
B=42

```
class A {
    public A() {
        Console.WriteLine("A");
    }
    public A(int x) {
        Console.WriteLine("A = {0}", x);
}   }
class B: A {
    public B(int x) : base (x) {
//Активизация конструктора базового класса
        Console.WriteLine("B = {0}", x);
}   }
class DerivedInitializerApp {
    public static void Main(){
        B b = new B(42);
}   }
```

Результат:
A=42
B=42

Использование интерфейсов

- C# не поддерживает множественное наследование. С учетом этого, следующая программа ошибочна:

```
class Foo {}
class Bar {}
class MyClass: Foo, Bar
{ public static void Main () }
```

- Но вы можете объединять характеристики поведения нескольких программных сущностей, реализовав несколько интерфейсов. В этом примере компилятор C# думает, что Bar должен быть интерфейсом. Поэтому компилятор C# выдаст вам сообщение об ошибке:

```
'Bar' -type in interface list is not an interface
```

Следующий пример верен, так как класс MyClass происходит от Control и реализует интерфейсы Foo к Bar:

```
class Control { }
interface IFoo { }
interface IBar { }
class MyClass: Control, IFoo, IBar { }
```

Изолированные классы

```
sealed class MyPoint{
    private int X;
    public int x{
        get{ return this.X; }
        set{ this.X = value; }
    }
    private int Y;
    public int y{
        get{ return this.Y; }
        set{ this.Y = value; }
    }
    public MyPoint(int x, int y) {
        this.x = x; this.y = y;
    }
}
class SealedApp {
    public static void Main(){
        MyPoint pt = new MyPoint(6, 16);
        Console.WriteLine("x = {0}, y = {1}", pt.x, pt.y);
    }}
```

Если класс никогда не будет использован как базовый, при определении класса примените модификатор **sealed**

Результат:
x=6
y=16

Абстрактные классы

Методы класса могут быть объявлены как **абстрактные** (в самом классе нет реализации этих методов).
Абстрактные методы пишутся с модификатором **abstract**. Класс, в котором есть хотя бы один абстрактный метод, называется **абстрактным классом** (в таком классе могут быть и обычные методы).
Нельзя создавать экземпляры абстрактного класса — такой класс может использоваться только в качестве базового класса для других классов.

```
abstract class Figure{//Площадь фигуры
    public abstract double square();
    public abstract double perimeter();
}
class Triangle : Figure{
    double a, b, c; //Стороны
    public Triangle(double a, double b, double c){
       this.a = a; this.b = b; this.c = c;
    }
    public override double square(){
       double p = (a + b + c) / 2;
       return Math.Sqrt(p * (p - a) * (p - b) * (p - c));
    }
    public override double perimeter(){
       return a + b + c;
}   }
class Rectangle : Figure{
    double a, b; //Стороны
    public Rectangle(double a, double b){
        this.a = a; this.b = b;
    }
    public override double square(){
        return a * b;
    }
    public override double perimeter(){
        return (a + b) * 2;
}   }
```

```
class Test{
 public static void Main(){
  Figure f1, f2;
  f1=new Triangle(3,4,5);
  f2=new Rectangle(2,6);
  Console.WriteLine(f1.perimeter()+", "
   + f1.square());
  Console.WriteLine(f2.perimeter() + ", "
   + f2.square());
 }
}
```

Результат:

12, 6

16, 12

Перегрузка

```
class Program{
        static private int Add(int a, int b){
            return a + b;
        }
        static private int Add(int a, int b, int c){
            return a + b + c;
        }
        static private double Add(double a, double b){
            return a + b;
        }
        static private double Add(double a, double b, double c){
            return a + b + c;
        }
        static void Main(string[] args){
            Console.WriteLine(Add(2.4, 2.2, 1.8));
        }}}
```

Перегрузка позволяет выполнить одно и тоже действие (метод) но с разными типами аргументов или их количеством.

Результат:
6,4

Передача параметров по ссылке

```
class Program    {
       static private int Inc(int a) {
            return a += 1;
        }
       static private int Inc2(ref int a) {
            return a += 1;
        }
       static void Main(string[] args)
        {
            int a = 5;
            Console.WriteLine(Inc(a));
            Console.WriteLine(a);

            Console.WriteLine(Inc2(ref a));
            Console.WriteLine(a);
        }
}
```

Обычно при передаче параметров в функцию, передается ***копия*** данного объекта, но бывает ситуация когда необходимо передать не копию а ***сам объект***.
Таким образом внутри метода меняется не копия объекта а сам объект.
Передача параметра по ссылке происходит через ключевое слово **ref**

Результат:
6
5
6
6

Перегрузка родительских методов

```
class car{
    string _name;
    public car(string name) {
        this._name = name;
    }
    public override string ToString() {
        return this._name;
    }
}
class Program{
    static void Main(string[] args){
      car mers = new car("Мерс");
      Console.WriteLine(mers.ToString());
}  }
```

Чаще всего необходимо перегрузить некоторые функции родительского класса.
Рассмотрим как перезагрузить метод ToString() пользовательского класса.
Происходит это с использованием ключевого слова **override**.

Результат:
Мерс

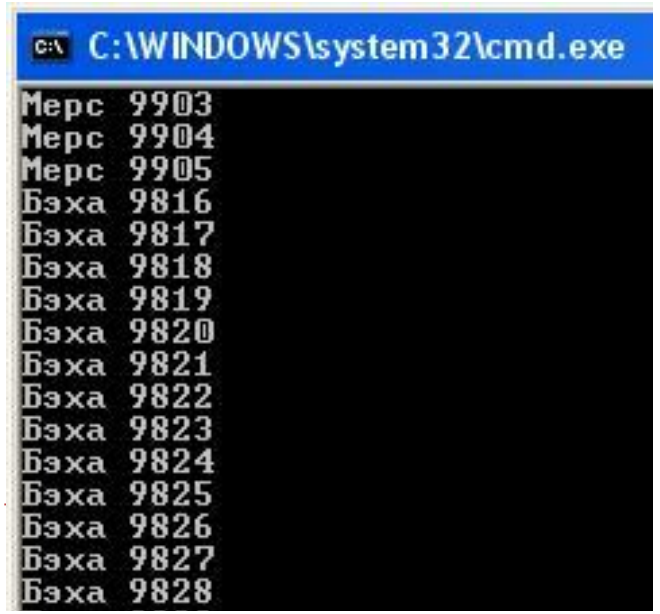
Многопоточность

```
class car{
  string _name;
  public car(string name){ this._name = name; }
  public void Run(){
    for (int i = 0; i < 10000; i++)
      Console.WriteLine(this._name + " " + i);
} }

class Program {
  static void Main(string[] args){
    car mers = new car("Мерс");
    car bmw  = new car("Бэха");
    System.Threading.Thread t1 =
     new System.Threading.Thread(
      new System.Threading.ThreadStart(mers.Run)
     );
    System.Threading.Thread t2 =
     new System.Threading.Thread(
       new System.Threading.ThreadStart(bmw.Run)
     );
    t1.Start(); t2.Start();
} }
```

В C# работа с потоками происходит через класс **Thread**

Сейчас создадим класс который будет выполнять некоторую большую работу запустим два экземпляра данной работы и посмотрим их параллельное выполнение



Делегаты

- В C# **делегат** подобен указателю на функцию в других языках программирования.
  Основное назначение - это обратный вызов (посылка сообщения объектам, которые их породили).
- Синтаксис:

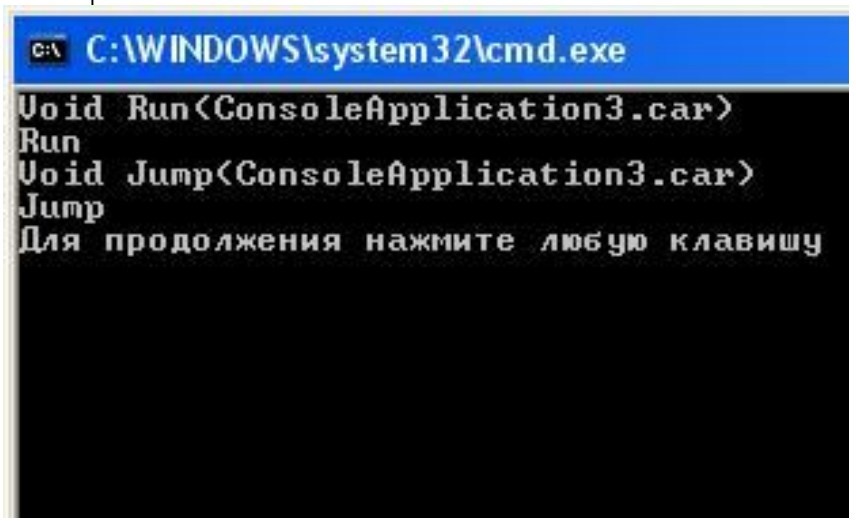
```
public delegate void Play(object music, int volume);
```

- Делегат в момент компиляции превращается в класс, наследуемый от System.MulticastDelegate

Делегаты

```
class garage{
    car c;
    public garage(){
      c = new car("BMW");
    }
    public void ProcCar(car.CarDelegate proc){
      Console.WriteLine(proc.Method.ToString());
      proc(c);
}   }
class car{
    public delegate void CarDelegate(car c);
    string _name;
    public car(string name){
        this._name = name;
}   }
class Program{
    static void Main(string[] args){
        car c = new car("BMW");
        garage gar = new garage();
        gar.ProcCar(new car.CarDelegate(Run));
        gar.ProcCar(new car.CarDelegate(Jump));
    }
    public static void Jump(car c){
        Console.WriteLine("Jump");
    }
    public static void Run(car c){
        Console.WriteLine("Run");
```

Создадим класс Car в котором будет делегат и два метода

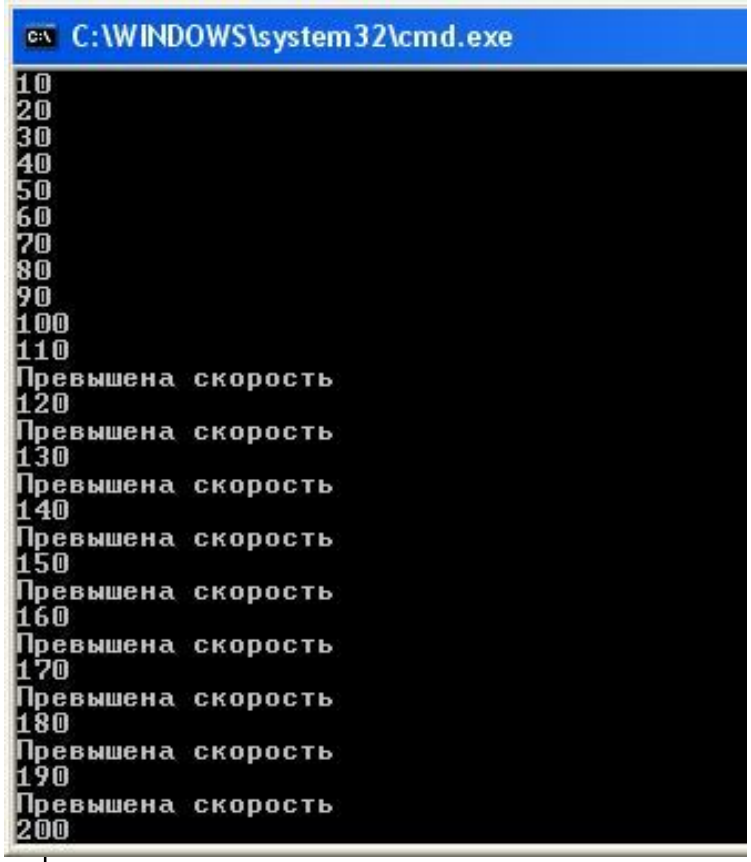


События

```
class car {
  int speed = 0;
  public delegate void CarDelegate(string msg);
  public static event CarDelegate MaxSpeed;
  string _name;
  public car(string name) {
      this._name = name;
  }
  public void SpeedUp() {
      this.speed += 10;
      Console.WriteLine(speed.ToString());
      if (speed > 100)
         MaxSpeed("Превышена скорость");
  }
  public static void Alert(string msg) {
      Console.WriteLine(msg);
  }
}

class Program {
 static void Main(string[] args) {
      car c = new car("BMW");
      car.MaxSpeed += new car.CarDelegate(car.Alert);
      for (int i = 0; i < 30; i++)
          c.SpeedUp();
} }
```

События так же можно использовать и в консольных приложениях



Динамическая идентификация типов

- ***Динамическая идентификация типов (RTTI)*** позволяет определить тип объекта во время выполнения программы.
- По ссылке на базовый класс можно определить тип объекта, доступного по этой ссылке.
- Динамическая идентификация типов позволяет также проверить заранее, насколько удачным будет исход приведения типов, предотвращая исключительную ситуацию в связи с неправильным приведением типов. Кроме того, динамическая идентификация типов является главной составляющей рефлексии.
- Для поддержки динамической идентификации типов в C# предусмотрены три ключевых слова: **is**, **as** и **typeof**. Каждое из этих ключевых слов рассматривается далее по очереди.

Динамическая идентификация типов

Оператор is позволяет определить конкретный тип объекта:

выражение is тип

где выражение обозначает отдельное выражение, описывающее объект, тип которого проверяется. Если выражение имеет совместимый или такой же тип, как и проверяемый тип, то результат этой операции получается истинным, в противном случае — ложным.

Оператор as используется, если преобразование типов требуется произвести во время выполнения, но не генерировать исключение в случае неудачного исхода:

выражение as тип

где выражение обозначает отдельное выражение, преобразуемое в указанный тип.

Оператор typeof позволяет получить информацию о самом типе:

typeof(тип)

где тип обозначает получаемый тип. Информация, описывающая тип, инкапсулируется в возвращаемом объекте класса Type.

Динамическая идентификация типов

```
class Add { }
    class Sum : Add { }

    class Program {
        static void Main() {
            Add a = new Add();
            Sum s = new Sum();

            // Проверяем принадлежность типу
            if (a is Add)
                Console.WriteLine("Переменная a имеет тип Add");
            if (s is Sum)
                Console.WriteLine("Тип переменной s унаследован от класса Add");

            // Выполняем приведение типов
            a = s as Add;
            if (a != null)
                Console.WriteLine("Преобразование прошло успешно");
            else
                Console.WriteLine("Ошибка при преобразовании");

        }
}
```

Результат:

Переменная a имеет тип Add

Тип переменной s унаследован от класса Add

Преобразование прошло успешно

Атрибуты

- **Атрибут** - это некоторая дополнительная информация, которая может быть приписана к типам, полям, методам, свойствам и некоторым другим конструкциям языка. Атрибуты помещаются в исполняемый файл и могут оттуда при необходимости извлекаться.
- Все атрибуты являются классами (потомками класса System.Attribute). Набор атрибутов .NET открыт для дополнения, т.е. вы можете определять собственные атрибуты и применять их к вышеуказанным элементам вашего кода.
- Атрибуты делятся на предопределенные (встроенные) и пользовательские (которые пишет программист).
- Встроенные атрибуты могут использоваться, например, при сериализации (сохранении в поток) данных класса.
- Атрибуты в C# заключаются в квадратные скобки.

Пользовательские атрибуты

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
class TestAttribute : Attribute {//объявление атрибута
     private string name; //поле атрибута
     public TestAttribute (string name){ //конструктор атрибута
        this.name = name;
     }
     public virtual string Name{ //свойство только для чтения
        get{
           return name;
        }
     }
}

[TestAttribute("ATTR1")] //применение атрибута к классу
class Test {
     static void Main(){
        GetAttribute(typeof(Test));
     }
     public static void GetAttribute(Type t){
        TestAttribute att =
          (TestAttribute)Attribute.GetCustomAttribute(t, typeof(TestAttribute));
        Console.WriteLine("{0}", att.Name);
}    }
```

Результат:
ATTR1

Встроенные атрибуты

```
//сначала необходимо добавить через Add - Reference
using System.ComponentModel.DataAnnotations;
…
class Person {
        [DisplayAttribute(Name = "Фамилия")] [RequiredAttribute()]
        public string LastName { get; set; }
        [Display(Name = "Имя")] [Required()]
        public string FirstName { get; set; }
        [Display(Name = "Отчество")]
        public string Patronym { get; set; }
}
class Program {
 static void Main(string[] args) {
   Person p = new Person();
   Console.Write("Фамилия:"); p.LastName = Console.ReadLine();
   Console.Write("Имя:"); p.LastName = Console.ReadLine();
   Console.Write("Отчество:"); p.LastName = Console.ReadLine();
   var t = typeof(Person); var pi = t.GetProperty("FirstName");
   var attrs = (Attribute[])pi.GetCustomAttributes(typeof(Attribute), false);
   foreach (var a in attrs) {
   // Console.WriteLine(a);
   if (a.GetType() == typeof(RequiredAttribute)) Console.Write("!Обязательно: ");
   if (a.GetType() == typeof(DisplayAttribute)) {
         DisplayAttribute da = (DisplayAttribute)a;
         Console.WriteLine(da.Name);
} } }
```

Результат:

```
Фамилия: Штирлиц
Имя: Максим
Отчество: Максимович
!Обязательно: Имя
```

Сериализация

- Для того, чтобы класс стал сериализуемым, достаточно объявить его с атрибутом **Serializable**.

```
namespace Serial
{
    [Serializable()]
    class Worker
    {
        public int age;
        public int id;
    }
}
```

- После этого экземпляр класса можно, например, целиком сохранять в файл и читать из файла (именно экземпляр класса целиком, а не поля класса по отдельности).

Сериализация

```
[Serializable()]
class Worker
{
    public int age;
    public int id;
}
```

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
namespace Serial{
  class Program{
    static void Main(string[] args){
       // Задаем экземпляр класса
       Worker w = new Worker();
       w.age = 22; w.id = 2;
       // Сериализуем класс
       FileStream fs = new FileStream("1.txt", FileMode.Create, FileAccess.Write);
       IFormatter bf = new BinaryFormatter();
       bf.Serialize(fs, w);
       fs.Close();
       // Десериализуем класс
       fs = new FileStream("1.txt", FileMode.Open, FileAccess.Read);
       Worker w1 = (Worker)bf.Deserialize(fs);
       Console.WriteLine("age: " + w1.age + ", id: " + w1.id);
       fs.Close();
}}}
```

Результат:
age: 22, id: 2

Nullable типы

- Как известно, value-типы (к ним относятся, например, int, byte и другие) не могут принимать значение null. Но иногда такая потребность возникает.
- Например, при работе с базами данных - в столбце таблицы могут хранится значения целого типа и, кроме того, могут допускаться неопределенные значения (NULL).
- Nullable-типы могут принимать, кроме собственно значений соотвествующего типа, значение null.
- Пример:

```
Nullable<int> n;
n = null;
```

Упрощенный синтаксис:

```
int? n;
```

С точки зрения реализации все nullable-типы являются экземплярами структуры System.Nullable.

Рефлексия

- В .NET **рефлексией (reflection)** называется процесс обнаружения типов во время выполнения.
- **Сборка (*assembly*)** — двоичный файл, содержащий управляемый код. Когда компилятор платформы .NET создает EXE или DLL модуль, содержимое этого модуля называется сборкой. Сборка содержит в себе: номер версии, метаданные и инструкции Common Intermediate Language (CIL).
- Иногда нам требуется динамически создать некий код. Созданная динамически сборка может существовать только в памяти или же может быть сохраненной на диск в виде файла.
- Сначала мы должны сгенерировать сборку, затем на основании этой сборки - модуль, потом на основании этого модуля - тип (например, класс), потом на основании этого типа (класса) - его члены (конструкторы, методы и т. п.). И, уже в самом конце, мы создаем непосредственно сгенерированный на предыдущих шагах тип.

Рефлексия – создание сборки

```
    // Создание имени сборки.
    AssemblyName an = new AssemblyName("MyAssembly");
    an.Version = new Version("1.0.0.0");
    // Создание сборки.
    AssemblyBuilder ab;
    ab = AppDomain.CurrentDomain.DefineDynamicAssembly(an, AssemblyBuilderAccess.Save);
    // Создание модуля в сборке.
    ModuleBuilder mb = ab.DefineDynamicModule("MyModule", "My.dll");
    // Создание типа в сборке.
    TypeBuilder tb = mb.DefineType("MyClass", TypeAttributes.Public);
    // Создание конструктора без параметров.
    ConstructorBuilder cb0 = tb.DefineConstructor(MethodAttributes.Public,
CallingConventions.Standard, null);
    // Добавление кода для конструктора.
    ILGenerator il0 = cb0.GetILGenerator();
    il0.Emit(OpCodes.Ret);
    // Создание конструктора с параметром типа string.
    ConstructorBuilder cb = tb.DefineConstructor(MethodAttributes.Public,
    CallingConventions.Standard, new Type[] { typeof(string)});
    // Добавление кода для конструктора.
    ILGenerator il = cb.GetILGenerator();
    il.EmitWriteLine("Constructor");
    il.Emit(OpCodes.Ret);
    // Непосредственное создание типа.
    tb.CreateType();
    ab.Save("qqq.dll"); // Сохранение типа в файл.
```

Применение пространства имен

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

```
using ns1;
using ns2;
```

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

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

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

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